

**SeawayEdge内核驱动开发**

**接口说明书**

中科海微（北京）科技有限公司

Seaway Technologies Co. Ltd

修订记录

| **序号** | **版本号** | **修订内容描述** | **修订日期** | **修订人** | **审核人** | **批准人** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | A01 | 第一次正式发布。 |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

目录

[1 驱动程序基础 50](#_Toc2053195512)

[1.1 驱动程序的进入点和退出点 50](#_Toc142246541)

[1.2 驱动设备表 50](#_Toc588315476)

[1.3 延迟、调度和定时器例程 56](#_Toc803494344)

[1.4 等待队列和唤醒事件 79](#_Toc952267272)

[1.5 高分辨率计时器 93](#_Toc1707903060)

[1.6 工作队列和Kevents 104](#_Toc1460303618)

[1.7 内部函数 117](#_Toc1879789810)

[1.8 引用计数 135](#_Toc1995405653)

[1.9 原子操作 141](#_Toc1678178419)

[1.10 内核对象操作 144](#_Toc94468435)

[1.11 内核实用函数 149](#_Toc740571912)

[1.12 void synchronize\_rcu\_bh(void) 166](#_Toc2124390619)

[1.13 设备资源管理 174](#_Toc570018511)

[2 设备驱动的基础架构 186](#_Toc376565110)

[2.1 基本设备驱动模型结构 186](#_Toc295496061)

[2.2 设备驱动库 202](#_Toc1420105363)

[2.3 设备驱动程序DMA管理 247](#_Toc577583183)

[2.4 设备驱动程序PnP支持 251](#_Toc814472241)

[2.5 用户空间 IO 设备 254](#_Toc774188509)

[3 设备电源管理 258](#_Toc182853590)

[3.1 设备电源管理基础知识 258](#_Toc171188273)

[3.2 挂起/休眠通知程序 271](#_Toc1680700978)

[3.3 设备电源管理数据类型 272](#_Toc1688928255)

[4 通用时钟框架 279](#_Toc378335739)

[4.1 介绍和接口分离 279](#_Toc2137170253)

[4.2 常见数据结构和api 279](#_Toc608962449)

[4.3 硬件时钟实现 281](#_Toc2072302388)

[4.4 支持您自己的时钟硬件 282](#_Toc1296448070)

[4.5 禁用未使用时钟的时钟门控制 284](#_Toc1033630028)

[4.6 锁定 284](#_Toc1224660013)

[5 独立于总线的设备访问 286](#_Toc1377565643)

[5.1 介绍 286](#_Toc724563594)

[5.2 内存映射 IO 286](#_Toc1508045868)

[5.3 端口空间访问 289](#_Toc1124901582)

[5.4 提供的公共函数 289](#_Toc1922344133)

[6 设备连接 293](#_Toc2093857863)

[6.1 介绍 293](#_Toc654580052)

[6.2 用法 293](#_Toc2115694030)

[6.3 API 293](#_Toc435335184)

[7 缓冲区共享和同步 295](#_Toc201652159)

[7.1 共享DMA缓冲区 295](#_Toc438641347)

[7.2 保留对象 312](#_Toc2081242525)

[7.3 DMA栅栏 318](#_Toc1229475339)

[8 设备链接 339](#_Toc704371139)

[8.1 用法 339](#_Toc1435870909)

[8.2 限制 340](#_Toc1408626224)

[8.3 示例 340](#_Toc921222240)

[8.4 替代方案 341](#_Toc1772576457)

[8.5 实现 341](#_Toc1799361615)

[8.6 状态机 342](#_Toc1005946251)

[8.7 API 344](#_Toc1947371373)

[9 基于消息的设备 346](#_Toc1819885731)

[9.1 融合消息设备 346](#_Toc209896696)

[10 声音设备 361](#_Toc1565621298)

[10.1 snd\_printk(fmt, ...) 361](#_Toc280028795)

[10.2 snd\_printd（fmt，...） 361](#_Toc1307286988)

[10.3 snd\_BUG（） 361](#_Toc667214859)

[10.4 snd\_printd\_ratelimit（） 361](#_Toc1868014226)

[10.5 snd\_printdd（format，...） 362](#_Toc1651660889)

[10.6 int register\_sound\_special\_device（const struct file\_operations \* fops，int unit，struct device \* dev） 362](#_Toc1090940301)

[10.7 int register\_sound\_mixer（const struct file\_operations \* fops，int dev） 363](#_Toc218260821)

[10.8 int register\_sound\_dsp（const struct file\_operations \* fops，int dev） 363](#_Toc407549471)

[10.9 void unregister\_sound\_special（int unit） 363](#_Toc1358608814)

[10.10 void unregister\_sound\_mixer（int unit） 364](#_Toc2092201994)

[10.11 void unregister\_sound\_dsp（int unit） 364](#_Toc741677180)

[10.12 int snd\_pcm\_stream\_linked（struct snd\_pcm\_substream \* substream） 364](#_Toc1373277072)

[10.13 snd\_pcm\_stream\_lock\_irqsave（substream，flags） 364](#_Toc1660994795)

[10.14 snd\_pcm\_group\_for\_each\_entry（s，substream） 365](#_Toc1199592212)

[10.15 int snd\_pcm\_running（struct snd\_pcm\_substream \* substream） 365](#_Toc969829048)

[10.16 ssize\_t bytes\_to\_samples（struct snd\_pcm\_runtime \* runtime，ssize\_t size） 365](#_Toc515929006)

[10.17 snd\_pcm\_sframes\_t bytes\_to\_frames（struct snd\_pcm\_runtime \* runtime，ssize\_t size） 366](#_Toc1827320903)

[10.18 ssize\_t samples\_to\_bytes（struct snd\_pcm\_runtime \* runtime，ssize\_t size） 366](#_Toc618780974)

[10.19 ssize\_t frames\_to\_bytes（struct snd\_pcm\_runtime \* runtime，snd\_pcm\_sframes\_t size） 366](#_Toc1736011244)

[10.20 int frame\_aligned（struct snd\_pcm\_runtime \* runtime，ssize\_t bytes） 366](#_Toc1428149766)

[10.21 size\_t snd\_pcm\_lib\_buffer\_bytes（struct snd\_pcm\_substream \* substream） 367](#_Toc488394643)

[10.22 size\_t snd\_pcm\_lib\_period\_bytes（struct snd\_pcm\_substream \* substream） 367](#_Toc766266067)

[10.23 snd\_pcm\_uframes\_t snd\_pcm\_playback\_avail（struct snd\_pcm\_runtime \* runtime） 367](#_Toc174357010)

[10.24 snd\_pcm\_uframes\_t snd\_pcm\_capture\_avail（struct snd\_pcm\_runtime \* runtime） 367](#_Toc1250572562)

[10.25 snd\_pcm\_sframes\_t snd\_pcm\_playback\_hw\_avail（struct snd\_pcm\_runtime \* runtime） 368](#_Toc950596345)

[10.26 snd\_pcm\_sframes\_t snd\_pcm\_capture\_hw\_avail（struct snd\_pcm\_runtime \* runtime） 368](#_Toc1541920382)

[10.27 int snd\_pcm\_playback\_ready（struct snd\_pcm\_substream \* substream） 368](#_Toc1370691925)

[10.28 int snd\_pcm\_capture\_ready（struct snd\_pcm\_substream \* substream） 368](#_Toc1162102106)

[10.29 int snd\_pcm\_playback\_data（struct snd\_pcm\_substream \* substream） 369](#_Toc86326077)

[10.30 int snd\_pcm\_playback\_empty（struct snd\_pcm\_substream \* substream） 369](#_Toc1330914414)

[10.31 int snd\_pcm\_capture\_empty（struct snd\_pcm\_substream \* substream） 369](#_Toc488888946)

[10.32 void snd\_pcm\_trigger\_done（struct snd\_pcm\_substream \* substream，struct snd\_pcm\_substream \* master） 370](#_Toc484082000)

[10.33 unsigned int params\_channels（const struct snd\_pcm\_hw\_params \* p） 370](#_Toc1298119164)

[10.34 unsigned int params\_rate（const struct snd\_pcm\_hw\_params \* p） 370](#_Toc1202419475)

[10.35 unsigned int params\_period\_size(const struct snd\_pcm\_hw\_params \*p) 370](#_Toc1242998055)

[10.36 unsigned int params\_periods(const struct snd\_pcm\_hw\_params \*p) 371](#_Toc347392369)

[10.37 unsigned int params\_buffer\_size(const struct snd\_pcm\_hw\_params \*p) 371](#_Toc1762993237)

[10.38 unsigned int params\_buffer\_bytes(const struct snd\_pcm\_hw\_params \*p) 371](#_Toc1795456600)

[10.39 int snd\_pcm\_hw\_constraint\_single(struct snd\_pcm\_runtime \*runtime，snd\_pcm\_hw\_param\_t var，unsigned int val) 371](#_Toc1946352203)

[10.40 int snd\_pcm\_format\_cpu\_endian(snd\_pcm\_format\_t format) 372](#_Toc1870564717)

[10.41 void snd\_pcm\_set\_runtime\_buffer(struct snd\_pcm\_substream \*substream，struct snd\_dma\_buffer \*bufp) 372](#_Toc1568090186)

[10.42 void snd\_pcm\_gettime(struct snd\_pcm\_runtime \*runtime，struct timespec \*tv) 372](#_Toc972440118)

[10.43 int snd\_pcm\_lib\_alloc\_vmalloc\_buffer(struct snd\_pcm\_substream \*substream，size\_t size) 373](#_Toc1450509556)

[10.44 int snd\_pcm\_lib\_alloc\_vmalloc\_32\_buffer(struct snd\_pcm\_substream \*substream，size\_t size) 373](#_Toc479746948)

[10.45 dma\_addr\_t snd\_pcm\_sgbuf\_get\_addr(struct snd\_pcm\_substream \*substream，unsigned int ofs) 373](#_Toc1453344198)

[10.46 void \*snd\_pcm\_sgbuf\_get\_ptr(struct snd\_pcm\_substream \*substream，unsigned int ofs) 374](#_Toc876934808)

[10.47 unsigned int snd\_pcm\_sgbuf\_get\_chunk\_size(struct snd\_pcm\_substream \*substream，unsigned int ofs，unsigned int size) 374](#_Toc463048695)

[10.48 void snd\_pcm\_mmap\_data\_open(struct vm\_area\_struct \*area) 374](#_Toc2126163784)

[10.49 void snd\_pcm\_mmap\_data\_close(struct vm\_area\_struct \*area) 375](#_Toc306831608)

[10.50 void snd\_pcm\_limit\_isa\_dma\_size(int dma，size\_t \*max) 375](#_Toc810599209)

[10.51 const char \*snd\_pcm\_stream\_str(struct snd\_pcm\_substream \*substream) 375](#_Toc104649095)

[10.52 struct snd\_pcm\_substream \*snd\_pcm\_chmap\_substream(struct snd\_pcm\_chmap \*info，unsigned int idx) 375](#_Toc48232772)

[10.53 u64 pcm\_format\_to\_bits(snd\_pcm\_format\_t pcm\_format) 376](#_Toc1046864085)

[10.54 const char \*snd\_pcm\_format\_name(snd\_pcm\_format\_t format) 376](#_Toc311156724)

[10.55 int snd\_pcm\_new\_stream(struct snd\_pcm \*pcm，int stream，int substream\_count) 376](#_Toc488379823)

[10.56 int snd\_pcm\_new(struct snd\_card \* card, const char \* id, int device, int playback\_count, int capture\_count, struct snd\_pcm \*\* rpcm) 377](#_Toc517186327)

[10.57 int snd\_pcm\_new\_internal(struct snd\_card \* card，const char \* id, int device, int playback\_count, int capture\_count, struct snd\_pcm \*\* rpcm) 377](#_Toc1484278480)

[10.58 int snd\_pcm\_notify(struct snd\_pcm\_notify \* notify, int nfree) 378](#_Toc1098369808)

[10.59 int snd\_device\_new(struct snd\_card \* card, enum snd\_device\_type type, void \* device\_data, struct snd\_device\_ops \* ops) 378](#_Toc531933444)

[10.60 void snd\_device\_disconnect(struct snd\_card \* card, void \* device\_data) 379](#_Toc230970847)

[10.61 void snd\_device\_free(struct snd\_card \* card, void \* device\_data) 379](#_Toc1424075400)

[10.62 int snd\_device\_register(struct snd\_card \* card, void \* device\_data) 380](#_Toc730001985)

[10.63 int snd\_info\_get\_line(struct snd\_info\_buffer \* buffer, char \* line, int len) 380](#_Toc569286584)

[10.64 const char \* snd\_info\_get\_str(char \* dest, const char \* src, int len) 380](#_Toc959969903)

[10.65 struct snd\_info\_entry \* snd\_info\_create\_module\_entry(struct module \* module, const char \* name, struct snd\_info\_entry \* parent) 381](#_Toc169519810)

[10.66 struct snd\_info\_entry \* snd\_info\_create\_card\_entry(struct snd\_card \* card, const char \* name, struct snd\_info\_entry \* parent) 381](#_Toc1556130748)

[10.67 void snd\_info\_free\_entry(struct snd\_info\_entry \* entry) 382](#_Toc1833628470)

[10.68 int snd\_info\_register(struct snd\_info\_entry \* entry) 382](#_Toc1403360840)

[10.69 int snd\_rawmidi\_receive(struct snd\_rawmidi\_substream \* substream, const unsigned char \* buffer, int count) 382](#_Toc472742879)

[10.70 int snd\_rawmidi\_transmit\_empty(struct snd\_rawmidi\_substream \* substream) 383](#_Toc1847557100)

[10.71 int \_\_snd\_rawmidi\_transmit\_peek(struct snd\_rawmidi\_substream \* substream, unsigned char \* buffer, int count) 383](#_Toc1426127727)

[10.72 int snd\_rawmidi\_transmit\_peek(struct snd\_rawmidi\_substream \* substream, unsigned char \* buffer, int count) 384](#_Toc863723522)

[10.73 int \_\_snd\_rawmidi\_transmit\_ack(struct snd\_rawmidi\_substream \* substream, int count) 384](#_Toc1759264181)

[10.74 int snd\_rawmidi\_transmit\_ack(struct snd\_rawmidi\_substream \* substream, int count) 384](#_Toc1398238171)

[10.75 int snd\_rawmidi\_transmit(struct snd\_rawmidi\_substream \* substream, unsigned char \* buffer, int count) 385](#_Toc275390876)

[10.76 int snd\_rawmidi\_new(struct snd\_card \* card, char \* id, int device, int output\_count, int input\_count, struct snd\_rawmidi \*\* rrawmidi) 385](#_Toc667193647)

[10.77 void snd\_rawmidi\_set\_ops(struct snd\_rawmidi \* rmidi, int stream, const struct snd\_rawmidi\_ops \* ops) 386](#_Toc1511504142)

[10.78 void snd\_request\_card(int card) 386](#_Toc1266054231)

[10.79 void \* snd\_lookup\_minor\_data(unsigned int minor, int type) 387](#_Toc1305485941)

[10.80 int snd\_register\_device(int type, struct snd\_card \* card, int dev, const struct file\_operations \* f\_ops, void \* private\_data, struct device \* device) 387](#_Toc461788988)

[10.81 int snd\_unregister\_device（struct device \* dev） 388](#_Toc281621058)

[10.82 int copy\_to\_user\_fromio(void \_\_user \* dst，const volatile void \_\_iomem \* src，size\_t count) 388](#_Toc151163818)

[10.83 int copy\_from\_user\_toio(volatile void \_\_iomem \* dst，const void \_\_user \* src，size\_t count) 388](#_Toc137134725)

[10.84 int snd\_pcm\_lib\_preallocate\_free\_for\_all(struct snd\_pcm \* pcm) 389](#_Toc573369844)

[10.85 int snd\_pcm\_lib\_preallocate\_pages(struct snd\_pcm\_substream \* substream，int type，struct device \* data，size\_t size，size\_t max) 389](#_Toc867844019)

[10.86 int snd\_pcm\_lib\_preallocate\_pages\_for\_all(struct snd\_pcm \* pcm，int type，void \* data，size\_t size，size\_t max) 390](#_Toc145496909)

[10.87 struct page \* snd\_pcm\_sgbuf\_ops\_page(struct snd\_pcm\_substream \* substream，unsigned long offset) 390](#_Toc1530159277)

[10.88 int snd\_pcm\_lib\_malloc\_pages(struct snd\_pcm\_substream \* substream，size\_t size) 391](#_Toc1270295714)

[10.89 int snd\_pcm\_lib\_free\_pages(struct snd\_pcm\_substream \* substream) 391](#_Toc1725130371)

[10.90 int snd\_pcm\_lib\_free\_vmalloc\_buffer(struct snd\_pcm\_substream \* substream) 391](#_Toc1089427250)

[10.91 struct page \* snd\_pcm\_lib\_get\_vmalloc\_page(struct snd\_pcm\_substream \* substream，unsigned long offset) 392](#_Toc558216428)

[10.92 void snd\_device\_initialize(struct device \* dev，struct snd\_card \* card) 392](#_Toc1734935300)

[10.93 int snd\_card\_new(struct device \* parent，int idx，const char \* xid，struct module \* module，int extra\_size，struct snd\_card \*\* card\_ret) 392](#_Toc524628134)

[10.94 int snd\_card\_disconnect(struct snd\_card \* card) 393](#_Toc2004677203)

[10.95 void snd\_card\_disconnect\_sync(struct snd\_card \* card) 393](#_Toc738813038)

[10.96 int snd\_card\_free\_when\_closed(struct snd\_card \* card) 394](#_Toc480282712)

[10.97 int snd\_card\_free(struct snd\_card \* card) 394](#_Toc1867995158)

[10.98 void snd\_card\_set\_id(struct snd\_card \* card，const char \* nid) 394](#_Toc1331185013)

[10.99 int snd\_card\_add\_dev\_attr(struct snd\_card \* card，const struct attribute\_group \* group) 395](#_Toc741879045)

[10.100 int snd\_card\_register(struct snd\_card \* card) 395](#_Toc471054833)

[10.101 int snd\_component\_add(struct snd\_card \* card，const char \* component) 395](#_Toc1393855389)

[10.102 int snd\_card\_file\_add(struct snd\_card \* card，struct file \* file) 396](#_Toc1775901447)

[10.103 int snd\_card\_file\_remove(struct snd\_card \* card，struct file \* file) 396](#_Toc1847893723)

[10.104 int snd\_power\_wait(struct snd\_card \* card，unsigned int power\_state) 396](#_Toc641299547)

[10.105 void snd\_dma\_program(unsigned long dma，unsigned long addr，unsigned int size，unsigned short mode) 397](#_Toc101062136)

[10.106 void snd\_dma\_disable(unsigned long dma) 397](#_Toc2039238622)

[10.107 unsigned int snd\_dma\_pointer(unsigned long dma，unsigned int size) 397](#_Toc1791997481)

[10.108 void snd\_ctl\_notify（struct snd\_card \*card，unsigned int mask，struct snd\_ctl\_elem\_id \*id） 398](#_Toc1790997639)

[10.109 struct snd\_kcontrol \*snd\_ctl\_new1（const struct snd\_kcontrol\_new \*ncontrol，void \*private\_data） 398](#_Toc19038674)

[10.110 void snd\_ctl\_free\_one（struct snd\_kcontrol \*kcontrol） 399](#_Toc7930515)

[10.111 int snd\_ctl\_add（struct snd\_card \*card，struct snd\_kcontrol \*kcontrol） 399](#_Toc144179491)

[10.112 int snd\_ctl\_replace（struct snd\_card \*card，struct snd\_kcontrol \*kcontrol，bool add\_on\_replace） 399](#_Toc863151421)

[10.113 int snd\_ctl\_remove（struct snd\_card \*card，struct snd\_kcontrol \*kcontrol） 400](#_Toc733897262)

[10.114 int snd\_ctl\_remove\_id（struct snd\_card \*card，struct snd\_ctl\_elem\_id \*id） 400](#_Toc1612697713)

[10.115 int snd\_ctl\_activate\_id（struct snd\_card \*card，struct snd\_ctl\_elem\_id \*id，int active） 401](#_Toc1219353604)

[10.116 int snd\_ctl\_rename\_id（struct snd\_card \*card，struct snd\_ctl\_elem\_id \*src\_id，struct snd\_ctl\_elem\_id \*dst\_id） 401](#_Toc239579107)

[10.117 struct snd\_kcontrol \*snd\_ctl\_find\_numid（struct snd\_card \*card，unsigned int numid） 402](#_Toc74213224)

[10.118 struct snd\_kcontrol \*snd\_ctl\_find\_id（struct snd\_card \*card，struct snd\_ctl\_elem\_id \*id） 402](#_Toc1761140508)

[10.119 int snd\_ctl\_register\_ioctl(snd\_kctl\_ioctl\_func\_t fcn) 402](#_Toc721411355)

[10.120 int snd\_ctl\_register\_ioctl\_compat(snd\_kctl\_ioctl\_func\_t fcn) 403](#_Toc67972523)

[10.121 int snd\_ctl\_unregister\_ioctl(snd\_kctl\_ioctl\_func\_t fcn) 403](#_Toc2100377504)

[10.122 int snd\_ctl\_unregister\_ioctl\_compat(snd\_kctl\_ioctl\_func\_t fcn) 403](#_Toc708520342)

[10.123 int snd\_ctl\_boolean\_mono\_info(struct snd\_kcontrol \*kcontrol, struct snd\_ctl\_elem\_info \*uinfo) 403](#_Toc304565379)

[10.124 int snd\_ctl\_boolean\_stereo\_info(struct snd\_kcontrol \*kcontrol, struct snd\_ctl\_elem\_info \*uinfo) 404](#_Toc1376794052)

[10.125 int snd\_ctl\_enum\_info(struct snd\_ctl\_elem\_info \*info, unsigned int channels, unsigned int items, const char \*const names) 404](#_Toc641335539)

[10.126 void snd\_pcm\_set\_ops（struct snd\_pcm \* pcm，int direction，const struct snd\_pcm\_ops \* ops） 404](#_Toc705979680)

[10.127 void snd\_pcm\_set\_sync(struct snd\_pcm\_substream \* substream) 405](#_Toc553332085)

[10.128 int snd\_interval\_refine(struct snd\_interval \* i, const struct snd\_interval \* v) 405](#_Toc1248161085)

[10.129 int snd\_interval\_ratnum(struct snd\_interval \* i,unsigned int rats\_count,const struct snd\_ratnum \* rats,unsigned int \* nump,unsigned int \* denp) 406](#_Toc1223091699)

[10.130 int snd\_interval\_list(struct snd\_interval \* i,unsigned int count,const unsigned int \* list,unsigned int mask) 406](#_Toc788716009)

[10.131 int snd\_interval\_ranges(struct snd\_interval \* i,unsigned int count,const struct snd\_interval \* ranges,unsigned int mask) 407](#_Toc1680893979)

[10.132 int snd\_pcm\_hw\_rule\_add（struct snd\_pcm\_runtime \* runtime，unsigned int cond，int var，snd\_pcm\_hw\_rule\_func\_t func，void \* private，int dep，...） 407](#_Toc637728768)

[10.133 int snd\_pcm\_hw\_constraint\_mask64(struct snd\_pcm\_runtime \* runtime，snd\_pcm\_hw\_param\_t var，u\_int64\_t mask) 408](#_Toc216521599)

[10.134 int snd\_pcm\_hw\_constraint\_integer(struct snd\_pcm\_runtime \* runtime，snd\_pcm\_hw\_param\_t var) 408](#_Toc1241216375)

[10.135 int snd\_pcm\_hw\_constraint\_minmax(struct snd\_pcm\_runtime \* runtime，snd\_pcm\_hw\_param\_t var，unsigned int min，unsigned int max) 409](#_Toc467467667)

[10.136 int snd\_pcm\_hw\_constraint\_list(struct snd\_pcm\_runtime \* runtime，unsigned int cond，snd\_pcm\_hw\_param\_t var，const struct snd\_pcm\_hw\_constraint\_list \* l) 409](#_Toc1233898543)

[10.137 int snd\_pcm\_hw\_constraint\_ranges(struct snd\_pcm\_runtime \* runtime，unsigned int cond，snd\_pcm\_hw\_param\_t var，const struct snd\_pcm\_hw\_constraint\_ranges \* r) 410](#_Toc2030716769)

[10.138 int snd\_pcm\_hw\_constraint\_ratnums(struct snd\_pcm\_runtime \* runtime，unsigned int cond，snd\_pcm\_hw\_param\_t var，const struct snd\_pcm\_hw\_constraint\_ratnums \* r) 410](#_Toc299134812)

[10.139 int snd\_pcm\_hw\_constraint\_ratdens(struct snd\_pcm\_runtime \* runtime，unsigned int cond，snd\_pcm\_hw\_param\_t var，const struct snd\_pcm\_hw\_constraint\_ratdens \* r) 411](#_Toc299567657)

[10.140 int snd\_pcm\_hw\_constraint\_msbits(struct snd\_pcm\_runtime \* runtime，unsigned int cond，unsigned int width，unsigned int msbits) 411](#_Toc1131942631)

[10.141 int snd\_pcm\_hw\_constraint\_step(struct snd\_pcm\_runtime \* runtime，unsigned int cond，snd\_pcm\_hw\_param\_t var，unsigned long step) 412](#_Toc2170444)

[10.142 int snd\_pcm\_hw\_constraint\_pow2(struct snd\_pcm\_runtime \* runtime，unsigned int cond，snd\_pcm\_hw\_param\_t var) 412](#_Toc2118913956)

[10.143 int snd\_pcm\_hw\_rule\_noresample(struct snd\_pcm\_runtime \* runtime，unsigned int base\_rate) 413](#_Toc865540291)

[10.144 int snd\_pcm\_hw\_param\_value(const struct snd\_pcm\_hw\_params \* params, snd\_pcm\_hw\_param\_t var, int \* dir) 413](#_Toc81446059)

[10.145 int snd\_pcm\_hw\_param\_first(struct snd\_pcm\_substream \* pcm, struct snd\_pcm\_hw\_params \* params, snd\_pcm\_hw\_param\_t var, int \* dir) 413](#_Toc916830474)

[10.146 int snd\_pcm\_hw\_param\_last(struct snd\_pcm\_substream \* pcm, struct snd\_pcm\_hw\_params \* params, snd\_pcm\_hw\_param\_t var, int \* dir) 414](#_Toc974609293)

[10.147 int snd\_pcm\_lib\_ioctl(struct snd\_pcm\_substream \* substream, unsigned int cmd, void \* arg) 415](#_Toc1400611782)

[10.148 void snd\_pcm\_period\_elapsed(struct snd\_pcm\_substream \* substream) 415](#_Toc1513965307)

[10.149 int snd\_pcm\_add\_chmap\_ctls(struct snd\_pcm \* pcm, int stream, const struct snd\_pcm\_chmap\_elem \* chmap, int max\_channels, unsigned long private\_value, struct snd\_pcm\_chmap \*\* info\_ret) 415](#_Toc1828665093)

[10.150 int snd\_hwdep\_new(struct snd\_card \* card, char \* id, int device, struct snd\_hwdep \*\* rhwdep) 416](#_Toc1735745834)

[10.151 void snd\_pcm\_stream\_lock(struct snd\_pcm\_substream \*substream) 417](#_Toc1262371190)

[10.152 void snd\_pcm\_stream\_unlock(struct snd\_pcm\_substream \*substream) 417](#_Toc1681641617)

[10.153 void snd\_pcm\_stream\_lock\_irq(struct snd\_pcm\_substream \* substream) 417](#_Toc318378752)

[10.154 void snd\_pcm\_stream\_unlock\_irq(struct snd\_pcm\_substream \* substream) 417](#_Toc1609920187)

[10.155 void snd\_pcm\_stream\_unlock\_irqrestore(struct snd\_pcm\_substream \* substream, unsigned long flags) 418](#_Toc1782114356)

[10.156 int snd\_pcm\_stop(struct snd\_pcm\_substream \* substream, snd\_pcm\_state\_t state) 418](#_Toc1041556583)

[10.157 int snd\_pcm\_stop\_xrun(struct snd\_pcm\_substream \* substream) 418](#_Toc1302283784)

[10.158 int snd\_pcm\_suspend(struct snd\_pcm\_substream \* substream) 419](#_Toc330227464)

[10.159 int snd\_pcm\_suspend\_all(struct snd\_pcm \* pcm) 419](#_Toc1035243600)

[10.160 int snd\_pcm\_kernel\_ioctl(struct snd\_pcm\_substream \* substream, unsigned int cmd, void \* arg) 419](#_Toc426677206)

[10.161 int snd\_pcm\_lib\_default\_mmap(struct snd\_pcm\_substream \* substream, struct vm\_area\_struct \* area) 420](#_Toc715903909)

[10.162 int snd\_pcm\_lib\_mmap\_iomem(struct snd\_pcm\_substream \* substream, struct vm\_area\_struct \* area) 420](#_Toc1993608069)

[10.163 void \* snd\_malloc\_pages(size\_t size, gfp\_t gfp\_flags) 421](#_Toc1530955189)

[10.164 void snd\_free\_pages(void \* ptr, size\_t size) 421](#_Toc1762286816)

[10.165 int snd\_dma\_alloc\_pages(int type, struct device \* device, size\_t size, struct snd\_dma\_buffer \* dmab) 421](#_Toc660057088)

[10.166 int snd\_dma\_alloc\_pages\_fallback(int type, struct device \* device, size\_t size, struct snd\_dma\_buffer \* dmab) 422](#_Toc1826441261)

[10.167 void snd\_dma\_free\_pages(struct snd\_dma\_buffer \* dmab) 422](#_Toc867023409)

[11 缓冲库 423](#_Toc1385890168)

[11.1 帧缓冲存储器 423](#_Toc1048418214)

[11.2 帧缓冲区颜色图 424](#_Toc661599063)

[11.3 帧缓冲视频模式数据库 425](#_Toc1972611322)

[11.4 帧缓冲区Macintosh视频模式数据库 430](#_Toc825946468)

[11.5 帧缓冲区字体 432](#_Toc347993468)

[12 电压和电流调节器API 433](#_Toc1128245895)

[12.1 介绍 433](#_Toc148154255)

[12.2 消费驱动接口 433](#_Toc1095016912)

[12.3 稳压器驱动程序接口 434](#_Toc14385194)

[12.4 机器接口 434](#_Toc1253787094)

[12.5 API参考 435](#_Toc1290144494)

[13 工业输入/输出 470](#_Toc316126899)

[13.1 介绍 470](#_Toc270248815)

[13.2 核心元素 470](#_Toc143920300)

[13.3 缓冲 494](#_Toc801895578)

[13.4 触发器 498](#_Toc1999094521)

[13.5 触发的缓冲区 505](#_Toc1399957132)

[13.6 IIO触发缓冲区设置 505](#_Toc1248680992)

[13.7 硬件消费者 507](#_Toc1371234060)

[14 输入子系统 511](#_Toc1683830463)

[14.1 输入核心 511](#_Toc599091475)

[14.2 多点触控库 531](#_Toc1527083189)

[14.3 轮询输入设备 536](#_Toc1110092226)

[14.4 矩阵键盘/键盘 539](#_Toc2129600893)

[14.5 稀疏密钥映射支持 541](#_Toc92264102)

[15 Linux USB API 545](#_Toc199569180)

[15.1 Linux-USB主机端API 545](#_Toc1937235293)

[15.2 适用于Linux的USB Gadget API 642](#_Toc1113997284)

[15.3 USB锚点 681](#_Toc1189917642)

[15.4 USB 批量流 683](#_Toc1578088230)

[15.5 USB 核心回调 684](#_Toc1505841160)

[15.6 USB DMA 686](#_Toc577596225)

[15.7 USB请求块 (URB) 689](#_Toc1033669135)

[15.8 USB电源管理 694](#_Toc1881931362)

[15.9 USB热插拔 706](#_Toc1481248118)

[15.10 系统暂停期间的USB设备持久性 709](#_Toc1706683202)

[15.11 USB错误代码 711](#_Toc285503035)

[15.12 Synopsys DesignWare Core超速USB 3.0控制器 722](#_Toc971041847)

[15.13 编写MUSB胶水层 765](#_Toc1572088976)

[15.14 USB Type-C连接器类 779](#_Toc1608110591)

[15.15 USB3 调试端口 790](#_Toc1433005442)

[16 PCI 支持库 793](#_Toc493362589)

[16.1 unsigned char pci\_bus\_max\_busnr(struct pci\_bus \*bus) 793](#_Toc510672256)

[16.2 int pci\_find\_capability（struct pci\_dev \*dev，int cap） 793](#_Toc1523953180)

[16.3 int pci\_bus\_find\_capability（struct pci\_bus \*bus，unsigned int devfn，int cap） 793](#_Toc43638491)

[16.4 int pci\_find\_next\_ext\_capability（struct pci\_dev \*dev，int start，int cap） 794](#_Toc1140194610)

[16.5 int pci\_find\_ext\_capability（struct pci\_dev \*dev，int cap） 794](#_Toc1254228089)

[16.6 int pci\_find\_next\_ht\_capability（struct pci\_dev \*dev，int pos，int ht\_cap） 795](#_Toc112012871)

[16.7 int pci\_find\_ht\_capability（struct pci\_dev \*dev，int ht\_cap） 795](#_Toc1404648125)

[16.8 struct resource \*pci\_find\_parent\_resource（const struct pci\_dev \*dev，struct resource \*res） 795](#_Toc633305404)

[16.9 struct resource \*pci\_find\_resource（struct pci\_dev \*dev，struct resource \*res） 796](#_Toc1034970496)

[16.10 struct pci\_dev \* pci\_find\_pcie\_root\_port（struct pci\_dev \* dev） 796](#_Toc131585572)

[16.11 int \_\_pci\_complete\_power\_transition(struct pci\_dev \* dev，pci\_power\_t state) 796](#_Toc1798035841)

[16.12 int pci\_set\_power\_state(struct pci\_dev \* dev，pci\_power\_t state) 797](#_Toc198499103)

[16.13 pci\_power\_t pci\_choose\_state(struct pci\_dev \* dev，pm\_message\_t state) 797](#_Toc1132320330)

[16.14 int pci\_save\_state(struct pci\_dev \* dev) 797](#_Toc2055190243)

[16.15 void pci\_restore\_state(struct pci\_dev \* dev) 798](#_Toc1455435753)

[16.16 struct pci\_saved\_state \* pci\_store\_saved\_state（struct pci\_dev \* dev） 798](#_Toc1669961341)

[16.17 int pci\_load\_saved\_state（struct pci\_dev \* dev，struct pci\_saved\_state \* state） 798](#_Toc1576475544)

[16.18 int pci\_load\_and\_free\_saved\_state（struct pci\_dev \* dev，struct pci\_saved\_state \*\* state） 798](#_Toc171231322)

[16.19 int pci\_reenable\_device(struct pci\_dev \* dev) 798](#_Toc256741874)

[16.20 int pci\_enable\_device\_io(struct pci\_dev \* dev) 799](#_Toc766029495)

[16.21 int pci\_enable\_device\_mem(struct pci\_dev \* dev) 799](#_Toc493258700)

[16.22 int pci\_enable\_device(struct pci\_dev \* dev) 799](#_Toc912093480)

[16.23 int pcim\_enable\_device(struct pci\_dev \* pdev) 800](#_Toc816846074)

[16.24 void pcim\_pin\_device(struct pci\_dev \* pdev) 800](#_Toc2016494094)

[16.25 void pci\_disable\_device(struct pci\_dev \*dev) 800](#_Toc1776804551)

[16.26 int pci\_set\_pcie\_reset\_state(struct pci\_dev \*dev, enum pcie\_reset\_state state) 800](#_Toc1993977122)

[16.27 bool pci\_pme\_capable(struct pci\_dev \*dev, pci\_power\_t state) 801](#_Toc1291178019)

[16.28 void pci\_pme\_active(struct pci\_dev \*dev, bool enable) 801](#_Toc506712398)

[16.29 int pci\_enable\_wake(struct pci\_dev \*pci\_dev, pci\_power\_t state, bool enable) 801](#_Toc1542612831)

[16.30 int pci\_wake\_from\_d3(struct pci\_dev \*dev, bool enable) 802](#_Toc123780386)

[16.31 int pci\_prepare\_to\_sleep(struct pci\_dev \*dev) 802](#_Toc1612777206)

[16.32 int pci\_back\_from\_sleep(struct pci\_dev \*dev) 802](#_Toc407908808)

[16.33 bool pci\_dev\_run\_wake(struct pci\_dev \*dev) 803](#_Toc955534832)

[16.34 void pci\_d3cold\_enable(struct pci\_dev \*dev) 803](#_Toc791209158)

[16.35 void pci\_d3cold\_disable(struct pci\_dev \*dev) 803](#_Toc633576282)

[16.36 int pci\_enable\_atomic\_ops\_to\_root(struct pci\_dev \*dev, u32 cap\_mask) 803](#_Toc1292649748)

[16.37 u8 pci\_common\_swizzle(struct pci\_dev \*dev, u8 \*pinp) 804](#_Toc1619741584)

[16.38 void pci\_release\_region(struct pci\_dev \*pdev, int bar) 804](#_Toc1494092916)

[16.39 int pci\_request\_region(struct pci\_dev \*pdev，int bar，const char \*res\_name) 804](#_Toc693354841)

[16.40 void pci\_release\_selected\_regions(struct pci\_dev \*pdev，int bars) 805](#_Toc968544065)

[16.41 int pci\_request\_selected\_regions(struct pci\_dev \*pdev，int bars，const char \*res\_name) 805](#_Toc394056195)

[16.42 void pci\_release\_regions(struct pci\_dev \*pdev) 806](#_Toc62902017)

[16.43 int pci\_request\_regions(struct pci\_dev \*pdev，const char \*res\_name) 806](#_Toc632245395)

[16.44 int pci\_request\_regions\_exclusive(struct pci\_dev \*pdev，const char \*res\_name) 806](#_Toc399268409)

[16.45 int pci\_remap\_iospace(const struct resource \*res，phys\_addr\_t phys\_addr) 807](#_Toc1765236835)

[16.46 void pci\_unmap\_iospace(struct resource \*res) 807](#_Toc848902540)

[16.47 int devm\_pci\_remap\_iospace(struct device \*dev，const struct resource \*res，phys\_addr\_t phys\_addr) 807](#_Toc1771122759)

[16.48 void \_\_iomem \* devm\_pci\_remap\_cfgspace(struct device \*dev，resource\_size\_t offset，resource\_size\_t size) 808](#_Toc989379446)

[16.49 void pci\_set\_master(struct pci\_dev \*dev) 809](#_Toc534470201)

[16.50 void pci\_clear\_master(struct pci\_dev \*dev) 809](#_Toc2064056453)

[16.51 int pci\_set\_cacheline\_size(struct pci\_dev \*dev) 809](#_Toc145971933)

[16.52 int pci\_set\_mwi(struct pci\_dev \*dev) 809](#_Toc923953057)

[16.53 int pcim\_set\_mwi(struct pci\_dev \*dev) 810](#_Toc424777542)

[16.54 int pci\_try\_set\_mwi(struct pci\_dev \*dev) 810](#_Toc1000505766)

[16.55 void pci\_clear\_mwi(struct pci\_dev \*dev) 810](#_Toc703453152)

[16.56 void pci\_intx(struct pci\_dev \*pdev, int enable) 810](#_Toc1039648929)

[16.57 bool pci\_check\_and\_mask\_intx(struct pci\_dev \*dev) 811](#_Toc1452597711)

[16.58 bool pci\_check\_and\_unmask\_intx(struct pci\_dev \*dev) 811](#_Toc1215629681)

[16.59 int pci\_wait\_for\_pending\_transaction(struct pci\_dev \*dev) 811](#_Toc2076114656)

[16.60 bool pcie\_has\_flr(struct pci\_dev \* dev) 811](#_Toc944726936)

[16.61 int pcie\_flr(struct pci\_dev \*dev) 812](#_Toc1679011081)

[16.62 int pci\_reset\_function(struct pci\_dev \*dev) 813](#_Toc1204116787)

[16.63 int pci\_reset\_function\_locked(struct pci\_dev \*dev) 813](#_Toc1852433428)

[16.64 int pci\_try\_reset\_function(struct pci\_dev \*dev) 813](#_Toc1778193837)

[16.65 int pci\_probe\_reset\_slot(struct pci\_slot \*slot) 814](#_Toc1721386807)

[16.66 int pci\_probe\_reset\_bus(struct pci\_bus \*bus) 814](#_Toc448372865)

[16.67 int pci\_reset\_bus(struct pci\_dev \*pdev) 814](#_Toc282624732)

[16.68 int pcix\_get\_max\_mmrbc(struct pci\_dev \*dev) 814](#_Toc1987527207)

[16.69 int pcix\_get\_mmrbc(struct pci\_dev \*dev) 815](#_Toc261638964)

[16.70 int pcix\_set\_mmrbc(struct pci\_dev \*dev, int mmrbc) 815](#_Toc1467042539)

[16.71 int pcie\_get\_readrq(struct pci\_dev \*dev) 815](#_Toc1324201766)

[16.72 int pcie\_set\_readrq(struct pci\_dev \*dev, int rq) 815](#_Toc1486047301)

[16.73 int pcie\_get\_mps(struct pci\_dev \*dev) 816](#_Toc762173297)

[16.74 int pcie\_set\_mps(struct pci\_dev \*dev, int mps) 816](#_Toc106648324)

[16.75 u32 pcie\_bandwidth\_available(struct pci\_dev \*dev, struct pci\_dev \*\*limiting\_dev, enum pci\_bus\_speed \*speed, enum pcie\_link\_width \*width) 816](#_Toc1437019870)

[16.76 enum pci\_bus\_speed pcie\_get\_speed\_cap(struct pci\_dev \*dev) 817](#_Toc1391860928)

[16.77 enum pcie\_link\_width pcie\_get\_width\_cap(struct pci\_dev \*dev) 817](#_Toc467250125)

[16.78 void pcie\_print\_link\_status(struct pci\_dev \*dev) 817](#_Toc1872637443)

[16.79 int pci\_select\_bars(struct pci\_dev \*dev, unsigned long flags) 817](#_Toc2044657716)

[16.80 int pci\_add\_dynid(struct pci\_driver \*drv, unsigned int vendor, unsigned int device, unsigned int subvendor, unsigned int subdevice, unsigned int class, unsigned int class\_mask, unsigned long driver\_data) 818](#_Toc528913518)

[16.81 const struct pci\_device\_id \*pci\_match\_id(const struct pci\_device\_id \*ids, struct pci\_dev \*dev) 819](#_Toc1014682093)

[16.82 int \_\_pci\_register\_driver(struct pci\_driver \*drv, struct module \*owner, const char \*mod\_name) 819](#_Toc594296224)

[16.83 void pci\_unregister\_driver(struct pci\_driver \*drv) 819](#_Toc390194571)

[16.84 struct pci\_driver \*pci\_dev\_driver(const struct pci\_dev \*dev) 820](#_Toc1732580506)

[16.85 struct pci\_dev \*pci\_dev\_get(struct pci\_dev \*dev) 820](#_Toc1749794669)

[16.86 void pci\_dev\_put(struct pci\_dev \*dev) 820](#_Toc1157939865)

[16.87 void pci\_stop\_and\_remove\_bus\_device(struct pci\_dev \*dev) 821](#_Toc998501941)

[16.88 struct pci\_bus \*pci\_find\_bus(int domain, int busnr) 821](#_Toc1384904729)

[16.89 struct pci\_bus \*pci\_find\_next\_bus(const struct pci\_bus \*from) 821](#_Toc1666014117)

[16.90 struct pci\_dev \*pci\_get\_slot(struct pci\_bus \*bus, unsigned int devfn) 822](#_Toc1807474833)

[16.91 struct pci\_dev \*pci\_get\_domain\_bus\_and\_slot(int domain, unsigned int bus, unsigned int devfn) 822](#_Toc2073331416)

[16.92 struct pci\_dev \*pci\_get\_subsys(unsigned int vendor, unsigned int device, unsigned int ss\_vendor, unsigned int ss\_device, struct pci\_dev \*from) 822](#_Toc1411452490)

[16.93 struct pci\_dev \*pci\_get\_device(unsigned int vendor, unsigned int device, struct pci\_dev \*from) 823](#_Toc1177634668)

[16.94 struct pci\_dev \*pci\_get\_class(unsigned int class, struct pci\_dev \*from) 824](#_Toc1296574324)

[16.95 int pci\_dev\_present(const struct pci\_device\_id \*ids) 824](#_Toc1008097359)

[16.96 void pci\_msi\_mask\_irq（struct irq\_data \* data） 824](#_Toc1593821530)

[16.97 void pci\_msi\_unmask\_irq（struct irq\_data \* data） 824](#_Toc1794925679)

[16.98 int pci\_msi\_vec\_count（struct pci\_dev \* dev） 825](#_Toc1613097544)

[16.99 int pci\_msix\_vec\_count（struct pci\_dev \* dev） 825](#_Toc1496862280)

[16.100 int pci\_msi\_enabled（void） 825](#_Toc2140899002)

[16.101 int pci\_enable\_msix\_range（struct pci\_dev \* dev，struct msix\_entry \* entries，int minvec，int maxvec） 825](#_Toc1001021129)

[16.102 int pci\_alloc\_irq\_vectors\_affinity（struct pci\_dev \* dev，unsigned int min\_vecs，unsigned int max\_vecs，unsigned int flags，struct irq\_affinity \* affd） 826](#_Toc775224505)

[16.103 void pci\_free\_irq\_vectors（struct pci\_dev \* dev） 827](#_Toc414969186)

[16.104 int pci\_irq\_vector（struct pci\_dev \* dev，unsigned int nr） 827](#_Toc1507707293)

[16.105 const struct cpumask \* pci\_irq\_get\_affinity（struct pci\_dev \* dev，int nr） 827](#_Toc1876922498)

[16.106 int pci\_irq\_get\_node（struct pci\_dev \* pdev，int vec） 827](#_Toc1049133103)

[16.107 struct irq\_domain \*pci\_msi\_create\_irq\_domain(struct fwnode\_handle \*fwnode, struct msi\_domain\_info \*info, struct irq\_domain \*parent) 828](#_Toc1939320251)

[16.108 int pci\_bus\_alloc\_resource(struct pci\_bus \*bus, struct resource \*res, resource\_size\_t size, resource\_size\_t align, resource\_size\_t min, unsigned long type\_mask, resource\_size\_t (\*alignf)(void \*, const struct resource \*, resource\_size\_t, resource\_size\_t), void \*alignf\_data) 828](#_Toc1796148038)

[16.109 void pci\_bus\_add\_device(struct pci\_dev \*dev) 829](#_Toc682448787)

[16.110 void pci\_bus\_add\_devices(const struct pci\_bus \*bus) 829](#_Toc206604482)

[16.111 struct pci\_ops \*pci\_bus\_set\_ops(struct pci\_bus \*bus, struct pci\_ops \*ops) 829](#_Toc2067955422)

[16.112 void pci\_cfg\_access\_lock(struct pci\_dev \*dev) 830](#_Toc1251434506)

[16.113 bool pci\_cfg\_access\_trylock(struct pci\_dev \*dev) 830](#_Toc404903624)

[16.114 void pci\_cfg\_access\_unlock(struct pci\_dev \*dev) 830](#_Toc1987014872)

[16.115 enum pci\_lost\_interrupt\_reason pci\_lost\_interrupt(struct pci\_dev \*pdev) 830](#_Toc240759207)

[16.116 int pci\_request\_irq(struct pci\_dev \*dev, unsigned int nr, irq\_handler\_t handler, irq\_handler\_t thread\_fn, void \*dev\_id, const char \*fmt, ...) 831](#_Toc580801101)

[16.117 void pci\_free\_irq(struct pci\_dev \*dev, unsigned int nr, void \*dev\_id) 831](#_Toc1210928892)

[16.118 bool pcie\_relaxed\_ordering\_enabled(struct pci\_dev \*dev) 832](#_Toc379365225)

[16.119 int pci\_scan\_slot(struct pci\_bus \*bus, int devfn) 832](#_Toc112388632)

[16.120 unsigned int pci\_scan\_child\_bus（struct pci\_bus \* bus） 832](#_Toc1277612311)

[16.121 unsigned int pci\_rescan\_bus（struct pci\_bus \* bus） 833](#_Toc141124624)

[16.122 struct pci\_slot \* pci\_create\_slot（struct pci\_bus \* parent，int slot\_nr，const char \* name，struct hotplug\_slot \* hotplug） 833](#_Toc1059609280)

[16.123 void pci\_destroy\_slot（struct pci\_slot \* slot） 834](#_Toc1918768036)

[16.124 void pci\_hp\_create\_module\_link（struct pci\_slot \* pci\_slot） 834](#_Toc2119937700)

[16.125 void pci\_hp\_remove\_module\_link（struct pci\_slot \* pci\_slot） 834](#_Toc891736523)

[16.126 int pci\_enable\_rom（struct pci\_dev \* pdev） 835](#_Toc127369648)

[16.127 void pci\_disable\_rom（struct pci\_dev \* pdev） 835](#_Toc1807961524)

[16.128 void \_\_iomem \* pci\_map\_rom（struct pci\_dev \* pdev，size\_t \* size） 835](#_Toc1663212465)

[16.129 void pci\_unmap\_rom（struct pci\_dev \* pdev，void \_\_iomem \* rom） 836](#_Toc1964749903)

[16.130 void \_\_iomem \* pci\_platform\_rom（struct pci\_dev \* pdev，size\_t \* size） 836](#_Toc1843063449)

[16.131 int pci\_enable\_sriov(struct pci\_dev \*dev, int nr\_virtfn) 836](#_Toc1063263015)

[16.132 void pci\_disable\_sriov(struct pci\_dev \*dev) 836](#_Toc1050066418)

[16.133 int pci\_num\_vf(struct pci\_dev \*dev) 837](#_Toc445676280)

[16.134 int pci\_vfs\_assigned(struct pci\_dev \*dev) 837](#_Toc58277224)

[16.135 int pci\_sriov\_set\_totalvfs(struct pci\_dev \*dev, u16 numvfs) 837](#_Toc212760736)

[16.136 int pci\_sriov\_get\_totalvfs(struct pci\_dev \*dev) 838](#_Toc309417697)

[16.137 int pci\_sriov\_configure\_simple(struct pci\_dev \*dev, int nr\_virtfn) 838](#_Toc1325324092)

[16.138 ssize\_t pci\_read\_legacy\_io(struct file \*filp, struct kobject \*kobj, struct bin\_attribute \*bin\_attr, char \*buf, loff\_t off, size\_t count) 838](#_Toc1021627560)

[16.139 ssize\_t pci\_write\_legacy\_io(struct file \*filp, struct kobject \*kobj, struct bin\_attribute \*bin\_attr, char \*buf, loff\_t off, size\_t count) 839](#_Toc1362643155)

[16.140 int pci\_mmap\_legacy\_mem(struct file \*filp, struct kobject \*kobj, struct bin\_attribute \*attr, struct vm\_area\_struct \*vma) 839](#_Toc1177894477)

[16.141 int pci\_mmap\_legacy\_io(struct file \*filp, struct kobject \*kobj, struct bin\_attribute \*attr, struct vm\_area\_struct \*vma) 840](#_Toc1368216893)

[16.142 void pci\_adjust\_legacy\_attr(struct pci\_bus \*b, enum pci\_mmap\_state mmap\_type) 840](#_Toc366428575)

[16.143 void pci\_create\_legacy\_files(struct pci\_bus \*b) 841](#_Toc1729444076)

[16.144 int pci\_mmap\_resource(struct kobject \*kobj, struct bin\_attribute \*attr, struct vm\_area\_struct \*vma, int write\_combine) 841](#_Toc575423187)

[16.145 void pci\_remove\_resource\_files(struct pci\_dev \*pdev) 841](#_Toc1018641468)

[16.146 int pci\_create\_resource\_files(struct pci\_dev \*pdev) 842](#_Toc567518792)

[16.147 ssize\_t pci\_write\_rom(struct file \*filp, struct kobject \*kobj, struct bin\_attribute \*bin\_attr, char \*buf, loff\_t off, size\_t count) 842](#_Toc1313460817)

[16.148 ssize\_t pci\_read\_rom(struct file \*filp, struct kobject \*kobj, struct bin\_attribute \*bin\_attr, char \*buf, loff\_t off, size\_t count) 843](#_Toc1351543806)

[16.149 void pci\_remove\_sysfs\_dev\_files(struct pci\_dev \*pdev) 843](#_Toc1462213123)

[17 PCI 热插拔支持库 844](#_Toc1760585640)

[17.1 int \_\_pci\_hp\_register(struct hotplug\_slot \*slot, struct pci\_bus \*bus, int devnr, const char \*name, struct module \*owner, const char \*mod\_name) 844](#_Toc2133163114)

[17.2 int \_\_pci\_hp\_initialize(struct hotplug\_slot \*slot,struct pci\_bus \*bus,int devnr,const char \*name,struct module \*owner,const char \*mod\_name) 844](#_Toc1980453980)

[17.3 int pci\_hp\_add(struct hotplug\_slot \*slot) 845](#_Toc1640997007)

[17.4 void pci\_hp\_deregister(struct hotplug\_slot \*slot) 845](#_Toc104218228)

[17.5 void pci\_hp\_del(struct hotplug\_slot \*slot) 845](#_Toc1396585691)

[17.6 void pci\_hp\_destroy(struct hotplug\_slot \*slot) 846](#_Toc419446927)

[17.7 int pci\_hp\_change\_slot\_info(struct hotplug\_slot \* slot, struct hotplug\_slot\_info \* info) 846](#_Toc1603172635)

[18 串行外设接口 (SPI) 847](#_Toc45157536)

[18.1 struct spi\_statistics 847](#_Toc900980161)

[18.2 struct spi\_device 849](#_Toc866370930)

[18.3 struct spi\_driver 851](#_Toc1157093850)

[18.4 void spi\_unregister\_driver（struct spi\_driver \* sdrv） 852](#_Toc1811913365)

[18.5 module\_spi\_driver 852](#_Toc1509811095)

[18.6 struct spi\_controller 852](#_Toc728300713)

[18.7 spi\_resstruct 859](#_Toc2040779138)

[18.8 spi\_transferstruct 859](#_Toc1913646129)

[18.9 struct spi\_message 862](#_Toc1935392631)

[18.10 void spi\_message\_init\_with\_transfers(struct spi\_message \*m, struct spi\_transfer \*xfers, unsigned int num\_xfers) 863](#_Toc209148108)

[18.11 struct spi\_replaced\_transfers 864](#_Toc1869004664)

[18.12 int spi\_sync\_transfer(struct spi\_device \*spi, struct spi\_transfer \*xfers, unsigned int num\_xfers) 865](#_Toc1118083179)

[18.13 int spi\_write(struct spi\_device \*spi, const void \*buf, size\_t len) 865](#_Toc1142078203)

[18.14 int spi\_read(struct spi\_device \*spi, void \*buf, size\_t len) 866](#_Toc699520935)

[18.15 ssize\_t spi\_w8r8(struct spi\_device \*spi, u8 cmd) 866](#_Toc1522870867)

[18.16 ssize\_t spi\_w8r16(struct spi\_device \*spi, u8 cmd) 866](#_Toc1180556723)

[18.17 ssize\_t spi\_w8r16be(struct spi\_device \*spi, u8 cmd) 867](#_Toc1015428828)

[18.18 struct spi\_board\_info 867](#_Toc259769487)

[18.19 int spi\_register\_board\_info(struct spi\_board\_info const \*info, unsigned n) 869](#_Toc111513658)

[18.20 int \_\_spi\_register\_driver(struct module \*owner, struct spi\_driver \*sdrv) 869](#_Toc1604309822)

[18.21 struct spi\_device \*spi\_alloc\_device(struct spi\_controller \*ctlr) 870](#_Toc1977990269)

[18.22 int spi\_add\_device(struct spi\_device \*spi) 870](#_Toc1035595523)

[18.23 struct spi\_device \*spi\_new\_device(struct spi\_controller \*ctlr, struct spi\_board\_info \*chip) 871](#_Toc2046479773)

[18.24 void spi\_unregister\_device(struct spi\_device \*spi) 871](#_Toc1087454459)

[18.25 void spi\_finalize\_current\_transfer(struct spi\_controller \*ctlr) 871](#_Toc1761256443)

[18.26 struct spi\_message \*spi\_get\_next\_queued\_message(struct spi\_controller \*ctlr) 872](#_Toc522447253)

[18.27 void spi\_finalize\_current\_message(struct spi\_controller \*ctlr) 872](#_Toc1857832235)

[18.28 int spi\_slave\_abort(struct spi\_device \*spi) 872](#_Toc174146265)

[18.29 struct spi\_controller \*\_\_spi\_alloc\_controller(struct device \*dev, unsigned int size, bool slave) 872](#_Toc2003548641)

[18.30 struct spi\_controller \*\_\_devm\_spi\_alloc\_controller(struct device \*dev, unsigned int size, bool slave) 873](#_Toc1098424327)

[18.31 int spi\_register\_controller(struct spi\_controller \*ctlr) 874](#_Toc1448234277)

[18.32 int devm\_spi\_register\_controller（struct device \* dev，struct spi\_controller \* ctlr） 874](#_Toc893838441)

[18.33 void spi\_unregister\_controller（struct spi\_controller \* ctlr） 875](#_Toc1094567122)

[18.34 struct spi\_controller \* spi\_busnum\_to\_master（u16 bus\_num） 875](#_Toc1044699252)

[18.35 void \* spi\_res\_alloc（struct spi\_device \* spi，spi\_res\_release\_t release，size\_t size，gfp\_t gfp） 876](#_Toc434030492)

[18.36 void spi\_res\_free（void \* res） 876](#_Toc1896013832)

[18.37 void spi\_res\_add（struct spi\_message \* message，void \* res） 876](#_Toc1942120238)

[18.38 void spi\_res\_release（struct spi\_controller \* ctlr，struct spi\_message \* message） 876](#_Toc1610889313)

[18.39 struct spi\_replaced\_transfers \* spi\_replace\_transfers（struct spi\_message \* msg，struct spi\_transfer \* xfer\_first，size\_t remove，size\_t insert，spi\_replaced\_release\_t release，size\_t extradatasize，gfp\_t gfp） 877](#_Toc890345862)

[18.40 int spi\_split\_transfers\_maxsize（struct spi\_controller \* ctlr，struct spi\_message \* msg，size\_t maxsize，gfp\_t gfp） 877](#_Toc376850338)

[18.41 int spi\_setup（struct spi\_device \* spi） 878](#_Toc794355763)

[18.42 int spi\_async（struct spi\_device \* spi，struct spi\_message \* message） 878](#_Toc1978958989)

[18.43 int spi\_async\_locked（struct spi\_device \* spi，struct spi\_message \* message） 879](#_Toc137003387)

[18.44 int spi\_sync（struct spi\_device \* spi，struct spi\_message \* message） 880](#_Toc513455725)

[18.45 int spi\_sync\_locked（struct spi\_device \* spi，struct spi\_message \* message） 880](#_Toc1061076429)

[18.46 int spi\_bus\_lock（struct spi\_controller \* ctlr） 881](#_Toc807337515)

[18.47 int spi\_bus\_unlock（struct spi\_controller \* ctlr） 881](#_Toc1119932859)

[18.48 int spi\_write\_then\_read（struct spi\_device \* spi，const void \* txbuf，unsigned n\_tx，void \* rxbuf，unsigned n\_rx） 882](#_Toc17395258)

[19 I2C和SMBus子系统 883](#_Toc304325214)

[19.1 int i2c\_master\_recv（const struct i2c\_client \* client，char \* buf，int count） 883](#_Toc1635308191)

[19.2 int i2c\_master\_recv\_dmasafe（const struct i2c\_client \* client，char \* buf，int count） 884](#_Toc1129051831)

[19.3 int i2c\_master\_send（const struct i2c\_client \* client，const char \* buf，int count） 884](#_Toc808618725)

[19.4 int i2c\_master\_send\_dmasafe（const struct i2c\_client \* client，const char \* buf，int count） 884](#_Toc1178392859)

[19.5 struct i2c\_device\_identity 885](#_Toc1154588579)

[19.6 struct i2c\_driver 886](#_Toc508012961)

[19.7 struct i2c\_client 887](#_Toc1926338702)

[19.8 struct i2c\_board\_info 888](#_Toc511102342)

[19.9 I2C\_BOARD\_INFO 890](#_Toc162473994)

[19.10 struct i2c\_algorithm 890](#_Toc1248701821)

[19.11 struct i2c\_lock\_operations 891](#_Toc1721307063)

[19.12 struct i2c\_timings 892](#_Toc1255599104)

[19.13 struct i2c\_adapter\_quirks 894](#_Toc1679825506)

[19.14 void i2c\_lock\_bus(struct i2c\_adapter \*adapter, unsigned int flags) 895](#_Toc2007255880)

[19.15 int i2c\_trylock\_bus(struct i2c\_adapter \*adapter, unsigned int flags) 895](#_Toc1128964437)

[19.16 void i2c\_unlock\_bus(struct i2c\_adapter \*adapter, unsigned int flags) 895](#_Toc1487271414)

[19.17 bool i2c\_check\_quirks(struct i2c\_adapter \*adap, u64 quirks) 896](#_Toc2008487665)

[19.18 module\_i2c\_driver 896](#_Toc356738462)

[19.19 builtin\_i2c\_driver 896](#_Toc2076472057)

[19.20 int i2c\_register\_board\_info(int busnum, struct i2c\_board\_info const \*info, unsigned len) 896](#_Toc509114602)

[19.21 struct i2c\_client \*i2c\_verify\_client(struct device \*dev) 897](#_Toc1114266166)

[19.22 struct i2c\_client \*i2c\_new\_client\_device(struct i2c\_adapter \*adap, struct i2c\_board\_info const \*info) 897](#_Toc1414050122)

[19.23 void i2c\_unregister\_device(struct i2c\_client \*client) 898](#_Toc1886362752)

[19.24 struct i2c\_client \*i2c\_new\_dummy(struct i2c\_adapter \*adapter, u16 address) 898](#_Toc797692203)

[19.25 struct i2c\_client \* i2c\_new\_secondary\_device(struct i2c\_client \* client, const char \* name, u16 default\_addr) 899](#_Toc72447600)

[19.26 struct i2c\_adapter \*i2c\_verify\_adapter(struct device \*dev) 899](#_Toc3585351)

[19.27 int i2c\_handle\_smbus\_host\_notify(struct i2c\_adapter \*adap, unsigned short addr) 900](#_Toc129452141)

[19.28 int i2c\_add\_adapter(struct i2c\_adapter \*adapter) 900](#_Toc301199376)

[19.29 int i2c\_add\_numbered\_adapter(struct i2c\_adapter \*adap) 900](#_Toc638956453)

[19.30 void i2c\_del\_adapter(struct i2c\_adapter \*adap) 901](#_Toc1522870571)

[19.31 void i2c\_parse\_fw\_timings(struct device \*dev, struct i2c\_timings \*t, bool use\_defaults) 901](#_Toc1175581851)

[19.32 void i2c\_del\_driver(struct i2c\_driver \*driver) 902](#_Toc1154617357)

[19.33 struct i2c\_client \*i2c\_use\_client(struct i2c\_client \*client) 902](#_Toc991684807)

[19.34 void i2c\_release\_client(struct i2c\_client \*client) 902](#_Toc625966882)

[19.35 int \_\_i2c\_transfer(struct i2c\_adapter \*adap, struct i2c\_msg \*msgs, int num) 903](#_Toc102999121)

[19.36 int i2c\_transfer(struct i2c\_adapter \*adap, struct i2c\_msg \*msgs, int num) 903](#_Toc234407165)

[19.37 int i2c\_transfer\_buffer\_flags(const struct i2c\_client \*client, char \*buf, int count, u16 flags) 903](#_Toc1196213557)

[19.38 int i2c\_get\_device\_id(const struct i2c\_client \*client, struct i2c\_device\_identity \*id) 904](#_Toc19349285)

[19.39 void i2c\_put\_dma\_safe\_msg\_buf(u8 \*buf, struct i2c\_msg \*msg, bool xferred) 905](#_Toc933402298)

[19.40 s32 i2c\_smbus\_read\_byte (const struct i2c\_client \*client) 905](#_Toc324381151)

[19.41 s32 i2c\_smbus\_write\_byte（const struct i2c\_client \*client，u8 value） 905](#_Toc1560508771)

[19.42 s32 i2c\_smbus\_read\_byte\_data（const struct i2c\_client \*client，u8 command） 905](#_Toc253133386)

[19.43 s32 i2c\_smbus\_write\_byte\_data（const struct i2c\_client \*client，u8 command，u8 value） 906](#_Toc247713795)

[19.44 s32 i2c\_smbus\_read\_word\_data（const struct i2c\_client \*client，u8 command） 906](#_Toc1502444679)

[19.45 s32 i2c\_smbus\_write\_word\_data（const struct i2c\_client \*client，u8 command，u16 value） 907](#_Toc1474998527)

[19.46 s32 i2c\_smbus\_read\_block\_data（const struct i2c\_client \*client，u8 command，u8 \*values） 907](#_Toc1896505968)

[19.47 s32 i2c\_smbus\_write\_block\_data（const struct i2c\_client \*client，u8 command，u8 length，const u8 \*values） 908](#_Toc1623515402)

[19.48 s32 i2c\_smbus\_xfer（struct i2c\_adapter \*adapter，u16 addr，unsigned short flags，char read\_write，u8 command，int protocol，union i2c\_smbus\_data \*data） 908](#_Toc496142632)

[19.49 s32 i2c\_smbus\_read\_i2c\_block\_data\_or\_emulated（const struct i2c\_client \*client，u8 command，u8 length，u8 \*values） 909](#_Toc2137698370)

[19.50 struct i2c\_client \* i2c\_setup\_smbus\_alert(struct i2c\_adapter \* adapter, struct i2c\_smbus\_alert\_setup \* setup) 909](#_Toc895090280)

[20 高速同步串行接口 (HSI) 911](#_Toc659388725)

[20.1 介绍 911](#_Toc1330682555)

[20.2 在Linux中的HSI子系统 911](#_Toc887002027)

[20.3 hsi-char设备 911](#_Toc11590315)

[20.4 内核HSI API 912](#_Toc1524895975)

[21 错误检测和纠正（EDAC）设备 926](#_Toc856808527)

[21.1 在EDAC子系统中使用的主要概念 926](#_Toc1503060154)

[21.2 内存控制器 927](#_Toc1081868617)

[21.3 PCI控制器 940](#_Toc221806770)

[21.4 EDAC块 943](#_Toc2022255845)

[22 SCSI 接口指南 946](#_Toc1977789493)

[22.1 介绍 946](#_Toc1956120585)

[22.2 SCSI上层 946](#_Toc691520172)

[22.3 SCSI中间层 947](#_Toc198033240)

[22.4 SCSI 较低层 1005](#_Toc1892495477)

[23 libATA开发人员指南 1006](#_Toc791186222)

[23.1 介绍 1006](#_Toc248090930)

[23.2 libata驱动程序API 1006](#_Toc1398501683)

[23.3 错误处理 1011](#_Toc409269766)

[23.4 libata库 1015](#_Toc206835821)

[23.5 libata核心内部 1039](#_Toc1661102701)

[23.6 libata SCSI转换/仿真 1075](#_Toc865684707)

[23.7 ATA 错误和异常 1100](#_Toc361162124)

[23.8 ata\_piix内部结构 1106](#_Toc1263031646)

[23.9 sata\_sil内部 1109](#_Toc2044507374)

[23.10 感谢 1110](#_Toc149599171)

[24 目标和iSCSI接口指南 1111](#_Toc1757400007)

[24.1 介绍和概述 1111](#_Toc131836811)

[24.2 目标核心设备接口 1111](#_Toc1725642420)

[24.3 目标核心传输接口 1111](#_Toc1105500205)

[24.4 目标支持的用户空间I / O 1116](#_Toc113431591)

[24.5 iSCSI辅助函数 1117](#_Toc1626755048)

[24.6 iSCSI引导信息 1123](#_Toc1257781779)

[24.7 iSCSI传输类 1126](#_Toc1856822232)

[24.8 iSCSI TCP接口 1131](#_Toc378895020)

[25 MTD NAND 驱动程序编程接口 1135](#_Toc799587785)

[25.1 介绍 1135](#_Toc1866723216)

[25.2 已知的错误和假设 1135](#_Toc1428492289)

[25.3 文档提示 1135](#_Toc1950211410)

[25.4 基本板载驱动程序 1136](#_Toc160263709)

[25.5 高级板级驱动程序功能 1140](#_Toc607663825)

[25.6 文件系统支持 1150](#_Toc1721165290)

[25.7 工具 1151](#_Toc1020303940)

[25.8 常量 1151](#_Toc591398285)

[25.9 结构 1153](#_Toc1076657679)

[25.10 提供公共函数 1183](#_Toc688401331)

[25.11 提供的内部函数 1202](#_Toc1466763728)

[25.12 贡献者 1240](#_Toc933192583)

[26 并行端口设备 1241](#_Toc1094668440)

[26.1 int parport\_yield(struct pardevice \*dev) 1241](#_Toc600067231)

[26.2 int parport\_yield\_blocking(struct pardevice \*dev) 1241](#_Toc746745105)

[26.3 int parport\_wait\_event（struct parport \*port，signed long timeout） 1241](#_Toc650546667)

[26.4 int parport\_wait\_peripheral（struct parport \*port，unsigned char mask，unsigned char result） 1242](#_Toc898585392)

[26.5 int parport\_negotiate（struct parport \*port，int mode） 1242](#_Toc1419677640)

[26.6 ssize\_t parport\_write（struct parport \*port，const void \*buffer，size\_t len） 1243](#_Toc1978777310)

[26.7 ssize\_t parport\_read（struct parport \*port，void \*buffer，size\_t len） 1243](#_Toc1378491728)

[26.8 long parport\_set\_timeout（struct pardevice \*dev，long inactivity） 1244](#_Toc1256888660)

[26.9 int \_\_parport\_register\_driver（struct parport\_driver \*drv，struct module \*owner，const char \*mod\_name） 1244](#_Toc1878556728)

[26.10 void parport\_unregister\_driver（struct parport\_driver \* drv） 1245](#_Toc598349302)

[26.11 struct parport \* parport\_get\_port（struct parport \* port） 1245](#_Toc1938283460)

[26.12 void parport\_put\_port（struct parport \* port） 1245](#_Toc1550670877)

[26.13 struct parport \* parport\_register\_port（unsigned long base，int irq，int dma，struct parport\_operations \* ops） 1246](#_Toc263889747)

[26.14 void parport\_announce\_port（struct parport \* port） 1246](#_Toc641246774)

[26.15 void parport\_remove\_port（struct parport \* port） 1247](#_Toc1361589972)

[26.16 struct pardevice \* parport\_register\_dev\_model（struct parport \* port，const char \* name，const struct pardev\_cb \* par\_dev\_cb，int id） 1247](#_Toc656916972)

[26.17 void parport\_unregister\_device（struct pardevice \* dev） 1248](#_Toc590119177)

[26.18 struct parport \* parport\_find\_number（int number） 1249](#_Toc1053525993)

[26.19 struct parport \* parport\_find\_base（unsigned long base） 1249](#_Toc608694836)

[26.20 int parport\_claim（struct pardevice \* dev） 1249](#_Toc1869497991)

[26.21 int parport\_claim\_or\_block（struct pardevice \* dev） 1249](#_Toc819495480)

[26.22 void parport\_release（struct pardevice \* dev） 1250](#_Toc1447904149)

[26.23 struct pardevice \* parport\_open（int devnum，const char \* name） 1250](#_Toc1787828086)

[26.24 void parport\_close（struct pardevice \* dev） 1250](#_Toc435452578)

[27 16x50 串口驱动 1251](#_Toc27209470)

[27.1 void uart\_update\_timeout(struct uart\_port \* port, unsigned int cflag, unsigned int baud) 1251](#_Toc2043029126)

[27.2 unsigned int uart\_get\_baud\_rate(struct uart\_port \* port, struct ktermios \* termios, struct ktermios \* old, unsigned int min, unsigned int max) 1251](#_Toc1074488799)

[27.3 unsigned int uart\_get\_divisor(struct uart\_port \* port, unsigned int baud) 1252](#_Toc743257170)

[27.4 void uart\_console\_write（struct uart\_port \* port，const char \* s，unsigned int count，void（\* putchar）（struct uart\_port \*，int） 1252](#_Toc10881591)

[27.5 int uart\_parse\_earlycon（char \* p，unsigned char \* iotype，resource\_size\_t \* addr，char \*\* options） 1252](#_Toc350789942)

[27.6 void uart\_parse\_options（const char \* options，int \* baud，int \* parity，int \* bits，int \* flow） 1253](#_Toc883944179)

[27.7 int uart\_set\_options（struct uart\_port \* port，struct console \* co，int baud，int parity，int bits，int flow） 1254](#_Toc157946507)

[27.8 int uart\_register\_driver（struct uart\_driver \* drv） 1254](#_Toc317155457)

[27.9 void uart\_unregister\_driver（struct uart\_driver \* drv） 1254](#_Toc377353945)

[27.10 int uart\_add\_one\_port（struct uart\_driver \* drv，struct uart\_port \* uport） 1255](#_Toc668544024)

[27.11 int uart\_remove\_one\_port（struct uart\_driver \* drv，struct uart\_port \* uport） 1255](#_Toc584970264)

[27.12 void uart\_handle\_dcd\_change（struct uart\_port \* uport，unsigned int status） 1255](#_Toc415091082)

[27.13 void uart\_handle\_cts\_change（struct uart\_port \* uport，unsigned int status） 1256](#_Toc1408929718)

[27.14 void uart\_insert\_char（struct uart\_port \* port，unsigned int status，unsigned int overrun，unsigned int ch，unsigned int flag） 1256](#_Toc1727078604)

[27.15 void uart\_get\_rs485\_mode（struct device \* dev，struct serial\_rs485 \* rs485conf） 1257](#_Toc1621124576)

[27.16 struct uart\_8250\_port \* serial8250\_get\_port（int line） 1257](#_Toc1115719343)

[27.17 void serial8250\_suspend\_port（int line） 1257](#_Toc67792197)

[27.18 void serial8250\_resume\_port（int line） 1257](#_Toc1217122069)

[27.19 int serial8250\_register\_8250\_port（struct uart\_8250\_port \* up） 1258](#_Toc1388876008)

[27.20 void serial8250\_unregister\_port（int line） 1258](#_Toc1839307213)

[28 脉冲宽度调制 (PWM) 1259](#_Toc209230326)

[28.1 enum pwm\_polarity 1259](#_Toc1103358943)

[28.2 struct pwm\_args 1259](#_Toc632463156)

[28.3 struct pwm\_device 1260](#_Toc1911693889)

[28.4 void pwm\_get\_state(const struct pwm\_device \*pwm, struct pwm\_state \*state) 1261](#_Toc1336349656)

[28.5 void pwm\_init\_state(const struct pwm\_device \*pwm, struct pwm\_state \*state) 1261](#_Toc1644688066)

[28.6 unsigned int pwm\_get\_relative\_duty\_cycle(const struct pwm\_state \*state, unsigned int scale) 1261](#_Toc2010304725)

[28.7 int pwm\_set\_relative\_duty\_cycle(struct pwm\_state \*state, unsigned int duty\_cycle, unsigned int scale) 1262](#_Toc831294824)

[28.8 struct pwm\_ops 1262](#_Toc43499586)

[28.9 struct pwm\_chip 1264](#_Toc953101922)

[28.10 struct pwm\_capture 1265](#_Toc703480081)

[28.11 int pwm\_config( struct pwm\_device \*  pwm , int  duty\_ns , int  period\_ns ) 1265](#_Toc1492244632)

[28.12 int pwm\_set\_polarity( struct pwm\_device \*  pwm , enum pwm\_polarity 极性) 1265](#_Toc1841500358)

[28.13 int pwm\_enable(struct pwm\_device \* pwm) 1266](#_Toc562196342)

[28.14 void pwm\_disable(struct pwm\_device \* pwm) 1266](#_Toc2053356841)

[28.15 int pwm\_set\_chip\_data(struct pwm\_device \* pwm，void \* data) 1266](#_Toc706219397)

[28.16 void \* pwm\_get\_chip\_data(struct pwm\_device \* pwm) 1266](#_Toc287288410)

[28.17 int pwmchip\_add\_with\_polarity(struct pwm\_chip \* chip，enum pwm\_polarity polarity) 1267](#_Toc913068414)

[28.18 int pwmchip\_add(struct pwm\_chip \* chip) 1267](#_Toc22692636)

[28.19 int pwmchip\_remove(struct pwm\_chip \* chip) 1267](#_Toc1290527733)

[28.20 struct pwm\_device \* pwm\_request(int pwm，const char \* label) 1268](#_Toc314773831)

[28.21 struct pwm\_device \* pwm\_request\_from\_chip(struct pwm\_chip \* chip，unsigned int index，const char \* label) 1268](#_Toc1151555056)

[28.22 void pwm\_free(struct pwm\_device \* pwm) 1268](#_Toc1063199428)

[28.23 int pwm\_apply\_state(struct pwm\_device \* pwm，const struct pwm\_state \* state) 1269](#_Toc2128843356)

[28.24 int pwm\_capture(struct pwm\_device \* pwm，struct pwm\_capture \* result，unsigned long timeout) 1269](#_Toc245241625)

[28.25 int pwm\_adjust\_config(struct pwm\_device \* pwm) 1269](#_Toc754872782)

[28.26 struct pwm\_device \* of\_pwm\_get（struct device\_node \* np，const char \* con\_id） 1270](#_Toc1960944245)

[28.27 struct pwm\_device \*pwm\_get(struct device \*dev, const char \*con\_id) 1270](#_Toc158395206)

[28.28 void pwm\_put(struct pwm\_device \*pwm) 1270](#_Toc1415988609)

[28.29 struct pwm\_device \*devm\_pwm\_get(struct device \*dev, const char \*con\_id) 1271](#_Toc106775409)

[28.30 struct pwm\_device \*devm\_of\_pwm\_get(struct device \*dev, struct device\_node \*np, const char \*con\_id) 1271](#_Toc1425453818)

[28.31 void devm\_pwm\_put(struct device \*dev, struct pwm\_device \*pwm) 1272](#_Toc274753194)

[29 W1: Dallas的 1-wire总线 1273](#_Toc687090508)

[29.1 内核内部的W1接口 1273](#_Toc910598037)

[30 RapidIO子系统指南 1292](#_Toc1452739337)

[30.1 介绍 1292](#_Toc1448454216)

[30.2 已知错误和限制 1292](#_Toc295385920)

[30.3 RapidIO驱动程序接口 1292](#_Toc1716449223)

[30.4 内部结构 1319](#_Toc1214260810)

[30.5 赞扬 1350](#_Toc544336229)

[31 编写s390通道设备驱动程序 1352](#_Toc378664583)

[31.1 介绍 1352](#_Toc1221600420)

[31.2 CSS总线 1352](#_Toc1494593620)

[31.3 CCW总线 1352](#_Toc518752381)

[31.4 ccwgroup总线 1380](#_Toc2035144294)

[31.5 通用接口 1385](#_Toc1698103489)

[32 VME 设备驱动程序 1388](#_Toc2115154640)

[32.1 驱动注册 1388](#_Toc2107225689)

[32.2 资源管理 1388](#_Toc1989332346)

[32.3 主窗口 1389](#_Toc535839079)

[32.4 从窗口 1389](#_Toc1448468882)

[32.5 DMA通道 1390](#_Toc541877382)

[32.6 中断 1391](#_Toc2002495994)

[32.7 位置监视器 1391](#_Toc586455374)

[32.8 插槽检测 1392](#_Toc1753014735)

[32.9 总线检测 1392](#_Toc1590497952)

[32.10 虚拟机接口 1392](#_Toc1770125055)

[33 Linux 802.11 驱动程序开发者指南 1411](#_Toc1400837494)

[33.1 介绍 1411](#_Toc1012539597)

[33.2 cfg80211子系统 1411](#_Toc1092587951)

[33.3 mac80211子系统（基础知识） 1513](#_Toc2140510607)

[33.4 mac80211子系统（高级） 1567](#_Toc915717305)

[34 用户空间 I/O HOWTO 1610](#_Toc1592930733)

[34.1 关于本文档 1610](#_Toc1855686029)

[34.2 关于UIO 1610](#_Toc610084022)

[34.3 编写您自己的内核模块 1613](#_Toc1595226976)

[34.4 在用户空间编写驱动程序 1617](#_Toc1793936484)

[34.5 通用PCI UIO驱动程序 1618](#_Toc20082708)

[34.6 通用Hyper-V UIO驱动程序 1621](#_Toc375140777)

[34.7 更多信息 1622](#_Toc2126535094)

[35 Linux固件 API 1623](#_Toc104987837)

[35.1 介绍 1623](#_Toc1446502272)

[35.2 固件API核心功能 1623](#_Toc1848801464)

[35.3 request\_firmware API 1630](#_Toc865317005)

[35.4 其他固件接口 1634](#_Toc623645551)

[36 PINCTRL (引脚控制) 子系统 1639](#_Toc1890578297)

[36.1 顶层接口 1639](#_Toc781396667)

[36.2 引脚组 1641](#_Toc1071280864)

[36.3 引脚配置 1643](#_Toc514584800)

[36.4 与GPIO子系统的交互 1645](#_Toc710087131)

[36.5 PINMUX 接口 1648](#_Toc867784338)

[36.6 什么是pinmuxing？ 1648](#_Toc1289921989)

[36.7 Pinmux约定 1649](#_Toc871452658)

[36.8 引脚多路复用驱动程序 1651](#_Toc666350466)

[36.9 引脚控制与GPIO子系统的交互 1655](#_Toc225062957)

[36.10 GPIO模式陷阱 1656](#_Toc914415932)

[36.11 板/机器配置 1660](#_Toc1195591192)

[36.12 复杂映射 1663](#_Toc296678965)

[36.13 驱动程序的引脚控制请求 1665](#_Toc1973820068)

[36.14 需要pin控制和GPIO的驱动程序 1667](#_Toc1813987667)

[36.15 系统引脚控制占用 1668](#_Toc2012866457)

[36.16 运行时 pinmuxing 1669](#_Toc936651608)

[37 通用输入/输出(GPIO) 1671](#_Toc1248443146)

[37.1 介绍 1671](#_Toc1668723632)

[37.2 GPIO 描述符驱动接口 1673](#_Toc101653204)

[37.3 GPIO 描述符消费者接口 1680](#_Toc1235900263)

[37.4 GPIO 映射 1689](#_Toc1313886457)

[37.5 使用GPIO的子系统驱动程序 1693](#_Toc2062824345)

[37.6 传统 GPIO 接口 1695](#_Toc912769247)

[37.7 核心 1710](#_Toc1437043808)

[37.8 ACPI支持 1744](#_Toc1794186894)

[37.9 设备树支持 1745](#_Toc2081239931)

[37.10 设备管理API 1746](#_Toc1185877981)

[37.11 sysfs助手 1753](#_Toc255498860)

[38 其他设备 1755](#_Toc1349529667)

[38.1 int misc\_register(struct miscdevice \*misc) 1755](#_Toc1970317302)

[38.2 void misc\_deregister(struct miscdevice \*misc) 1755](#_Toc925057974)

[39 DMAEngine文档 1756](#_Toc1815248385)

[39.1 DMAEngine文档 1756](#_Toc1726917413)

[39.2 DMAEngine客户端文档 1763](#_Toc1059471086)

[39.3 DMA 测试文档 1767](#_Toc1743625125)

[39.4 PXA DMA 文档 1769](#_Toc545628913)

[40 Linux内核SLIMbus支持 1773](#_Toc629968101)

[40.1 概述 1773](#_Toc779493797)

[40.2 硬件说明: 1773](#_Toc1301999479)

[41 SoundWire文档 1795](#_Toc1994364270)

[41.1 SoundWire子系统概述 1795](#_Toc1355523514)

[41.2 SoundWire中的音频流 1798](#_Toc1777172422)

[41.3 SoundWire 错误处理 1804](#_Toc1734334078)

[41.4 SoundWire锁 1805](#_Toc1157308215)

[42 FPGA子系统 1808](#_Toc1119778626)

[42.1 介绍 1808](#_Toc1720168521)

[42.2 FPGA管理器 1809](#_Toc1447476533)

[42.3 FPGA桥 1821](#_Toc1043336915)

[42.4 FPGA区域 1826](#_Toc1159552650)

# **驱动程序基础**

## 驱动程序的进入点和退出点

### module\_init

module\_init (x)

驱动程序初始化进入点

**参数**

x——在内核引导时或模块插入时运行的函数

**说明**

module\_init()将在do\_initcalls()期间（如果是内置的）或在模块插入时（如果是模块）调用。每个模块只能有一个。module\_exit

### module\_exit (x)

驱动程序退出进入点

**参数**

x——当驱动程序被移除时运行的函数

**说明**

当驱动程序是模块时，且module\_exit（）与rmmod一起使用时，将使用cleanup\_module（）包装驱动程序清理代码。如果驱动程序是静态编译到内核中的，那么module\_exit（）将不起作用。每个模块只能有一个。

## 驱动设备表

### struct usb\_device\_id

为探测和热插拔 USB 设备提供标识

**定义**

struct usb\_device\_id {

\_\_u16 match\_flags;

\_\_u16 idVendor;

\_\_u16 idProduct;

\_\_u16 bcdDevice\_lo;

\_\_u16 bcdDevice\_hi;

\_\_u8 bDeviceClass;

\_\_u8 bDeviceSubClass;

\_\_u8 bDeviceProtocol;

\_\_u8 bInterfaceClass;

\_\_u8 bInterfaceSubClass;

\_\_u8 bInterfaceProtocol;

\_\_u8 bInterfaceNumber;

kernel\_ulong\_t driver\_info ;

};

**成员**

**match\_flags**

用于对新设备设置匹配的位掩码，可控制除 driver\_info 以外的任何字段，尽管某些字段仅在与其他字段共同使用时才有意义。这通常由 USB\_DEVICE\_\*() 宏设置，该宏设置此结构中除 driver\_info 以外的所有其他字段。

**idVendor**

设备的USB供应商ID；数字由USB论坛分配给其会员。

**idProduct**

供应商分配的产品ID。

**bcdDevice\_lo**

供应商分配的产品版本号范围的低端。这也用于标识由单个设备组成的范围内的各个产品版本。

**bcdDevice\_hi**

版本号范围的高端。产品版本的范围是包含的。

**bDeviceClass**

设备类别；数字由 USB 论坛分配。产品可以选择实现类别，或者是供应商特定的。设备类别指定设备上所有接口的行为。

**bDeviceSubClass**

设备子类别；与 bDeviceClass 相关联。

**bDeviceProtocol**

设备协议；与 bDeviceClass 相关联。

**bInterfaceClass**

接口类别；数字由 USB 论坛分配。产品可以选择实现类别，或者是供应商特定的。接口类别仅指定给定接口的行为；其他接口可能支持其他类别

**bInterfaceSubClass**

接口子类别；与 bInterfaceClass 相关联。

**bInterfaceProtocol**

接口协议；与 bInterfaceClass 相关联。

**bInterfaceNumbe**

接口编号；复合设备可以使用固定接口编号来区分供应商特定的接口

**driver**\_**info**

保存驱动程序使用的信息。通常它保存指向驱动程序理解的描述符的指针，或者可能是设备标志。

**说明**

在大多数情况下，驱动程序将使用USB\_DEVICE()或类似的用于此目的的宏来创建设备ID表。它们将使用MODULE\_DEVICE\_TABLE()将其导出到用户空间，并通过其usb\_driver结构提供给USB核心。

有关如何执行匹配的信息，请参见usb\_match\_id()函数。简而言之，您通常会使用几个宏来帮助构造这些条目。您提供的每个条目都将标识一个或多个特定的产品，或者标识一类已同意采取相同行为的产品。您应该将更具体的匹配项放在表的开头，以便driver\_info记录特定产品的特点。

### **struct mdio\_device\_id**

在MDIO/MII总线上标识PHY设备

**定义**

struct mdio\_device\_id {

\_\_u32 phy\_id;

\_\_u32 phy\_id\_mask;

};

**成员**

**phy**\_**id**

此PHY类型的(mdio\_read(MII\_PHYSID1)<<16|mdio\_read(MII\_PHYSID2)) & phy\_id\_mask的结果

**phy\_id\_mask**

定义phy\_id的重要位。值0被用于终止struct mdio\_device\_id数组。

### struct amba\_id

标识AMBA总线上的设备

**定义**

struct amba\_id {

unsigned int id;

unsigned int mask;

void \*data;

};

**成员**

**id**

硬件设备 ID 的有效位

**mask**

指定匹配时id字段的哪些位是有效的位掩码。当 ((hardware 设备ID) & mask) == id 时，将驱动程序与设备进行绑定。

**data**

驱动程序使用的私有数据。

### struct mips\_cdmm\_device\_id

标识 MIPS CDMM 总线上的设备

**定义**

struct mips\_cdmm\_device\_id {

\_\_u8 type;

};

**成员**

**type**

设备类型标识符

### struct mei\_cl\_device\_id

MEI 客户端设备标识符

**定义**

struct mei\_cl\_device\_id {

char name[MEI\_CL\_NAME\_SIZE];

uuid\_le uuid;

\_\_u8 version;

kernel\_ulong\_t driver\_info;

};

**成员**

**name**

帮助器名称

**uuid**

客户端uuid

**version**

客户端协议版本

**driver\_info**

驱动程序使用的信息

**说明**

通过 uuid 和名称对mei 客户端设备进行标识

### struct rio\_device\_id

RIO 设备标识符

#### 定义

struct rio\_device\_id {

\_\_u16 did,vid;

\_\_u16 asm\_did,asm\_vid;

};

**成员**

**did**

RapidIO 设备ID

**vid**

RapidIO 供应商ID

**asm\_did**

RapidIO集成设备ID

**asm\_vid**

RapidIO 集成供应商ID

**说明**

根据设备/供应商 ID 和集成设备/集成供应商 ID 标识 RapidIO 设备。

### struct fsl\_mc\_device\_id

MC对象设备标识符

**定义**

struct fsl\_mc\_device\_id {

\_\_u16 vendor;

const char obj\_type[16];

};

**成员**

**vendor**

供应商ID

**obj\_type**

MC 对象类型

**说明**

MC 对象设备驱动程序支持的“设备 ID”表中条目的类型。表的最后一个条目将供应商设置为 0x0。

### struct tb\_service\_id

Thunderbolt服务标识符

**定义**

struct tb\_service\_id {

\_\_u32 match\_flags;

char protocol\_key[8 + 1];

\_\_u32 protocol\_id;

\_\_u32 protocol\_version;

\_\_u32 protocol\_revision;

kernel\_ulong\_t driver\_data;

};

**成员**

**match\_flags**

用于匹配此结构的标志

**protocol\_key**

服务支持的协议密钥

**protocol\_id**

服务支持的协议 ID

**protocol\_version**

协议版本

**protocol\_revision**

协议软件的修订版

**driver\_data**

驱动程序特定数据

**说明**

Thunderbolt XDomain 服务是携带服务支持协议信息的设备。Thunderbolt XDomain 服务驱动程序会匹配该信息。

### struct typec\_device\_id

USB Type-C备用模式标识符

**定义**

struct typec\_device\_id {

\_\_u16 svid;

\_\_u8 mode;

kernel\_ulong\_t driver\_data;

};

**成员**

**svid**

标准或供应商ID

**mode**

模式索引

**driver\_data**

驱动程序特定数据

## 延迟、调度和定时器例程

### struct prev\_cputime

系统和用户CPU时间的快照

**定义**

struct prev\_cputime {

#ifndef CONFIG\_VIRT\_CPU\_ACCOUNTING\_NATIVE;

u64 utime;

u64 stime;

raw\_spinlock\_t lock;

#endif;

};

**成员**

**utime**

在用户模式下花费的时间

**stime**

在系统模式下花费的时间

**lock**

保护上述两个字段

**说明**

存储以前的用户/系统时间值，以便我们可以保证单调性。

### struct task\_cputime

收集的CPU时间计数

**定义**

struct task\_cputime {

u64 utime;

u64 stime;

unsigned long long sum\_exec\_runtime;

};

**成员**

utime

在用户模式下花费的时间（以纳秒为单位）

stime

在内核模式下花费的时间（以纳秒为单位）

sum\_exec\_runtime

t在CPU上花费的总时间（以纳秒为单位）

**说明**

这种结构将跟踪线程和线程组的三种CPU时间组合在一起。大多数考虑CPU时间的事情都希望将这些计数组合在一起，并并行处理这三个计数。

### struct util\_est

估计FAIR任务的利用率

**定义**

struct util\_est {

unsigned int enqueued;

unsigned int ewma;

#define UTIL\_EST\_WEIGHT\_SHIFT 2;

};

**成员**

**enqueued**

任务/ cpu的瞬时估计利用率

**ewma**

任务的指数加权移动平均（EWMA）利用率

**说明**

支持数据结构以跟踪FAIR任务利用率的指数加权移动平均（EWMA）。每当任务完成激活时，将新样本添加到移动平均值中。选择样本的权重，以降低EWMA对任务的工作负载瞬时变化的敏感程度。

enqueued属性对于任务和CPU的含义略有不同：任务：任务出队时的util\_avg。cfs\_rq：该CPU上每个RUNNABLE任务的util\_est.enqueued的总和。

因此，任务的util\_est.enqueued表示该任务当前排队的CPU上估计利用率的贡献。仅对任务进行的是过去瞬时估计利用率的移动平均。这允许吸收原本几乎是周期性任务的利用率的零星下降。

### int pid\_alive(const struct task\_struct \*p)

检查任务结构是否过时

**参数**

**const struct task\_struct \*p**

要检查的任务结构

**说明**

测试进程是否尚未死亡（至多为僵死状态）。 如果pid\_alive失败，则任务结构中的指针可能已过时，不得引用。

**返回**

1表示进程仍在运行。否则为0。

### int is\_global\_init(struct task\_struct \*tsk)

检查任务结构是否为init。由于init可以自由拥有子线程，因此需要检查tgid。

**参数**

**struct task\_struct \*tsk**

要检查的任务结构

**说明**

检查任务结构是否为内核创建的第一个用户空间任务

**返回**

1表示任务结构为init。否则为0

### int task\_nice(const struct task\_struct \*p)

返回给定任务的nice值

参数

**const struct task\_struct \*p**

要查询的任务

**返回**

nice值[ -20 … 0 … 19 ]

### bool is\_idle\_task(const struct task\_struct \*p)

指定任务是否为空闲任务?

**参数**

**const struct task\_struct \*p**

要查询的任务

**返回**

如果p是空闲任务，则为1。否则为0

### int wake\_up\_process(struct task\_struct \*p)

唤醒特定进程

**参数**

**struct task\_struct \*p**

被唤醒的进程

**说明**

尝试唤醒指定的进程并将其移动到可运行进程集。

**返回**

如果唤醒了该进程，则为1，如果该进程已经在运行，则为0。

在访问任务状态之前，此函数执行完整的内存屏障。

### void preempt\_notifier\_register(struct preempt\_notifier \*notifier)

告诉我当前正在抢占和重新调度

**参数**

struct preempt\_notifier \*notifier

要注册的通知器结构

### void preempt\_notifier\_unregister(struct preempt\_notifier \*notifier)

不再对抢占通知感兴趣

**参数**

struct preempt\_notifier \*notifier

要注销的通知器结构

**说明**

这不能从抢占通知程序中调用。

### \_\_visible void notrace preempt\_schedule\_notrace(void)

由跟踪调用preempt\_schedule

**参数**

空

无参数

**说明**

跟踪基础设施使用preempt\_enable\_notrace来防止递归和由跟踪基础设施本身引起的跟踪抢占启用。 但是，由于跟踪可以发生在来自用户空间或即将进入用户空间的区域，所以抢占启用可能会在调用user\_exit（）之前发生。 这将导致在系统仍处于用户模式时调用调度程序。

为了防止这种情况发生，preempt\_enable\_notrace会使用此函数而不是preempt\_schedule（）来在需要时退出用户上下文，然后调用调度程序。

### int sched\_setscheduler(struct task\_struct \* p，int policy，const struct sched\_param \* param)

更改线程的调度策略和/或RT优先级。

**参数**

struct task\_struct \* p

要查询的任务。

int policy

新政策。

const struct sched\_param \* param

包含新RT优先级的结构。

**返回**

成功则返回0。否则为错误代码。

注意，任务可能已经死亡。

### sched\_setscheduler\_nocheck（struct task\_struct \* p，int policy，const struct sched\_param \* param）

从内核空间更改线程的调度策略和/或RT优先级。

**参数**

struct task\_struct \* p

有问题的任务。

int policy

新政策。

const struct sched\_param \* param

包含新RT优先级的结构。

**说明**

就像sched\_setscheduler一样，只是不必检查当前上下文是否具有权限。例如，这在stop\_machine（）中是必需的我们创建临时高优先级的工作线程，但调用者可能没有这个能力。

**返回**

成功时为0。否则为错误代码。

### void yield（void）

将当前处理器让给其他线程。

**参数**

void

没有参数

**说明**

永远不要使用此功能，因为你有99％的机会使用不当。

调度程序随时在自由地选择调用任务作为最合适的任务来运行，如果从代码中删除yield（）调用会破坏它，那么它已经被破坏了。

典型的破碎用法是:

while（！event）

yield（）；

其中一个假设yield（）将让“另一个”进程运行，该进程将使事件成为真。如果当前任务是SCHED\_FIFO任务，那就永远不会发生。永远不要使用yield（）作为进度保证！

如果你想使用yield（）等待某些东西，请使用wait\_event（）。如果您想使用yield（）对他人“友好”，请使用cond\_resched（）。如果你仍然想使用yield（），不要使用它！

### int yield\_to（struct task\_struct \* p，bool preempt）

将当前处理器让给线程组中的另一个线程，或加速该线程朝向它所在的处理器。

**参数**

struct task\_struct \* p

目标任务

bool preempt

是否允许任务抢占

**说明**

调用者的工作是确保目标任务结构在我们可以进行任何检查之前不能消失。

**返回**

true（> 0），如果我们确实提升了目标任务。false（0）如果我们未能提升目标。-ESRCH如果没有任务可屈服。

### int cpupri\_find（struct cpupri \* cp，struct task\_struct \* p，struct cpumask \* lowest\_mask）

查找系统中最佳（最低-pri）CPU

**参数**

struct cpupri \* cp

cpupri上下文

struct task\_struct \* p

任务

struct cpumask \* lowest\_mask

要填充所选CPU的掩码（或NULL）

**注**

此函数返回当前调用期间计算的建议CPU。在调用返回时，CPU的优先级可能已经多次更改。虽然不理想，但这不是正确性问题，因为正常的重新平衡逻辑将纠正由于竞争而导致的任何不一致性当前优先级配置的不确定性。

**返回**

（int）bool-找到CPU

### void cpupri\_set（struct cpupri \* cp，int cpu，int newpri）

更新CPU优先级设置

**参数**

struct cpupri \* cp

cpupri上下文

int cpu

目标CPU

int newpri

要分配给此CPU的优先级（INVALID，NORMAL，RT1-RT99，HIGHER）

**注**

假定cpu\_rq（cpu） - > lock已锁定

**返回**

（void）

### int cpupri\_init（struct cpupri \* cp）

初始化cpupri结构

**参数**

struct cpupri \* cp

cpupri上下文

**返回**

-ENOMEM在内存分配失败时。

### void cpupri\_cleanup（struct cpupri \* cp）

清理cpupri结构

**参数**

struct cpupri \* cp

cpupri上下文

### void update\_tg\_load\_avg（struct cfs\_rq \* cfs\_rq，int force）

更新tg的负载平均值

**参数**

struct cfs\_rq \* cfs\_rq

平均值已更改的cfs\_rq

int force

无论差异有多小都要更新

**说明**

此函数“确保”tg->load\_avg= tg->cfs\_rq [] - > avg.load的总和。但是，由于tg->load\_avg是全局值，因此存在性能方面的考虑。

为了避免必须查看其他cfs\_rq，我们使用差分更新，在其中存储我们传播的上一个值。这反过来允许在差分“小”的情况下跳过更新。

在update\_cfs\_share（）之前，必须更新tg的load\_avg。

### int update\_cfs\_rq\_load\_avg（u64 now，struct cfs\_rq \* cfs\_rq）

更新cfs\_rq的负载/工具平均值

**参数**

u64当前时间，按照cfs\_rq\_clock\_pelt()

struct cfs\_rq \* cfs\_rq

要更新的cfs\_rq

**说明**

cfs\_rq平均值是所有实体（阻止和可运行）平均值的直接总和。直接推论是，所有（公平）任务都必须附加。

例如，cfs\_rq-> avg用于task\_h\_load（）和update\_cfs\_share（）。

由于这些条件都表示已更改cfs\_rq->avg.load，因此我们应在此函数返回true时调用update\_tg\_load\_avg（）。

如果负载衰减或我们删除了负载，则返回true。

由于这些条件都表示已更改cfs\_rq->avg.load，因此我们应在此函数返回true时调用update\_tg\_load\_avg（）。

### void attach\_entity\_load\_avg（struct cfs\_rq \* cfs\_rq，struct sched\_entity \* se，int flags）

将此实体附加到其cfs\_rq负载平均值

**参数**

struct cfs\_rq \* cfs\_rq

cfs\_rq 要附加到 struct sched\_entity \*se

sched\_entity 要附加

int flags

迁移提示

**说明**

在此之前必须调用 update\_cfs\_rq\_load\_avg()，因为我们依赖于 cfs\_rq->avg.last\_update\_time 是当前时间。

### void detach\_entity\_load\_avg(struct cfs\_rq \*cfs\_rq, struct sched\_entity \*se)

将此实体从其 cfs\_rq 负载平均值中分离

**参数**

struct cfs\_rq \*cfs\_rq

要从中分离的 cfs\_rq

struct sched\_entity \*se

要分离的 sched\_entity

**说明**

在此之前必须调用 update\_cfs\_rq\_load\_avg()，因为我们依赖于 cfs\_rq->avg.last\_update\_time 是当前时间。

### void cpu\_load\_update(struct rq \* this\_rq, unsigned long this\_load, unsigned long pending\_updates)

更新 rq->cpu\_load[] 统计数据

**参数**

struct rq \* this\_rq

要更新统计数据的 rq

unsigned long this\_load

当前负载

unsigned long pending\_updates

错过的更新数

**说明**

更新 rq->cpu\_load[] 统计数据。 此函数通常在每个调度器时钟滴答声（TICK\_NSEC）调用一次。

此函数计算衰减平均值:

load[i]’ = (1 - 1/2^i) \* load[i] + (1/2^i) \* load

由于 NOHZ，它可能不会在每个时钟滴答声上调用，这就需要使用 pending\_updates 参数。

load[i]\_n = (1 - 1/2^i) \* load[i]\_n-1 + (1/2^i) \* load\_n-1

= A \* load[i]\_n-1 + B ; A := (1 - 1/2^i), B := (1/2^i) \* load = A \* (A \* load[i]\_n-2 + B) + B = A \* (A \* (A \* load[i]\_n-3 + B) + B) + B = A^3 \* load[i]\_n-3 + (A^2 + A + 1) \* B = A^n \* load[i]\_0 + (A^(n-1) + A^(n-2) + ... + 1) \* B = A^n \* load[i]\_0 + ((1 - A^n) / (1 - A)) \* B = (1 - 1/2^i)^n \* (load[i]\_0 - load) + load

在上述中，我们假设 load\_n := load，这对于 NOHZ\_FULL 是正确的，因为负载的任何更改都会导致时钟滴答返回。

对于常规 NOHZ，这缩小为:

load[i]\_n = (1 - 1/2^i)^n \* load[i]\_0

请参见 decay\_load\_misses()。 对于 NOHZ\_FULL，我们可以减去并添加额外的项。

### unsigned long cpu\_util(int cpu)

**参数**

int cpu

要获取利用率的 CPU

**说明**

返回的单位必须是容量的单位，以便我们可以将利用率与可用于 CFS 任务的 CPU 容量（即 cpu\_capacity）进行比较。

cfs\_rq.avg.util\_avg 是可运行任务的运行时间总和加上当前不可运行任务的最近利用率的总和。它表示 CPU 利用率的量在 [0..capacity\_orig] 范围内，其中 capacity\_orig 是在最高频率（arch\_scale\_freq\_capacity()）提供的 cpu\_capacity。 CPU 的利用率趋向于等于或小于 CPU 的当前容量（capacity\_curr <= capacity\_orig），因为它是在此 CPU 上运行时间按 capacity\_curr 缩放的结果。

CPU 的估计利用率被定义为其 cfs\_rq.avg.util\_avg 和当前 RUNNABLE 任务的估计利用率之和中的较大值。 这使得可以正确地表示一个 CPU 的预期利用率，该 CPU 在经过长时间的睡眠期后刚刚运行了一个大任务。 然而，它也保留了“阻塞利用率”的好处，以说明其他任务在同一 CPU 上唤醒的潜力。

不过，由于 cfs\_rq.avg.util\_avg 中的不幸取舍或在迁移任务和新任务唤醒之后直到平均时间稳定为止，cfs\_rq.avg.util\_avg 可能会高于 capacity\_curr，甚至高于 capacity\_orig。 我们需要检查利用率是否保持在 [0..capacity\_orig] 范围内并限制它（如果需要）。 如果没有利用率限制，那么一个组可能被视为过载（CPU0 利用率为 121% + CPU1 利用率为 80%），而 CPU1 具有 20% 的可用容量。 我们允许利用率超过 capacity\_curr（但不是 capacity\_orig），因为它有助于预测任务迁移后所需的容量（调度程序驱动的 DVFS）。

**返回**

指定 CPU 的（估计的）利用率

### int get\_sd\_load\_idx(struct sched\_domain \* sd, enum cpu\_idle\_type idle)

获取给定调度域的负载索引。

**参数**

struct sched\_domain \* sd

要获取其 load\_idx 的调度域。

enum cpu\_idle\_type idle

该 CPU 的空闲状态，其 sd load\_idx 被获取。

**返回**

负载索引。

### void update\_sg\_lb\_stats(struct lb\_env \* env, struct sched\_group \* group, int load\_idx, int local\_group, struct sg\_lb\_stats \* sgs, bool \* overload)

更新负载平衡的 sched\_group 统计数据。

**参数**

struct lb\_env \* env

负载平衡环境。

struct sched\_group \* group

要更新统计数据的 sched\_group。

int load\_idx

此 CPU 的 sched\_domain 的负载索引，用于负载计算。

int local\_group

组是否包含此 CPU。

struct sg\_lb\_stats \* sgs

用于保存此组的统计数据的变量。

bool \* overload

指示任何 CPU 上有多个可运行任务。

### bool update\_sd\_pick\_busiest(struct lb\_env \*env, struct sd\_lb\_stats \*sds, struct sched\_group \*sg, struct sg\_lb\_stats \*sgs)

如果被选中的组比之前选中的最繁忙组更繁忙，则返回1

**参数**

struct lb\_env \*env

负载平衡环境。

struct sd\_lb\_stats \*sds

调度域统计信息

struct sched\_group \*sg

待检查是否为最繁忙组的调度组候选项

struct sg\_lb\_stats \*sgs

调度组统计信息

**说明**

确定sg是否比先前选定的最繁忙组更繁忙。

**返回**

如果sg是比之前选定的最繁忙组更繁忙，则返回true，否则返回false。

### void update\_sd\_lb\_stats(struct lb\_env \*env, struct sd\_lb\_stats \*sds)

更新负载平衡所需的调度域统计信息。

**参数**

struct lb\_env \*env

负载平衡环境。

struct sd\_lb\_stats \*sds

存储该调度域统计信息的变量。

### int check\_asym\_packing(struct lb\_env \* env, struct sd\_lb\_stats \* sds)

检查组是否已装入调度域。

**参数**

struct lb\_env \*env

负载平衡环境。

struct sd\_lb\_stats \*sds

所需打包的调度域的统计信息

**说明**

主要用于兄弟级别上的调用。某些内核像POWER7 prefer to use lower numbered SMT threads. 对于POWER7，只有在更高的线程空闲时才能向更低的SMT mode移动。当在低SMT模式下时，线程会表现更好，因为它们共享了更少的内核资源。因此，当我们有空闲的线程时，我们希望它们是更高的线程。

此打包函数在空闲线程上运行。它检查这个域（在P7中为内核）中最繁忙的CPU是否具有比当前运行打包函数的CPU更高的CPU编号。这里假定较低的CPU编号等价于较低的SMT线程编号。

**返回**

如果需要打包并将任务移动到该CPU，则返回1。环境（env->imbalance）返回不平衡的量。

### void fix\_small\_imbalance(struct lb\_env \* env, struct sd\_lb\_stats \* sds)

计算负载平衡期间调度域组之间存在的小不平衡。

**参数**

struct lb\_env \*env

负载平衡环境。

struct sd\_lb\_stats \*sds

需要计算其不平衡度的调度域统计信息。

### void calculate\_imbalance(struct lb\_env \*env, struct sd\_lb\_stats \*sds)

计算负载平衡期间给定调度域组内部存在的不平衡程度。

**参数**

struct lb\_env \*env

负载均衡环境

struct sd\_lb\_stats \*sds

需要计算其中不平衡度的调度域统计信息。

### struct sched\_group \*find\_busiest\_group(struct lb\_env \* env)

返回调度域内最繁忙的组，如果存在不平衡现象。

**参数**

struct lb\_env \*env

负载平衡环境。

**说明**

同时计算需要移动以恢复平衡的可运行负载量。

**返回**

如果存在不平衡现象，则返回最繁忙的组。

### DECLARE\_COMPLETION (work)

声明并初始化完成结构

**参数**

工作

完成结构的标识符

**说明**

这个宏声明并初始化完成结构。通常用于静态声明。自动变量应使用\_ONSTACK变体。

### DECLARE\_COMPLETION\_ONSTACK

DECLARE\_COMPLETION\_ONSTACK (work)

在内核堆栈上声明并初始化完成结构

**参数**

工作

完成结构的标识符

**说明**

这个宏在内核堆栈上声明并初始化完成结构。

### void init\_completion(struct completion \*x)

初始化动态分配的完成结构

**参数**

struct completion \*x

指向需初始化的完成结构的指针

**说明**

此内联函数将初始化动态创建的完成结构。

### void reinit\_completion(struct completion \*x)

重新初始化完成结构

**参数**

struct completion \*x

指向需要重新初始化的完成结构的指针

**说明**

应使用此内联函数重新初始化完成结构，以便它可以被重用。在使用complete\_all()后，这一点尤其重要。

### unsigned long \_\_round\_jiffies(unsigned long j, int cpu)

将 jiffies 舍入为整秒的函数

**参数**

unsigned long j

要舍入的绝对时间（以 jiffies 表示）

int cpu

超时将发生的处理器编号

**说明**

\_\_round\_jiffies()将未来（以 jiffies 表示）的绝对时间上舍入为整秒。例如，如果某个定时器的确切触发时间不太重要，只要它每 X 秒左右触发一次即可，那么这种舍入就很有用。

通过将这些定时器舍入为整秒，所有这样的定时器都将同时触发，而不是在各个时间点分散触发。这样做的目的是让 CPU 少唤醒，从而节省电力。

由于每个处理器的舍入都有偏差，以避免所有处理器在确切的相同时刻重复触发与此相关的锁争用或伪共享缓存行。

返回是 j 参数的舍入版本。

### unsigned long \_\_round\_jiffies\_relative(unsigned long j, int cpu)

将 jiffies 舍入为整秒的函数

**参数**

unsigned long j

要舍入的相对时间（以 jiffies 表示）

int cpu

超时将发生的处理器编号

**说明**

\_\_round\_jiffies\_relative()将相对（以 jiffies 表示）的未来时间向上或向下舍入为（大约）整秒。例如，如果某个定时器的确切触发时间不太重要，只要它每 X 秒左右触发一次即可，那么这种舍入就很有用。

通过将这些定时器舍入为整秒，所有这样的定时器都将同时触发，而不是在各个时间点分散触发。这样做的目的是让 CPU 少唤醒，从而节省电力。

由于每个处理器的舍入都有偏差，以避免所有处理器在确切的相同时刻重复触发与此相关的锁争用或伪共享缓存行。

返回是 j 参数的舍入版本。

### unsigned long round\_jiffies（unsigned long j）

将jiffies舍入为整数秒的函数

**参数**

unsigned long j

应舍入的（绝对）jiffies时间

**说明**

round\_jiffies（）将将来的时间（以jiffies为单位）向上或向下舍入到（大约）整秒。 这对于时间器来说非常有用，因为它们触发的确切时间并不太重要，只要它们大约每X秒触发一次即可。

将这些定时器舍入到整数秒，所有这样的定时器将在同一时间触发，而不是在不同的时间间隔触发。 目标是让CPU更少地唤醒，从而节省电力。

返回是j参数的舍入版本。

### unsigned long round\_jiffies\_relative（unsigned long j）

将jiffies舍入为整数秒的函数

**参数**

unsigned long j

应舍入的（相对）jiffies时间

**说明**

round\_jiffies\_relative（）将将来的时间差（以jiffies为单位）向上或向下舍入到（大约）整秒。 这对于时间器来说非常有用，因为它们触发的确切时间并不太重要，只要它们大约每X秒触发一次即可。

将这些定时器舍入到整数秒，所有这样的定时器将在同一时间触发，而不是在不同的时间间隔触发。 目标是让CPU更少地唤醒，从而节省电力。

返回是j参数的舍入版本。

### unsignedlong \_\_round\_jiffies\_up（unsigned long j，int cpu）

将jiffies舍入为整数秒的函数

**参数**

unsigned long j

应舍入的（绝对）jiffies时间

int cpu

超时将发生的处理器编号

**说明**

除了它永远不会向下舍入之外，它与\_\_round\_jiffies（）相同。 这对于超时来说非常有用，因为它们触发的确切时间并不太重要，只要它们不太早触发即可。

### unsigned long \_\_round\_jiffies\_up\_relative（unsigned long j，int cpu）

将jiffies舍入为整数秒的函数

**参数**

unsigned long j

应舍入的（相对）jiffies时间

int cpu

超时将发生的处理器编号

**说明**

除了它永远不会向下舍入之外，它与\_\_round\_jiffies\_relative（）相同。 这对于超时来说非常有用，因为它们触发的确切时间并不太重要，只要它们不太早触发即可。

### unsigned long round\_jiffies\_up（unsigned long j）

将jiffies舍入为整数秒的函数

**参数**

unsigned long j

应舍入的（绝对）jiffies时间

**说明**

除了它永远不会向下舍入之外，它与round\_jiffies（）相同。 这对于超时来说非常有用，因为它们触发的确切时间并不太重要，只要它们不太早触发即可。

### unsigned long round\_jiffies\_up\_relative（unsigned long j）

将jiffies舍入为整数秒的函数

**参数**

unsigned long j

应舍入的（相对）jiffies时间

**说明**

除了它永远不会向下舍入之外，它与round\_jiffies\_relative（）相同。 这对于超时来说非常有用，因为它们触发的确切时间并不太重要，只要它们不太早触发即可。

### void init\_timer\_key

void init\_timer\_key（struct timer\_list \* timer，void（\* func）（struct timer\_list \*），unsigned int flags，const char \* name，struct lock\_class\_key \* key）

初始化计时器

**参数**

struct timer\_list \* timer

要初始化的计时器

void（\* func）（struct timer\_list \*）

计时器回调函数

unsigned int flags

计时器标志

const char \* name

计时器名称

struct lock\_class\_key \* key

用于跟踪计时器同步锁依赖关系的假锁的lockdep类别键

**说明**

在调用任何其他计时器函数之前，必须对计时器执行init\_timer\_key（）。

### int mod\_timer\_pending（struct timer\_list \* timer，unsigned long expires）

修改挂起计时器的超时

**参数**

struct timer\_list \*timer

要修改的挂起计时器

unsigned long expires

以jiffies为单位的新计时器超时时间

**说明**

mod\_timer\_pending()对于挂起的计时器与mod\_timer()相同，但不会重新激活和修改已经删除的计时器。

它对于计时器的非序列化使用很有用。

### int mod\_timer(struct timer\_list \*timer, unsigned long expires)

修改计时器的超时时间

**参数**

struct timer\_list \*timer

要修改的计时器

unsigned long expires

以jiffies为单位的新计时器超时时间

**说明**

mod\_timer()是更新活动计时器的到期字段的更有效的方法（如果计时器未活动则它将被激活）。

mod\_timer(timer，expires)相当于

del\_timer(timer); timer->expires = expires; add\_timer(timer);

请注意，如果有多个非序列化并发用户使用相同的计时器，则mod\_timer()是修改超时的唯一安全方法，因为add\_timer()无法修改已经运行的计时器。

该函数返回它是否已经修改了一个挂起的计时器。（即mod\_timer()的未活动计时器返回0，mod\_timer()的活动计时器返回1。）

### int timer\_reduce(struct timer\_list \*timer, unsigned long expires)

如果减少时间，则修改计时器的超时时间

**参数**

struct timer\_list \*timer

要修改的计时器

unsigned long expires

以jiffies为单位的新计时器超时时间

**说明**

timer\_reduce()与mod\_timer()非常相似，但它仅在减少过期时间时修改正在运行的计时器（它将启动一个未运行的计时器）。

### void add\_timer(struct timer\_list \*timer)

开始一个计时器

**参数**

struct timer\_list \*timer

要添加的计时器

**说明**

内核从将来计时器中断的“到期时间”调用 ->function(timer) 回调。当前时间为“jiffies”。

在调用该函数之前，计时器的->expires、->function字段必须被设置。

超时时间（->expires字段）较早的计时器将在下一个计时器时刻执行。

### void add\_timer\_on(struct timer\_list \*timer, int cpu)

在特定CPU上启动计时器

**参数**

struct timer\_list \*timer

要添加的计时器

int cpu

启动计时器的CPU

**说明**

这在SMP上不是非常可扩展的。不能进行双倍添加。

### int del\_timer(struct timer\_list \*timer)

停用一个计时器。

**参数**

struct timer\_list \*timer

要停用的计时器

**说明**

del\_timer()停用一个计时器——这适用于活动计时器和未活动计时器。

该函数返回它是否已停用一个挂起计时器。（即不活动计时器的del\_timer()返回0，而活动计时器的del\_timer()返回1。）

### int try\_to\_del\_timer\_sync(struct timer\_list \*timer)

尝试停用一个计时器

**参数**

struct timer\_list \*timer

要删除的计时器

**说明**

此函数尝试停用计时器。成功（ret >= 0）退出后，计时器未排队，并且处理程序不在任何CPU上运行。

### int del\_timer\_sync(struct timer\_list \*timer)

停用一个计时器并等待处理程序完成。

**参数**

struct timer\_list \*timer

要停用的计时器

**说明**

此函数与del\_timer()在SMP上唯一不同除了停用计时器之外，它还确保处理程序在其他CPU上执行完毕。

同步规则调用者必须防止重新启动计时器，否则此函数没有意义。除非计时器是irqsafe计时器，否则不能在中断上下文中调用它。调用者不能持有阻止计时器处理程序完成的锁。计时器的处理程序不能调用add\_timer\_on()。退出后，计时器未排队并且处理程序不在任何CPU上运行。

**注意**

对于不是irqsafe计时器，当调用此函数时，您必须不持有在中断上下文中持有的锁。即使该锁与所讨论的计时器无关。

CPU0 CPU1

---- ----

<SOFTIRQ>

call\_timer\_fn();

base->running\_timer = mytimer;

spin\_lock\_irq(somelock);

<IRQ>

spin\_lock(somelock);

del\_timer\_sync(mytimer);

while(base->running\_timer == mytimer);

则del\_timer\_sync()将永远不会返回，也永远不会释放somelock。另一个CPU上的中断正在等待获取somelock，但它已中断了CPU0正在等待完成的softirq。

该函数返回它是否已停用一个挂起计时器。（即不活动计时器的del\_timer\_sync()返回0，而活动计时器的del\_timer\_sync()返回1。）

### signed long schedule\_timeout(signed long timeout)

休眠，直到超时

**参数**

signed long timeout

以jiffies为单位的超时值

**说明**

让当前任务休眠，直到超时的jiffies已经过去。函数的行为取决于当前任务的状态（也可以参见set\_current\_state()说明）

TASK\_UNINTERRUPTIBLE - 保证至少有timeout个jiffies经过，然后该例程才返回，除非当前任务被显式唤醒（例如通过wake\_up\_process()）。

TASK\_INTERRUPTIBLE - 如果向当前任务发送信号或者当前任务被显式唤醒，该例程可能会提前返回。

当此函数返回时，当前任务状态保证为TASK\_RUNNING。

指定MAX\_SCHEDULE\_TIMEOUT的超时值将在无限制的超时下调度CPU。在这种情况下，返回将是MAX\_SCHEDULE\_TIMEOUT。

计时器已过期时返回0，否则返回以jiffies为单位的剩余时间。在所有情况下，返回保证为非负数。

### void msleep(unsigned int msecs)

即使有waitqueue中断也可以安全睡眠

**参数**

unsigned int msecs

睡眠时间（以毫秒为单位）

### unsigned long msleep\_interruptible(unsigned int msecs)

睡眠等待信号

**参数**

unsigned int msecs

睡眠时间（以毫秒为单位）

### void usleep\_range(unsigned long min, unsigned long max)

近似睡眠

**参数**

unsigned long min

最少的睡眠时间（以微秒为单位）

unsigned long max

最多的睡眠时间（以微秒为单位）

**说明**

在非原子环境下，如果确切的唤醒时间是灵活的，则应使用usleep\_range()而不是udelay()。睡眠通过避免CPU占用的忙等待来提高响应能力，而范围通过允许hrtimers利用已经调度的中断而不是仅为此睡眠而调度一个新的中断来降低功耗。

## 等待队列和唤醒事件

### int waitqueue\_active(struct wait\_queue\_head \*wq\_head)

无锁测试队列中是否有等待者

**参数**

struct 等待队列头\* wq\_head

要测试等待者的等待队列

**说明**

如果等待列表不为空，则返回true。

**注意**

此函数是无锁的，并且需要小心使用，错误的使用会导致零散且不明显的故障。

在持有wait\_queue\_head::lock的同时使用或用于带有额外smp\_mb()的唤醒时使用

CPU0 - 唤醒器CPU1 - 等待者

for (;;) {

cond = true; prepare\_to\_wait(wq\_head, wait, state); smp\_mb(); // smp\_mb() from set\_current\_state() if (waitqueue\_active(wq\_head)) if (cond)

wake\_up(wq\_head); break;

schedule();

} finish\_wait(wq\_head, wait);

如果没有明确的smp\_mb()，则可能会将waitqueue\_active()加载提升到cond存储器之上，以便我们在观察到空等待列表时，等待者可能不观察到cond。

还要注意，此“优化”是用一个spin\_lock()来交换一个smp\_mb()，（当锁没有竞争时），代价大约相同。

### bool wq\_has\_sleeper(struct wait\_queue\_head \*wq\_head)

检查是否有正在等待的进程

**参数**

struct 等待队列头\* wq\_head

等待队列头

**说明**

如果wq\_head有正在等待的进程，则返回true。

请参阅waitqueue\_active的注释。

### wait\_event

wait\_event(wq\_head，condition)

等待条件为真时睡眠

**参数**

wq\_head

要等待的等待队列

condition

用于等待事件的C表达式

**说明**

将进程放入睡眠状态（TASK\_UNINTERRUPTIBLE），直到条件计算为true。每次唤醒waitqueue wq\_head时都会检查条件。

更改可能改变等待条件结果的任何变量后都必须调用wake\_up()。

### wait\_event\_freezable

wait\_event\_freezable(wq\_head，condition)

睡眠（或冻结）直到条件成立

**参数**

wq\_head

要等待的等待队列

condition

用于等待事件的C表达式

**说明**

将进程放入睡眠状态（TASK\_INTERRUPTIBLE - 以减少系统负载），直到条件计算为true。每次唤醒waitqueue wq\_head时都会检查条件。

更改可能改变等待条件结果的任何变量后都必须调用wake\_up()。

### wait\_event\_timeout

wait\_event\_timeout(wq\_head，condition，timeout)

睡眠直到条件成立或超时

**参数**

wq\_head

要等待的等待队列

condition

用于等待事件的C表达式

timeout

循环周期（以jiffies为单位）

**说明**

将进程设置为睡眠状态（TASK\_UNINTERRUPTIBLE），直到条件计算为true。每次唤醒waitqueue wq\_head时都会检查条件。

更改可能改变等待条件结果的任何变量后都必须调用wake\_up()。

**返回**

如果在超时时间结束后条件计算为false，则返回0；如果在超时时间结束后条件计算为true，则返回1；如果在超时之前条件计算为true，则返回剩余的jiffies（至少为1）。

### wait\_event\_cmd

wait\_event\_cmd(wq\_head，condition，cmd1，cmd2)

睡眠直到条件为真

**参数**

wq\_head

要等待的等待队列

condition

用于等待事件的C表达式

cmd1

将在睡眠之前执行的命令

cmd2

将在睡眠之后执行的命令

**说明**

将进程放入睡眠状态（TASK\_UNINTERRUPTIBLE），直到条件计算为true。每次唤醒waitqueue wq\_head时都会检查条件。

更改可能改变等待条件结果的任何变量后都必须调用wake\_up()。

### wait\_event\_interruptible

wait\_event\_interruptible（wq\_head，condition）

睡眠直到条件为真或收到信号结束。

**参数**

wq\_head

等待队列头

condition

等待事件的C表达式条件

**说明**

进程被置于睡眠状态（TASK\_INTERRUPTIBLE），直到条件为真或收到信号为止。每次唤醒等待队列（wq\_head）时都会检查条件是否满足。

如果改变了任何可能改变等待条件结果的变量，则必须调用wake\_up()。

如果被信号中断则返回 -ERESTARTSYS，如果条件为真则返回 0。

### wait\_event\_interruptible\_timeout (wq\_head, condition, timeout)

睡眠直到条件为真或超时结束。

**参数**

wq\_head

等待队列头

condition

等待事件的C表达式条件

timeout

超时时间（以jiffies为单位）。

**说明**

进程被置于睡眠状态（TASK\_INTERRUPTIBLE），直到条件为真或收到信号为止，每次唤醒等待队列（wq\_head）时都会检查条件是否满足。

如果改变了任何可能改变等待条件结果的变量，则必须调用wake\_up()。

返回为 0 或 1，分别代表超时结束时条件是否满足。如果条件在超时前满足，则返回余下的jiffies，如果被信号中断则返回 -ERESTARTSYS。

### wait\_event\_hrtimeout (wq\_head, condition, timeout)

睡眠直到条件为真或超时结束。

**参数**

wq\_head

等待队列头

condition

等待事件的C表达式条件

timeout

超时时间(以ktime\_t为单位）。

**说明**

进程被置于睡眠状态（TASK\_UNINTERRUPTIBLE），直到条件为真或收到信号为止，每次唤醒等待队列（wq\_head）时都会检查条件是否满足。

如果改变了任何可能改变等待条件结果的变量，则必须调用wake\_up()。

返回为 0 或 -ETIME，分别代表条件是否满足和超时结束。

### wait\_event\_interruptible\_hrtimeout (wq, condition, timeout)

睡眠直到条件为真或超时结束。

**参数**

wq\_head

等待队列头

condition

等待事件的C表达式条件

timeout

超时时间(以ktime\_t为单位）。

**说明**

进程被置于睡眠状态（TASK\_INTERRUPTIBLE），直到条件为真或收到信号为止，每次唤醒等待队列（wq）时都会检查条件是否满足。

如果改变了任何可能改变等待条件结果的变量，则必须调用wake\_up()。

返回为 0，-ERESTARTSYS或 -ETIME，分别代表条件是否满足、如果被信号中断或超时结束。

### wait\_event\_idle (wq\_head, condition)

等待条件为真而不贡献系统负载。

**参数**

wq\_head

等待队列头

condition

等待事件的C表达式条件

**说明**

进程被置于睡眠状态（TASK\_IDLE），直到条件为真，每次唤醒等待队列（wq\_head）时都会检查条件是否满足。

如果改变了任何可能改变等待条件结果的变量，则必须调用wake\_up()。

### wait\_event\_idle\_exclusive (wq\_head, condition)

等待条件为真同时贡献系统负载。

**参数**

wq\_head

等待队列头

condition

等待事件的C表达式条件

**说明**

进程被置于睡眠状态（TASK\_IDLE），直到条件为真。每次唤醒等待队列（wq\_head）时都会检查条件是否满足。

将进程加入带有 WQ\_FLAG\_EXCLUSIVE 标志的等待队列。因此，当其他进程等待相同的列表时，唤醒该进程时不会考虑其他进程。

如果改变了任何可能改变等待条件结果的变量，则必须调用wake\_up()。

### wait\_event\_idle\_timeout (wq\_head, condition, timeout)

睡眠直到条件为真或超时结束，不贡献系统负载。

**参数**

wq\_head

等待队列头

condition

等待事件的C表达式条件

timeout

超时时间（以jiffies为单位）。

**说明**

进程被置于睡眠状态（TASK\_IDLE），直到条件为真，每次唤醒等待队列（wq\_head）时都会检查条件是否满足。

如果改变了任何可能改变等待条件结果的变量，则必须调用wake\_up()。

返回为 0 或 1，分别代表超时结束时条件是否满足。如果条件在超时前满足，则返回余下的jiffies。

### wait\_event\_idle\_exclusive\_timeout (wq\_head, condition, timeout)

睡眠直到条件为真或超时结束，贡献系统负载。

**参数**

wq\_head

等待队列头

condition

等待事件的C表达式条件

timeout

超时时间（以jiffies为单位）

**说明**

进程被置于睡眠状态（TASK\_IDLE），直到条件为真。每次唤醒等待队列（wq\_head）时都会检查条件是否满足。

将进程加入带有 WQ\_FLAG\_EXCLUSIVE 标志的等待队列。因此，当其他进程等待相同的列表时，唤醒该进程时不会考虑其他进程。

如果改变了任何可能改变等待条件结果的变量，则必须调用wake\_up()。

返回为 0 或 1，分别代表超时结束时条件是否满足。如果条件在超时前满足，则返回余下的jiffies。

### wait\_event\_interruptible\_locked

wait\_event\_interruptible\_locked (wq, condition)

等待直到条件为true

**参数**

wq

等待要等待的队列

条件

待等待事件的C表达式

**说明**

进程被置于睡眠(TASK\_INTERRUPTIBLE)状态，直到条件被评估为true或接收到信号。每次唤醒等待队列wq时都会检查条件。

必须在持有wq.lock的情况下调用此函数。当睡眠时，这个自旋锁会被解锁，但是在持有锁的情况下进行条件测试，并且当这个宏退出时，锁是被持有的。

锁使用spin\_lock()/spin\_unlock()函数锁定/解锁，这与它们在此宏外部锁定/解锁的方式必须匹配。

在更改可能改变等待条件结果的任何变量后必须调用wake\_up\_locked()。

如果被信号中断，则返回-ERESTARTSYS；如果条件评估为true，则返回0。

### wait\_event\_interruptible\_locked\_irq

wait\_event\_interruptible\_locked\_irq (wq, condition)

等待直到条件为true

**参数**

wq

等待要等待的队列

条件

待等待事件的C表达式

**说明**

进程被置于睡眠(TASK\_INTERRUPTIBLE)状态，直到条件被评估为true或接收到信号。每次唤醒等待队列wq时都会检查条件。

必须在持有wq.lock的情况下调用此函数。当睡眠时，这个自旋锁会被解锁，但是在持有锁的情况下进行条件测试，并且当这个宏退出时，锁是被持有的。

锁使用spin\_lock\_irq()/spin\_unlock\_irq()函数锁定/解锁，这与它们在此宏外部锁定/解锁的方式必须匹配。

在更改可能改变等待条件结果的任何变量后必须调用wake\_up\_locked()。

如果被信号中断，则返回-ERESTARTSYS；如果条件评估为true，则返回0。

### wait\_event\_interruptible\_exclusive\_locked

wait\_event\_interruptible\_exclusive\_locked (wq, condition)

独占睡眠直到条件为true

**参数**

wq

等待要等待的队列

条件

待等待事件的C表达式

**说明**

进程被置于睡眠(TASK\_INTERRUPTIBLE)状态，直到条件被评估为true或接收到信号。每次唤醒等待队列wq时都会检查条件。

必须在持有wq.lock的情况下调用此函数。当睡眠时，这个自旋锁会被解锁，但是在持有锁的情况下进行条件测试，并且当这个宏退出时，锁是被持有的。

锁使用spin\_lock()/spin\_unlock()函数锁定/解锁，这与它们在此宏外部锁定/解锁的方式必须匹配。

进程被放置在等待队列中，并设置WQ\_FLAG\_EXCLUSIVE标志，因此当其他进程在列表上等待时，如果唤醒了该进程，则不再考虑其他进程。

在更改可能改变等待条件结果的任何变量后必须调用wake\_up\_locked()。

如果被信号中断，则返回-ERESTARTSYS；如果条件评估为true，则返回0。

### wait\_event\_interruptible\_exclusive\_locked\_irq

wait\_event\_interruptible\_exclusive\_locked\_irq (wq, condition)

独占睡眠直到条件为true

**参数**

wq

等待要等待的队列

条件

待等待事件的C表达式

**说明**

进程被置于睡眠(TASK\_INTERRUPTIBLE)状态，直到条件被评估为true或接收到信号。每次唤醒等待队列wq时都会检查条件。

必须在持有wq.lock的情况下调用此函数。当睡眠时，这个自旋锁会被解锁，但是在持有锁的情况下进行条件测试，并且当这个宏退出时，锁是被持有的。

锁使用spin\_lock\_irq()/spin\_unlock\_irq()函数锁定/解锁，这与它们在此宏外部锁定/解锁的方式必须匹配。

进程被放置在等待队列中，并设置WQ\_FLAG\_EXCLUSIVE标志，因此当其他进程在列表上等待时，如果唤醒了该进程，则不再考虑其他进程。

在更改可能改变等待条件结果的任何变量后必须调用wake\_up\_locked()。

如果被信号中断，则返回-ERESTARTSYS；如果条件评估为true，则返回0。

### wait\_event\_killable

wait\_event\_killable（wq\_head，condition）

睡眠直到条件为真

**参数**

wq\_head

等待队列上等待

condition

等待事件的C表达式

**说明**

进程睡眠（TASK\_KILLABLE），直到条件求值为真或接收到信号。每次唤醒waitqueue wq\_head时都会检查条件。

在更改任何可能改变等待条件结果的变量后，必须调用wake\_up（）。

如果它被信号中断，则该函数将返回-ERESTARTSYS，如果条件求值为true，则返回0。

### wait\_event\_killable\_timeout

wait\_event\_killable\_timeout（wq\_head，condition，timeout）

睡眠直到条件为真或超时

**参数**

wq\_head

等待队列上等待

condition

等待事件的C表达式

timeout

按照滴答声超时

**说明**

进程睡眠（TASK\_KILLABLE），直到条件求值为真或收到kill信号。每次唤醒waitqueue wq\_head时都会检查条件。

在更改任何可能改变等待条件结果的变量后，必须调用wake\_up（）。

只有kill信号会中断此进程。

**返回**

如果在超时之后条件求值为false，则返回0，如果在超时之后条件求值为true，则返回1，如果在超时之前条件求值为true，则返回剩余的jiffies（至少为1），或者在被kill信号中断时返回-ERESTARTSYS。

只有kill信号会中断此进程。

### wait\_event\_lock\_irq\_cmd

wait\_event\_lock\_irq\_cmd（wq\_head，condition，lock，cmd）

睡眠直到条件为真。在锁下检查条件。预计使用已锁定。

**参数**

wq\_head

等待队列上等待

condition

等待事件的C表达式

lock

一个已锁定的spinlock\_t，在cmd和schedule（）之前和之后将被释放和重新获取。

cmd

在休眠之前在临界区外调用的命令

**说明**

进程被置于睡眠状态（TASK\_UNINTERRUPTIBLE）直到条件求值为真。每次唤醒waitqueue wq\_head时都会检查条件。

在更改任何可能改变等待条件结果的变量后，必须调用wake\_up（）。

预计在持有锁时调用此函数。在调用cmd和睡眠之前，锁被释放，并在之后重新获得。

### wait\_event\_lock\_irq

wait\_event\_lock\_irq（wq\_head，condition，lock）

睡眠直到条件为真。在锁下检查条件。预计使用已锁定。

**参数**

wq\_head

等待队列上等待

condition

等待事件的C表达式

lock

一个已锁定的spinlock\_t，在schedule（）之前和之后将被释放和重新获取。

**说明**

进程被置于睡眠状态（TASK\_UNINTERRUPTIBLE）直到条件求值为真。每次唤醒waitqueue wq\_head时都会检查条件。

在更改任何可能改变等待条件结果的变量后，必须调用wake\_up（）。

预计在持有锁时调用此函数。在睡眠之前释放锁并在之后重新获得。

### wait\_event\_interruptible\_lock\_irq\_cmd

wait\_event\_interruptible\_lock\_irq\_cmd（wq\_head，condition，lock，cmd）

睡眠直到条件为真。在锁下检查条件。预计使用已锁定。

**参数**

wq\_head

等待队列上等待

condition

等待事件的C表达式

lock

一个已锁定的spinlock\_t，在cmd和schedule（）之前和之后将被释放和重新获取。

cmd

在休眠之前在临界区外调用的命令

**说明**

进程被放入睡眠状态（TASK\_INTERRUPTIBLE），直到条件求值为真或收到信号。每次唤醒waitqueue wq\_head时都会检查条件。

在更改任何可能改变等待条件结果的变量后，必须调用wake\_up（）。

预计在持有锁时调用此函数。在调用cmd和睡眠之前，锁被释放，并在之后重新获得。

如果它被信号中断，则该宏将返回-ERESTARTSYS，如果条件求值为true，则返回0。

### wait\_event\_interruptible\_lock\_irq

wait\_event\_interruptible\_lock\_irq（wq\_head，condition，lock）

睡眠直到条件为真。在锁下检查条件。预计使用已锁定。

**参数**

wq\_head

等待队列上等待

condition

等待事件的C表达式

lock

一个已锁定的spinlock\_t，在schedule（）之前和之后将被释放和重新获取。

**说明**

进程被放入睡眠状态（TASK\_INTERRUPTIBLE），直到条件求值为true或收到信号。每次唤醒waitqueue wq\_head时都会检查条件。

在更改任何可能改变等待条件结果的变量后，必须调用wake\_up（）。

这个应该在保持锁的同时调用。在睡眠之前放下锁，之后重新获取。

如果被信号中断，宏将返回-ERESTARTSYS，如果条件评估为真，则返回0。

### wait\_event\_interruptible\_lock\_irq\_timeout

wait\_event\_interruptible\_lock\_irq\_timeout（wq\_head，condition，lock，timeout）

睡眠，直到条件为真或超时。该条件在锁下检查。这预计在获取锁时调用。

**参数**

wq\_head

要等待的等待队列

condition

要等待的事件的C表达式

lock

已锁定的spinlock\_t，将在schedule（）之前释放并重新获取。

超时

超时，在jiffies中

**说明**

将进程置于睡眠状态（TASK\_INTERRUPTIBLE），直到条件评估为真或接收到信号。每次唤醒waitqueue wq\_head时都会检查条件。

在更改可能改变等待条件结果的任何变量后，必须调用wake\_up（）。

这个应该在保持锁的同时调用。在睡眠之前放下锁，之后重新获取。

如果超时已过，则函数返回0，如果由信号中断，则返回-ERESTARTSYS，否则返回调用时剩余的jiffies（如果在超时之前评估为true）。

### void \_\_wake\_up（struct wait\_queue\_head \* wq\_head，unsigned int mode，int nr\_exclusive，void \* key）

叫醒在等待队列上阻塞的线程。

**参数**

待等待的等待队列 \* wq\_head

mode

哪些线程

nr\_exclusive

唤醒一个或唤醒多个线程的数量

键

直接传递给唤醒函数

**说明**

如果此函数唤醒任务，它将在访问任务状态之前执行完整的内存屏障。

### void \_\_wake\_up\_sync\_key（struct wait\_queue\_head \* wq\_head，unsigned int mode，void \* key）

叫醒在等待队列上阻塞的线程。

**参数**

待等待的等待队列 \* wq\_head

mode

哪些线程

键

要传递给唤醒目标的不透明值

**说明**

同步唤醒不同之处在于唤醒程序知道它很快就会执行调度，因此尽管唤醒目标线程，但不会将其迁移到另一个CPU-即两个线程彼此“同步”。这可以避免CPU之间不必要的反弹。

在UP上它可以防止额外的抢占。

如果此函数唤醒任务，它将在访问任务状态之前执行完整的内存屏障。

### void finish\_wait（struct wait\_queue\_head \* wq\_head，struct wait\_queue\_entry \* wq\_entry）

在队列中等待后进行清理

**参数**

等待等待的等待队列 \* wq\_head

等待说明符

**说明**

如果仍在队列中则将当前线程设置回运行状态并删除给定等待队列中的等待说明符。

## 高分辨率计时器

### ktime\_t ktime\_set（const s64 secs，const unsigned long nsecs）

从秒/纳秒值设置ktime\_t变量

**参数**

**const s64 sec**

要设置的秒

**const unsigned long nsecs**

要设置的纳秒

**返回**

值的ktime\_t表示。

### int ktime\_compare（const ktime\_t cmp1，const ktime\_t cmp2）

比较两个ktime\_t变量大小

**const ktime\_t cmp1**

可以比较1

**const ktime\_t cmp2**

可比较2

**返回**

…

cmp1 <cmp2返回 <0 ；cmp1 == cmp2返回 0 ；cmp1> cmp2返回> 0

### bool ktime\_after（const ktime\_t cmp1，const ktime\_t cmp2）

比较ktime\_t值是否大于另一个值。

**参数**

**const ktime\_t cmp1**

可以比较1

**const ktime\_t cmp2**

可比较2

**返回**

如果cmp1发生在cmp2之后，则为true。

### bool ktime\_before（const ktime\_t cmp1，const ktime\_t cmp2）

比较ktime\_t值是否小于另一个值。

**参数**

**const ktime\_t cmp1**

可以比较1

**const ktime\_t cmp2**

可比较2

**返回**

如果cmp1比cmp2先发生，则为true。

### bool ktime\_to\_timespec\_cond（const ktime\_t kt，struct timespec \* ts）

仅在变量包含数据的情况下将ktime\_t变量转换为timespec格式

**参数**

const ktime\_t kt

要转换的ktime\_t变量

struct timespec \* ts

将结果存储在的timespec变量

**返回**

如果成功转换，则为true，如果kt为0，则为false。

### bool ktime\_to\_timespec64\_cond（const ktime\_t kt，struct timespec64 \* ts）

仅在变量包含数据的情况下将ktime\_t变量转换为timespec64格式

**参数**

const ktime\_t kt

要转换的ktime\_t变量

struct timespec \* ts

将结果存储在的timespec变量

**返回**

如果成功转换，则为真，如果kt为0，则为假。

struct hrtimer

基本的hrtimer结构

**定义**

struct hrtimer {

struct timerqueue\_node node;

ktime\_t \_softexpires;

enum hrtimer\_restart (\*function)(struct hrtimer \*);

struct hrtimer\_clock\_base \*base;

u8 state;

u8 is\_rel;

u8 is\_soft;

};

**成员**

node

时间队列节点，还管理node.expires，即hrtimers内部表示中的绝对到期时间。该时间与计时器所基于的时钟有关。通过将松弛度添加到\_softexpires值来设置。对于非范围计时器，与\_softexpires相同。

\_softexpires

hrtimer的绝对最早过期时间。定时器被装备时给定的过期时间。

function

计时器过期回调函数

base

指向定时器基准（每个CPU和每个时钟）的指针

state

状态信息（见上面的位值）

is\_rel

如果定时器是相对装备的，则设置

is\_soft

如果hrtimer将在软中断上下文中过期，则设置。

**说明**

hrtimer结构必须通过hrtimer\_init()进行初始化

### struct hrtimer\_sleeper

简单的休眠结构

**定义**

struct hrtimer\_sleeper {

struct hrtimer timer;

struct task\_struct \*task;

};

**成员**

timer

嵌入式计时器结构

task

要唤醒的任务

**说明**

当计时器过期时，将任务设置为NULL。

### struct hrtimer\_clock\_base

特定时钟的计时器基准

**定义**

struct hrtimer\_clock\_base {

struct hrtimer\_cpu\_base \*cpu\_base;

unsigned int index;

clockid\_t clockid;

seqcount\_raw\_spinlock\_t seq;

struct hrtimer \*running;

struct timerqueue\_head active;

ktime\_t(\*get\_time)(void);

ktime\_t offset;

};

**成员**

cpu\_base

每个CPU时钟基准

index

用于per\_cpu支持时将计时器移动到另一个CPU上的时钟类型索引。

clockid

用于per\_cpu支持的时钟ID

seq

\_\_run\_hrtimer周围的seqcount

running

指向当前正在运行的hrtimer的指针

active

用于活动计时器的红黑树根节点

get\_time

检索时钟当前时间的函数

offset

此时钟相对于单调基准的偏移量

### struct hrtimer\_cpu\_base

每个CPU时钟基准

**定义**

struct hrtimer\_cpu\_base {

raw\_spinlock\_t lock;

unsigned int cpu;

unsigned int active\_bases;

unsigned int clock\_was\_set\_seq;

unsigned int hres\_active : 1,in\_hrtirq : 1,hang\_detected : 1,softirq\_activated : 1;

#ifdef CONFIG\_HIGH\_RES\_TIMERS;

unsigned int nr\_events;

unsigned short nr\_retries;

unsigned short nr\_hangs;

unsigned int max\_hang\_time;

#endif;

ktime\_t expires\_next;

struct hrtimer \*next\_timer;

ktime\_t softirq\_expires\_next;

struct hrtimer \*softirq\_next\_timer;

struct hrtimer\_clock\_base clock\_base[HRTIMER\_MAX\_CLOCK\_BASES];

};

**成员**

lock

保护基准和关联时钟基准和计时器的锁

CPU

CPU编号

active\_bases

用于标记具有活动计时器的基准的位域

clock\_was\_set\_seq

时钟被设置事件的序列计数器

hres\_active

高分辨率模式的状态

in\_hrtirq

hrtimer\_interrupt()当前正在执行

hang\_detected

最后一个hrtimer中断检测到挂起

softirq\_activated

如果引发了softirq，则显示-然后不需要更新与softirq相关的设置。

nr\_events

hrtimer中断事件的总数

nr\_retries

hrtimer中断重试的总数

nr\_hangs

hrtimer中断挂起的总数

max\_hang\_time

在hrtimer\_interrupt中花费的最长时间

expires\_next

下一个事件的绝对时间，对于远程hrtimer入队是必需的；它是总的第一个到期时间（硬件和软件hrtimer都会考虑在内）

next\_timer

指向第一个到期计时器的指针

softirq\_expires\_next

检查软队列是否也需要过期的时间

softirq\_next\_timer

指向第一个即将到期的基于softirq的计时器的指针

clock\_base

此CPU的时钟基准数组

**注**

next\_timer仅是\_\_remove\_hrtimer()的优化。

不要解引用指针，因为它在跨CPU删除时不是可靠的。

### hrtimer\_start

void hrtimer\_start(struct hrtimer \*timer, ktime\_t tim, const enum hrtimer\_mode mode)

重新启动hrtimer

**参数**

struct hrtimer \*timer

要添加的计时器

ktime\_t tim

到期时间

const enum hrtimer\_mode mode

计时器模式绝对（HRTIMER\_MODE\_ABS）或相对（HRTIMER\_MODE\_REL），以及固定（HRTIMER\_MODE\_PINNED）；软中断基础模式仅用于调试目的！

### void hrtimer\_start( struct [hrtimer](https://www.kernel.org/doc/html/v4.19/driver-api/basics.html" \l "c.hrtimer" \o "计时器) \*  timer , ktime\_t  tim , const enum hrtimer\_mode  mode )

（重新）启动一个 hrtimer

**参数**

struct hrtimer \* timer

要添加的定时器

ktime\_t tim

到期时间

const enum hrtimer\_mode mode

定时器模式：绝对（HRTIMER\_MODE\_ABS）或相对（HRTIMER\_MODE\_REL），以及固定（HRTIMER\_MODE\_PINNED）；基于软中断的模式仅用于调试目的！

### u64 hrtimer\_forward\_now(struct hrtimer \*timer, ktime\_t interval)

延迟计时器过期时间，使其在n之后过期

**参数**

struct hrtimer \*timer

要延迟计时器过期时间

ktime\_t interval

要延迟的时间

**说明**

推迟计时器到期时间，使其在hrtimer时钟基准的当前时间之后过期。返回超限次数。

可以安全地从定时器的回调函数中调用。如果从其他上下文中调用定时器，则定时器既不能在队列中，也不能在运行回调函数，调用者需要注意序列化。

**注意**

这仅更新计时器到期值，而不重新排队计时器。

### u64 hrtimer\_forward(struct hrtimer \*timer, ktime\_t now, ktime\_t interval)

延迟计时器过期时间

**参数**

struct hrtimer \*timer

要延迟计时器过期时间

ktime\_t now

向前调整此时间

ktime\_t interval

要延迟的时间

**说明**

推迟计时器到期时间，使其在将来过期。返回超限次数。

可以安全地从定时器的回调函数中调用。如果从其他上下文中调用定时器，则定时器既不能在队列中，也不能在运行回调函数，调用者需要注意序列化。

**注意**

这仅更新计时器到期值，而不重新排队计时器。

### void hrtimer\_start\_range\_ns(struct hrtimer \*timer, ktime\_t tim, u64 delta\_ns, const enum hrtimer\_mode mode)

重新启动hrtimer

**参数**

struct hrtimer \*timer

要添加的计时器

ktime\_t tim

到期时间

u64 delta\_ns

计时器的“松动”范围

const enum hrtimer\_mode mode

计时器模式绝对（HRTIMER\_MODE\_ABS）或相对（HRTIMER\_MODE\_REL），以及固定（HRTIMER\_MODE\_PINNED）；软中断基础模式仅用于调试目的！

### int hrtimer\_try\_to\_cancel（struct hrtimer \*timer）

尝试停止计时器

**参数**

struct hrtimer \* timer

要停止的高精度计时器

**返回**

当计时器未激活时返回0

当计时器已激活时返回1

当计时器正在执行回调函数且无法停止时返回-1

### int hrtimer\_cancel（struct hrtimer \*timer）

取消计时器并等待处理程序完成。

**参数**

struct hrtimer \* timer

要取消的计时器

**返回**

当未激活计时器时返回0，当计时器已激活时返回1

### ktime\_t \_\_hrtimer\_get\_remaining（const struct hrtimer \* timer, bool adjust）

获取计时器的剩余时间

**参数**

const struct hrtimer \* timer

要读取的计时器

bool adjust

当CONFIG\_TIME\_LOW\_RES = y时，调整相对计时器

### void hrtimer\_init（struct hrtimer \* timer，clockid\_t clock\_id，enum hrtimer\_mode mode）

将计时器初始化到给定时钟

**参数**

struct hrtimer \* timer

要初始化的计时器

clockid\_t clock\_id

要使用的时钟

enum hrtimer\_mode mode

与初始化相关的模式HRTIMER\_MODE\_ABS，HRTIMER\_MODE\_REL，HRTIMER\_MODE\_ABS\_SOFT，HRTIMER\_MODE\_REL\_SOFT

**说明**

以上的PINNED变体可以被传递，但是当hrtimer启动时会忽略固定位的设置。

### int schedule\_hrtimeout\_range（ktime\_t \* expires，u64 delta，const enum hrtimer\_mode mode）

等待超时

**参数**

ktime\_t \* expires

超时值（ktime\_t）

u64 delta

过期超时的松弛度（ktime\_t）

enum型hrtimer\_mode mode

计时器模式

**说明**

使当前任务睡眠，直到给定的到期时间已经过去。该程序将立即返回，除非已设置当前任务状态（请参见set\_current\_state（））。

delta参数使内核可以在既节能又高效的时间内调度实际的唤醒操作。内核为“expires + delta”提供正常的尽力而为的行为，但可以决定更早地触发计时器，但不会超过到期时间。

您可以通过以下方式设置任务状态 -

TASK\_UNINTERRUPTIBLE-除非当前任务被显示唤醒（例如通过wake\_up\_process（）），否则至少保证超时时间已过去。

TASK\_INTERRUPTIBLE-如果信号已传递到当前任务或当前任务已被明确唤醒，则可能早些时候返回。

当此例程返回时，当前任务状态保证为TASK\_RUNNING。

当计时器已过期时返回0。如果任务在计时器过期之前被信号唤醒（仅在TASK\_INTERRUPTIBLE状态下可能），或者被显式唤醒，则返回-EINTR。

### int schedule\_hrtimeout（ktime\_t \* expires，const enum hrtimer\_mode mode）

等待超时

**参数**

ktime\_t \* expires

超时值（ktime\_t）

const enum hrtimer\_mode mode

计时器模式

**说明**

使当前任务睡眠，直到给定的到期时间已经过去。该程序将立即返回，除非已设置当前任务状态（请参见set\_current\_state（））。

您可以通过以下方式设置任务状态 -

TASK\_UNINTERRUPTIBLE-除非当前任务被唤醒（例如通过wake\_up\_process（）），否则至少保证超时时间已过去。

TASK\_INTERRUPTIBLE-如果信号已传递到当前任务或当前任务已被明确唤醒，则可能早些时候返回。

当此例程返回时，当前任务状态保证为TASK\_RUNNING。

当计时器已过期时返回0。如果任务在计时器过期之前被信号（仅在STATE\_INTERRUPTIBLE状态下可能）或显式唤醒唤醒，则返回-EINTR。

## 工作队列和Kevents

### struct workqueue\_attrs

工作队列属性的结构 。

**定义**

struct workqueue\_attrs {

int nice;

cpumask\_var\_t cpumask;

bool no\_numa;

};

**成员**

nice

优美水平

cpumask

允许使用的CPU

no\_numa

禁用NUMA亲和性

与其他字段不同，no\_numa不是worker\_pool的属性。它仅修改apply\_workqueue\_attrs（）如何选择池，因此不参与池哈希计算或相等性比较。

**说明**

这可以用于更改未绑定工作队列的属性。

### work\_pending（work）

找出工作项当前是否待处理

**参数**

work

所考虑的工作项

### delayed\_work\_pending（w）

查找可延迟的工作项当前是否待处理

**参数**

w

所考虑的工作项

### alloc\_workqueue（fmt，flags，max\_active，args ...）

分配工作队列

**参数**

fmt

工作队列名称的printf格式

flags

WQ\_\*标志

max\_active

最大的正在飞行的工作项，0为默认

args ...

用于格式的一些参数

**说明**

使用指定参数分配工作队列。有关WQ\_\*标志的详细信息，请参见Documentation / core-api / workqueue.rst。

\_\_lock\_name宏的舞蹈是为了保证单个lock\_class\_key不会以不同的名称结束，这在lockdep中是不允许的。

**返回**

成功时返回分配的工作队列指针，失败时返回NULL。

### alloc\_ordered\_workqueue（fmt，flags，args ...）

分配有序工作队列

**参数**

fmt

工作队列名称的printf格式

flags

WQ\_\*标志（仅WQ\_FREEZABLE和WQ\_MEM\_RECLAIM有意义）

args...

args for fmt

**说明**

分配有序的工作队列。有序工作队列按照排队顺序最多同时执行一个工作项。它们作为无限制工作队列实现，最大活动度为1。

**返回**

成功时指向分配的工作队列的指针，失败时为NULL。

### bool queue\_work(struct workqueue\_struct \* wq, struct work\_struct \* work)

在工作队列上排队工作

**参数**

struct workqueue\_struct \* wq

要使用的工作队列

struct work\_struct \* work

要排队的工作

**说明**

如果工作已经在队列中，返回false，否则返回true。

我们将工作排队到提交它的CPU上，但如果该CPU出现故障，它可以由另一个CPU处理。

### bool queue\_delayed\_work(struct workqueue\_struct \* wq, struct delayed\_work \* dwork, unsigned long delay)

在延迟后将工作排队到工作队列上

**参数**

struct workqueue\_struct \* wq

要使用的工作队列

struct delayed\_work \* dwork

可延迟工作要排队

unsigned long delay

在排队之前等待的节拍数

**说明**

相当于queue\_delayed\_work\_on()，但尝试使用本地CPU。

### bool mod\_delayed\_work( struct workqueue\_struct \*  wq , struct delayed\_work \*  dwork , unsigned long  delay )

修改延迟或排队延迟的工作

**参数**

struct workqueue\_struct \* wq

要使用的工作队列

struct delayed\_work \* dwork

工作排队

unsigned long delay

排队前等待的jiffies数

**说明**

[mod\_delayed\_work\_on()](https://www.kernel.org/doc/html/v4.19/driver-api/basics.html" \l "c.mod_delayed_work_on" \o "mod_delayed_work_on)在本地 CPU 上。

### bool schedule\_work\_on(int cpu, struct work\_struct \* work)

将工作任务放在特定的CPU上

**参数**

int cpu

要放置工作任务的CPU

struct work\_struct \* work

要完成的工作

**说明**

这将一个作业放在特定的CPU上。

### bool schedule\_work(struct work\_struct \* work)

将工作任务放入全局工作队列

**参数**

struct work\_struct \* work

要完成的工作

**说明**

如果work已经在内核全局工作队列中排队，则返回false，否则返回true。

如果作业已经排队，它将保留在内核全局工作队列的同一位置中；如果未排队，则将其放在内核全局工作队列中。

### void flush\_scheduled\_work(void)

确保任何计划的工作都完成。

**参数**

void

无参数

**说明**

强制执行内核全局工作队列，并阻塞直到其完成。

在调用此函数之前请三思！如果您不很小心，很容易陷入困境。

其中之一是当前在工作队列上的工作项需要获取由您的代码或其调用者持有的锁。

您的代码正在工作例程的上下文中运行。

当发生这些情况时，它们将由lockdep检测到，但第一个可能不会经常发生。这取决于工作队列上有什么工作项以及它们需要什么锁，这是您无法控制的。

在大多数情况下，刷新整个工作队列是过度的；您仅需要知道特定的工作项是否已排队并且未运行。在这种情况下，应使用cancel\_delayed\_work\_sync()或cancel\_work\_sync()。

### bool schedule\_delayed\_work\_on(int cpu, struct delayed\_work \* dwork, unsigned long delay)

延迟后将工作排队到全局工作队列上的CPU

**参数**

int cpu

要使用的CPU

struct delayed\_work \* dwork

要完成的工作

unsigned long delay

等待的节拍数

**说明**

等待一段时间后，在指定的CPU上将作业放入内核全局工作队列。

### bool schedule\_delayed\_work(struct delayed\_work \* dwork, unsigned long delay)

在延迟后将工作任务放入全局工作队列

**参数**

struct delayed\_work \* dwork

要完成的工作

unsigned long delay

要等待的节拍数，为0表示立即执行

**说明**

等待一段时间后，在内核全局工作队列中放置作业。

### bool queue\_work\_on(int cpu, struct workqueue\_struct \* wq, struct work\_struct \* work)

在特定CPU上排队工作

**参数**

int cpu

要执行工作的CPU号

struct workqueue\_struct \* wq

要使用的工作队列

struct work\_struct \* work

要排队的工作

**说明**

我们将工作排队到特定的CPU上，调用者必须确保它不会消失。

**返回**

如果work已经在队列上，则返回false，否则返回true。

### bool queue\_delayed\_work\_on(int CPU，struct workqueue\_struct \* wq，struct delayed\_work \* dwork，unsigned long delay)

在延迟后将工作排队到特定CPU上

**参数**

int cpu

要执行工作的CPU号

struct workqueue\_struct \* wq

要使用的工作队列

struct delayed\_work \* dwork

要排队的作业

unsigned long delay

在排队之前等待的节拍数

**返回**

如果work已经在队列上，则返回false，否则返回true。如果延迟为零且dwork处于空闲状态，则将立即调度它。

### bool mod\_delayed\_work\_on(int cpu, struct workqueue\_struct \* wq, struct delayed\_work \* dwork, unsigned long delay)

修改或在特定CPU上排队延迟工作

**参数**

int cpu

要执行工作的CPU号

struct workqueue\_struct \* wq

要使用的工作队列

struct delayed\_work \* dwork

要排队的工作

unsigned long delay

在排队之前等待的节拍数

**说明**

如果dwork处于空闲状态，则相当于queue\_delayed\_work\_on()；否则，修改dwork的计时器，以便在延迟之后到期。如果延迟为零，则无论其当前状态如何，都保证立即调度work。

如果dwork空闲并排队，返回false；如果dwork挂起且其计时器被修改，返回true。此函数可在任何上下文中调用，包括IRQ处理程序。有关详细信息，请参阅try\_to\_grab\_pending()。

### bool queue\_rcu\_work(struct workqueue\_struct \* wq，struct rcu\_work \* rwork)

在RCU优雅周期后排队工作。

**参数**

struct workqueue\_struct\* wq

要使用的工作队列

struct rcu\_work \* rwork

要排队的工作

如果rwork已挂起，则返回false；否则返回true。请注意，仅在返回true后才能保证完整的RCU优雅周期。当返回false时，rwork保证在执行之后被执行，但执行可能在完整的RCU优雅周期之前发生。

### void flush\_workqueue(struct workqueue\_struct\* wq)

确保任何已预定的工作已运行到完成。

**参数**

struct workqueue\_struct \* wq

要刷新的工作队列

**说明**

此函数睡眠，直到在入口排队的所有工作项已完成执行，但不会被新的传入项锁定。

### void drain\_workqueue(struct workqueue\_struct \* wq)

排空工作队列

**参数**

struct workqueue\_struct \* wq

要排空的工作队列

**说明**

等待工作队列变为空。在排空过程中，仅允许链队列。即，只有当前挂起或正在运行的工作项wq才能在其中排队。重复刷新wq直到其为空。清洁的次数由链接深度决定，应相对较短。如果太久，请发出警报。

### bool flush**\_**work(struct work\_struct \* work)

等待工作完成执行最后一个队列实例

**参数**

struct work\_struct \* work

要刷新的工作

**说明**

等待work完成执行。如果自flush开始以来未重新排队，则在返回时保证work处于空闲状态。

如果已空闲，则返回false；如果等待了工作的完成执行，则返回true。

### bool cancel\_work\_sync(struct work\_struct \* work)

取消工作并等待其完成

**参数**

struct work\_struct \* work

要取消的工作

**说明**

取消work并等待其执行完成。即使工作自行重新排队或迁移到其他工作队列，也可以使用此函数。从此函数返回时，work保证在任何CPU上都不是挂起或正在执行。

不得使用cancel\_work\_sync（delayed\_work->work）用于延迟的工作。请改用cancel\_delayed\_work\_sync()。

调用者必须确保在此函数返回之前，上次排队work的工作队列不能被销毁。

如果work已挂起，则返回true；否则返回false。

### bool flush**\_**delayed\_work(struct delayed\_work \* dwork)

等待dwork完成执行最后一个队列

**参数**

struct delayed\_work \* dwork

要刷新的延迟工作

**说明**

取消延迟计时器，并将挂起的工作项排队以进行立即执行。与flush\_work（）类似，此函数仅考虑dwork的最后一个排队实例。

如果已空闲，则返回false；如果等待了工作的完成执行，则返回true。

### bool flush\_rcu\_work(struct rcu\_work \* rwork)

等待rwork完成执行最后一个队列

**参数**

struct rcu\_work \* rwork

要刷新的rcu工作

如果已空闲，则返回false；如果等待了工作的完成执行，则返回true。

### bool cancel\_delayed\_work(struct delayed\_work \* dwork)

取消延迟工作

**参数**

struct delayed\_work \* dwork

要取消的延迟工作

**说明**

取消挂起的延迟工作。

如果dwork挂起并被取消，则返回true；如果没有挂起，则返回false。

注在返回时，工作回调函数可能仍在运行，除非它返回true并且工作不重新启用。显式刷新或使用cancel\_delayed\_work\_sync（）等待它。

此函数可在任何上下文中调用，包括IRQ处理程序。

### bool cancel\_delayed\_work\_sync(struct delayed\_work \* dwork)

取消延迟工作并等待其完成

**参数**

struct delayed\_work \* dwork

要取消的延迟工作

**说明**

这是延迟工作的cancel\_work\_sync（）。

如果dwork已挂起，则返回true；否则返回false。

### int execute\_in\_process\_context(work\_func\_t fn, struct execute\_work \* ew)

可靠地使用用户上下文执行例程

**参数**

work\_func\_t fn

要执行的函数

struct execute\_work \* ew

用于执行工作结构的保证存储（工作执行时必须可用）

**说明**

如果有进程上下文可用，则立即执行函数；否则，将函数计划为延迟执行。

**返回**

0-函数已执行

1-函数已安排执行

### int apply\_workqueue\_attrs(struct workqueue\_struct \* wq, const struct workqueue\_attrs \* attrs)

将新的workqueue\_attrs应用于未绑定的工作队列

**参数**

struct workqueue\_struct \* wq

目标工作队列

const struct workqueue\_attrs \* attrs

要应用的workqueue\_attrs，使用alloc\_workqueue\_attrs（）分配

**说明**

将attrs应用于未绑定的工作队列wq。除非禁用，在NUMA机器上，此功能将为具有属于attrs->cpumask的可能性CPU的每个NUMA节点映射单独的pwq，以便工作项与其发出的NUMA节点相匹配。更早的pwq将随着正在进行的工作项完成而释放。请注意，重复将自己排队回到back-to-back的工作项将留在其当前的pwq上。

执行GFP\_KERNEL分配。

成功返回0，失败返回-errno。

### void destroy\_workqueue(struct workqueue\_struct \* wq)

安全地终止一个工作队列。

**参数**

struct workqueue\_struct \* wq

目标工作队列

**说明**

安全地销毁一个工作队列。所有当前挂起的工作将被完成。

### void workqueue\_set\_max\_active(struct workqueue\_struct \* wq, int max\_active)

调整工作队列的max\_active

**参数**

struct workqueue\_struct \* wq

目标工作队列

int max\_active

新的max\_active值。

**说明**

将wq的max\_active设置为max\_active。

**上下文**

不要从IRQ上下文调用。

### struct work\_struct \* current\_work(void)

检索当前任务的工作struct

**参数**

无参数

**说明**

确定当前任务是否是工作队列的工作线程以及它正在执行的任务。有助于找出当前任务正在运行的上下文。

**返回**

如果当前任务是工作队列的工作线程，则返回工作struct ，否则返回NULL。

### bool workqueue\_congested(int cpu, struct workqueue\_struct \* wq)

测试工作队列是否拥堵

**参数**

int cpu

所需CPU

struct workqueue\_struct \* wq

目标工作队列

**说明**

测试wq的CPU工作队列是否拥堵。该函数周围没有同步，测试结果不可靠，仅用于提供提示或调试。

如果cpu是WORK\_CPU\_UNBOUND，则在本地CPU上执行测试。请注意，每个CPU和无绑定的工作队列都可能与多个具有单独拥堵状态的pool\_workqueues相关联。一个工作队列在一个CPU上拥堵并不意味着该工作队列在其他CPU / NUMA节点上也被争用。

**返回**

如果拥堵，则为true，否则为false。

### unsigned int work\_busy(struct work\_struct \* work)

测试工作当前是否挂起或正在运行

**参数**

struct work\_struct \* work

要测试的工作

**说明**

测试work当前是否挂起或正在运行。该函数周围没有同步，测试结果不可靠，仅用于提供提示或调试。

**返回**

OR的WORK\_BUSY\_\*位掩码。

### void set\_worker\_desc(const char \* fmt, ...)

为当前工作项设置说明

**参数**

const char \* fmt

printf样式的格式字符串

...

格式字符串的参数

**说明**

可以由运行中的工作函数调用此函数来说明工作项的内容。如果工作任务被转储，则将一起打印此信息以帮助调试。说明最多可以为WORKER\_DESC\_LEN，包括尾随的'0'。

### long work\_on\_cpu(int cpu, long (\*fn) (void \*, void \* arg)

在特定的cpu上以线程上下文运行函数

**参数**

int cpu

要运行的CPU

long (\*)(void \*) fn

要运行的函数

void \* arg

该函数的参数

**说明**

由调用者负责确保CPU不会下线。调用者不得持有任何锁，这些锁会阻止fn完成。

**返回**

fn返回的值。

### long work\_on\_cpu\_safe(int cpu, long (\*fn) (void \*, void \* arg)

在特定的CPU上以线程上下文运行函数

**参数**

int cpu

要运行的CPU

long (\*)(void \*) fn

要运行的函数

void \* arg

函数参数

**说明**

禁用CPU热插拔并调用work\_on\_cpu()。调用者不得持有任何锁，这些锁将阻止fn完成。

**返回**

fn返回的值。

## 内部函数

### int wait\_task\_stopped(struct wait\_opts \*wo, int ptrace, struct task\_struct \*p)

等待TASK\_STOPPED或TASK\_TRACED

**参数**

struct wait\_opts \*wo

等待选项

int ptrace

等待ptrace

struct task\_struct \*p

要等待的任务

**说明**

处理p的sys\_wait4()工作，状态为TASK\_STOPPED或TASK\_TRACED。

**上下文**

read\_lock(tasklist\_lock)，如果返回为非零，则释放，grabs并释放p->sighand->siglock。

**返回**

如果等待条件不存在，则返回0，搜索其他等待条件应继续。非零返回，返回-errno表示失败，并且返回p的pid表示成功，意味着tasklist\_lock已释放，等待条件搜索应终止。

### bool task\_set\_jobctl\_pending(struct task\_struct \*task, unsigned long mask)

设置作业控制挂起位

**参数**

struct task\_struct \*task

目标任务

unsigned long mask

要设置的挂起位

**说明**

从task->jobctl清除掩码。mask必须是JOBCTL\_PENDING\_MASK | JOBCTL\_STOP\_CONSUME | JOBCTL\_STOP\_SIGMASK | JOBCTL\_TRAPPING的子集。如果设置了停止signo，则清除现有的signo。如果任务已被杀死或退出，则此函数会成为noop。

**上下文**

必须使用task->sighand->siglock调用。

**返回**

如果设置了掩码，则为true，如果任务正在死亡，则为false。

### void task\_clear\_jobctl\_trapping(struct task\_struct \*task)

清除作业控制陷阱位

**参数**

struct task\_struct \*task

目标任务

**说明**

如果设置了JOBCTL\_TRAPPING，则ptracer正在等待我们进入TRACED。清除并唤醒ptracer。请注意，我们不需要任何其他锁定。 task->siglock保证task->parent指向ptracer。

**上下文**

必须使用task->sighand->siglock调用。

### void task\_clear\_jobctl\_pending(struct task\_struct \*任务，unsigned long 掩码)

清除任务控制作业等待标志位

**参数**

struct task\_struct \*任务

目标任务

unsigned long 掩码

待清除的等待标志位

**说明**

从任务->jobctl中清除掩码。 掩码必须是JOBCTL\_PENDING\_MASK的子集。 如果正在清除JOBCTL\_STOP\_PENDING，则其他STOP位将一起清除。

如果清除掩码后没有停止或陷阱等待，此函数将调用task\_clear\_jobctl\_trapping（）。

**环境**

必须在持有task->sighand->siglock的情况下调用。

### bool task\_participate\_group\_stop(struct task\_struct \*task)

参与组停止

**参数**

struct task\_struct \*task

参与组停止的任务

**说明**

任务已设置JOBCTL\_STOP\_PENDING，并且正在参与组停止。 如果设置了JOBCTL\_STOP\_CONSUME，则清除组停止状态并使用组停止计数。 如果消耗完成组停止，则将设置适当的SIGNAL\_\*标志。

**上下文**

必须在持有task->sighand->siglock的情况下调用。

**返回**

如果应通知父项完成组停止，则返回true；否则返回false。

### void ptrace\_trap\_notify(struct task\_struct \*t)

安排陷阱通知ptracer

**参数**

struct task\_struct \*t

要通知的被跟踪者

**说明**

该函数安排了一个粘性的ptrace陷阱，该陷阱在下一个TRAP\_STOP时被清除，以通知ptracer事件。 t必须已被ptracer扣押。

如果t正在运行，则将执行STOP陷阱。 如果被救为停止并且ptracer正在侦听事件，则唤醒跟踪器以便它可以重新为新事件捕获。 如果受到其他陷阱的困扰，则STOP陷阱将在PTRACE\_CONT完成现有陷阱后最终被执行而不返回到用户空间。

**上下文**

必须在持有task->sighand->siglock的情况下调用。

### void do\_notify\_parent\_cldstop(struct task\_struct \*tsk，bool for\_ptracer，int why)

通知父项已停止/继续状态更改

**参数**

struct task\_struct \*tsk

报告状态更改的任务

bool for\_ptracer

通知是针对ptracer的

int why

CLD\_{CONTINUED | STOPPED | TRAPPED}进行报告

**说明**

通知tsk的父项已更改已停止/继续状态。 如果for\_ptracer为false，则tsk的组长通知其真正的父项。 如果为true，则tsk向tsk->parent报告，应该是ptracer。

**上下文**

必须在tasklist\_lock至少读取锁定的情况下调用。

### bool do\_signal\_stop(int signr)

处理SIGSTOP和其他停止信号的组停止

**参数**

int signr

引起组停止的信号

**说明**

如果尚未设置JOBCTL\_STOP\_PENDING，则初始化使用signr进行组停止并参与其中。 如果已经设置，请参与现有的组停止。 如果参与了组停止（因此睡眠），则使用siglock释放并返回true。

如果被跟踪，则此函数不处理停止本身。 相反，“ JOBCTL\_TRAP\_STOP”已安排，使用siglock未设置并返回false。 呼叫者必须确保后续发生中断陷阱处理。

**上下文**

必须在持有current->sighand->siglock的情况下调用，在真实返回时释放。

**返回**

如果组已停止取消或已安排ptrace陷阱，则返回false。如果参与了组停止，则返回true。

### void do\_**jobctl**\_trap(void)

处理ptrace jobctl陷阱

**参数**

void

没有参数

**说明**

当PT\_SEIZED时，它用于组停止和显式SEIZE / INTERRUPT陷阱。两者都会生成带有附带的siginfo的PTRACE\_EVENT\_STOP陷阱。如果停止，则exit\_code的低八位包含停止信号；否则，SIGTRAP。

当！PT\_SEIZED时，它仅用于组停止陷阱，其中停止信号编号为exit\_code，没有siginfo。

**上下文**

必须在当前->sighand->siglock的情况下调用，在中间睡眠后可能释放和重新获取。

### void signal\_delivered(struct ksignal \*ksig，int stepping)

在成功传递信号后调用以更新阻止的信号

**参数**

struct ksignal \*ksig

内核信号结构

int stepping

如果调试器使用单步或块步，则为非零

**说明**

成功传送信号后应调用此函数。它相应地更新已阻止的信号（ksig->ka.sa.sa\_mask始终被阻止），并且除非ksig->ka.sa.sa\_flags中设置了SA\_NODEFER，否则信号本身会被阻止。会通知跟踪。

### long sys\_restart\_syscall(void)

重新启动系统调用

**参数**

void

没有参数

### void set\_current\_blocked(sigset\_t \* newset)

更改current->blocked掩码

**参数**

sigset\_t \* newset

新掩码

**说明**

直接更改->blocked是错误的，应使用此帮助程序来确保进程不会错过我们要阻止的共享信号。

### long sys\_rt\_sigprocmask(int how，sigset\_t \_\_user \* nset，sigset\_t \_\_user \* oset，size\_t sigsetsize)

更改当前阻止信号的列表

**参数**

int how

是否添加，删除或设置信号

sigset\_t \_\_user \* nset

存储挂起信号

sigset\_t \_\_user \* oset

如果非空，则为信号掩码的先前值

size\_t sigsetsize

sigset\_t类型的大小

### long sys\_rt\_sigpending(sigset\_t \_\_user \* uset，size\_t sigsetsize)

检查已经被阻止而已经被触发的挂起信号

**参数**

sigset\_t \_\_user \* uset

存储待处理信号

size\_t sigsetsize

sigset\_t 类型或更大的大小

### int do\_sigtimedwait(const sigset\_t \* which，kernel\_siginfo\_t \* info，const struct timespec64 \* ts)

等待在 which 中指定的排队信号

**参数**

const sigset\_t \* which

等待的排队信号

kernel\_siginfo\_t \* info

如果非空，在此处返回信号的 siginfo

const struct timespec64 \* ts

进程时间暂停的上限

### long sys\_rt\_sigtimedwait(const sigset\_t \_\_user \* uthese，siginfo\_t \_\_user \* uinfo，const struct \_\_kernel\_timespec \_\_user \* uts，size\_t sigsetsize)

同步等待在 uthese 中指定的排队信号

**参数**

const sigset\_t \_\_user \* uthese

等待的排队信号

siginfo\_t \_\_user \* uinfo

如果非空，在此处返回信号的 siginfo

const struct \_\_kernel\_timespec \_\_user \* uts

进程时间暂停的上限

size\_t sigsetsize

sigset\_t 类型的大小

### long sys\_kill(pid\_t pid，int sig)

向进程发送信号

**参数**

pid\_t pid

进程的 PID

int sig

要发送的信号

### long sys\_tgkill(pid\_t tgid，pid\_t pid，int sig)

向一个特定的线程发送信号

**参数**

pid\_t tgid

线程所在线程组的 ID

pid\_t pid

线程的 PID

int sig

要发送的信号

该系统调用还会检查 tgid，并返回 -ESRCH，即使 PID 存在但不再属于目标进程。该方法解决了线程退出和 PID 被重新使用的问题。

### long sys\_tkill(pid\_t pid，int sig)

向一个特定的任务发送信号

**参数**

pid\_t pid

任务的 PID

int sig

要发送的信号

仅向一个任务发送信号，即使它是 CLONE\_THREAD 任务。

### long sys\_rt\_sigqueueinfo(pid\_t pid，int sig，siginfo\_t \_\_user \* uinfo)

向信号发送信号信息

**参数**

pid\_t pid

线程的 PID

int sig

要发送的信号

siginfo\_t \_\_user \* uinfo

要发送的信号信息

### long sys\_sigpending(old\_sigset\_t \_\_user \* uset)

检查待处理信号

**参数**

old\_sigset\_t \_\_user \* uset

待处理信号掩码的返回位置

### long sys\_sigprocmask(int how，old\_sigset\_t \_\_user \* nset，old\_sigset\_t \_\_user \* oset)

检查和更改阻塞的信号

**参数**

int how

是添加、删除还是设置信号

old\_sigset\_t \_\_user \* nset

要添加或删除的信号（如果非空）

old\_sigset\_t \_\_user \* oset

如果非空，则为信号掩码的先前值

**说明**

一些平台具有其自己的具有特殊参数的版本；其他平台仅支持 sys\_rt\_sigprocmask。

### long sys\_rt\_sigaction(int sig，const struct sigaction \_\_user \* act，struct sigaction \_\_user \* oact，size\_t sigsetsize)

更改进程执行的操作

**参数**

int sig

要发送的信号

const struct sigaction \_\_user \* act

新的 sigaction

struct sigaction \_\_user \* oact

用于保存先前的 sigaction

size\_t sigsetsize

sigset\_t 类型的大小

### long sys\_rt\_sigsuspend(sigset\_t \_\_user \* unewset，size\_t sigsetsize)

将信号掩码替换为 unewset 值，直到接收信号

**参数**

sigset\_t \_\_user \* unewset

新的信号掩码值

size\_t sigsetsize

sigset\_t 类型的大小

### kthread\_create

kthread\_create (threadfn，data，namefmt，arg...)

在当前节点上创建一个 kthread。

**参数**

threadfn

要在线程中运行的函数

data

线程 fn() 的数据指针

namefmt

线程名称的 printf 样式格式字符串

arg...

namefmt 的参数。

**说明**

此宏将在当前节点上创建一个 kthread，并将其保留在停止状态。这只是 kthread\_create\_on\_node() 的辅助功能；有关更多详细信息，请参阅那里的文档。

### kthread\_run

kthread\_run (threadfn，data，namefmt，...)

创建和唤醒线程。

**参数**

threadfn

要在信号到达(current)之前运行的函数。

data

线程 fn 的数据 ptr。

namefmt

线程的 printf-style 名称。

...

可变参数

**说明**

方便的 kthread\_create() 包装器，后跟 wake\_up\_process()。返回 kthread 或 ERR\_PTR(-ENOMEM)。

### bool kthread\_should\_stop(void)

现在应该返回这个 kthread 吗？

**参数**

void

没有参数

**说明**

当某人在您的 kthread 上调用 kthread\_stop() 时，它将被唤醒，这将返回 true。然后您应该返回，您的返回将被传递到 kthread\_stop()。

### bool kthread\_should\_park(void)

现在应该将此 kthread 停泊吗？

**参数**

void

没有参数

**说明**

当某人在您的 kthread 上调用 kthread\_park() 时，它将被唤醒，这将返回 true。然后，您应该进行必要的清理并调用 kthread\_parkme()，类似于 kthread\_should\_stop()，但这会保持线程处于活动状态并处于停泊位置。 kthread\_unpark() “重新启动”线程并再次调用线程函数。

### bool kthread\_freezable\_should\_stop(bool \* was\_frozen)

现在应该返回此可冻结的 kthread 吗？

**参数**

bool \* was\_frozen

可选的输出参数，表示当前是否已被冻结

**说明**

freezable kthreads 的 kthread\_should\_stop()，如果必要会进入冰箱。此函数在 kthread\_stop() / 冰箱死锁和可冻结的 kthread 中是安全的，并且可冻结的 kthread 应该使用此函数而不是直接调用 try\_to\_freeze()。

### struct task\_struct \*kthread\_create\_on\_node(int (\*threadfn)(void \* data)，void \* data，int node，const char namefmt[]，...)

创建一个 kthread。

**参数**

int (\*threadfn)(void \* data)

要运行直到 signal\_pending(current) 的函数。

void \*data

threadfn 的数据指针。

int node

为线程分配任务和线程结构的节点。

const char namefmt[]

线程的 printf 格式化名称。

...

变量参数

**说明**

该帮助函数创建并命名内核线程。该线程将停止使用 wake\_up\_process() 来启动它。另请参阅 kthread\_run()。新线程具有 SCHED\_NORMAL 策略，并与所有 CPU 亲和。

如果要将线程绑定到特定 CPU，则在节点中指定其节点，以获取 kthread 堆栈的 NUMA 亲和性，否则指定 NUMA\_NO\_NODE。唤醒后，线程将在将 data 作为参数的情况下运行 threadfn()。threadfn() 可以直接返回，如果它是一个独立的线程，没有人会调用 kthread\_stop()，或者在 'kthread\_should\_stop()' 为 true 时返回（这意味着已调用 kthread\_stop()）。返回应为零或负错误号；它将被传递给 kthread\_stop()。

返回 task\_struct 或 ERR\_PTR(-ENOMEM) 或 ERR\_PTR(-EINTR)。

### void kthread\_bind(struct task\_struct \*p, unsigned int cpu)

将新创建的 kthread 绑定到 CPU。

**参数**

struct task\_struct \*p

由 kthread\_create() 创建的线程。

unsigned int cpu

要在其上运行 kthread 的 CPU（可能不在线，必须可用）。

**说明**

此函数等同于 set\_cpus\_allowed()，除了 cpu 不需要在线，并且线程必须已停止（即刚刚从 kthread\_create() 返回）。

### void kthread\_unpark(struct task\_struct \*k)

唤醒由 kthread\_create() 创建的线程。

**参数**

struct task\_struct \*k

由 kthread\_create() 创建的线程。

**说明**

将 kthread\_should\_park() 设置为返回 false，唤醒线程，并等待其返回。如果线程被标记为 percpu，则绑定到 CPU 后。

### int kthread\_park(struct task\_struct \*k)

挂起由 kthread\_create() 创建的线程。

**参数**

struct task\_struct \*k

由 kthread\_create() 创建的线程。

**说明**

将 kthread\_should\_park() 设置为返回 true，唤醒它并等待它返回。这也可以在 kthread\_create() 后调用，而不是调用 wake\_up\_process()线程将停留在不调用 threadfn() 的情况下。

如果线程停在 0，则返回 0；如果线程已退出，则返回 -ENOSYS。如果由 kthread 自身调用，则只设置停放位。

### int kthread\_stop(struct task\_struct \*k)

停止由 kthread\_create() 创建的线程。

**参数**

struct task\_struct \*k

由 kthread\_create() 创建的线程。

**说明**

将 kthread\_should\_stop() 设置为返回 true，唤醒它并等待它退出。这也可以在 kthread\_create() 后调用，而不是调用 wake\_up\_process()线程将停止而不调用 threadfn()。

如果 threadfn() 可以调用 kthread\_exit()，则调用者必须确保 task\_struct 不能消失。

返回 threadfn() 的结果，如果从未调用 wake\_up\_process()，则返回 -EINTR。

### int kthread\_worker\_fn(void \*worker\_ptr)

kthread 处理 kthread\_worker 的函数

**参数**

void \*worker\_ptr

已初始化的 kthread\_worker 的指针

**说明**

此函数实现 kthread worker 的主循环。它处理 work\_list，直到使用 kthread\_stop() 停止。当队列为空时，它会睡眠。

处理工作时，不允许保留任何锁、禁用抢占或中断。在一个工作完成之后、新工作开始之前，定义了一个安全冻结点。

还需要注意的是，作业不能同时由多个工作线程处理。另请参阅 kthread\_queue\_work()。

### struct kthread\_worker \*kthread\_create\_worker(unsigned int flags, const char namefmt[], ...)

创建 kthread worker。

**参数**

unsigned int flags

修改 worker 默认行为的标志

const char namefmt[]

kthread worker 的 printf 格式名称（任务）。

...

变量参数

**说明**

在成功时返回分配的 worker 的指针，在需要时未分配的结构返回 ERR\_PTR(-ENOMEM)，而调用者在遇到致命信号后则返回 ERR\_PTR(-EINTR)。

### struct kthread\_worker \*kthread\_create\_worker\_on\_cpu(int cpu, unsigned int flags, const char namefmt[], ...)

创建 kthread worker 并将其绑定到给定 CPU 及其相关联的 NUMA 节点。

**参数**

int cpu

CPU 数。

unsigned int flags

修改 worker 默认行为的标志

const char namefmt[]

kthread worker 的 printf 格式名称（任务）。

...

变量参数

**说明**

如果要将 kthread worker 绑定到给定的 CPU 和相关联的 NUMA 节点，则使用有效的 CPU 数。一个良好的实践是将 CPU 数字添加到 worker 名称中。例如，使用 kthread\_create\_worker\_on\_cpu(cpu,“helper/d”,cpu)。

在成功时返回分配的 worker 的指针，未分配所需结构时返回 ERR\_PTR(-ENOMEM)，工作线程被 SIGKILL 时则返回 ERR\_PTR(-EINTR)。

### bool kthread\_queue\_work(struct kthread\_worker \*worker, struct kthread\_work \*work)

队列 kthread\_work。

**参数**

struct kthread\_worker \*worker

目标 kthread\_worker。

struct kthread\_work \*work

要排队的 kthread\_work。

**说明**

将工作队列排队到工作处理器任务以进行异步执行。必须已使用 kthread\_worker\_create() 创建 task。如果成功排队，则返回 true；如果已经有待处理的，则返回 false。

如果需要被其他工作者使用，则重新初始化工作。例如，当工作者停止并再次启动时。

### void kthread\_delayed\_work\_timer\_fn（struct timer\_list\* t）

定时器到期时排队关联的kthread延迟工作的回调。

**参数**

struct timer\_list \* t

指向已过期的计时器的指针

**说明**

函数的格式由struct timer\_list定义。它应该从irqsafe timer中调用，并且中断已关闭。

### bool kthread\_queue\_delayed\_work（struct kthread\_worker \* worker，struct kthread\_delayed\_work \* dwork，unsigned long delay）

延迟一段时间后排队关联的kthread工作。

**参数**

struct kthread\_worker \* worker

目标kthread\_worker

struct kthread\_delayed\_work \* dwork

要排队的kthread\_delayed\_work

unsigned long delay

排队前要等待的jiffies数

**说明**

如果工作没有挂起，则启动一个定时器，该定时器将在给定的延迟后排队工作。如果延迟为零，则立即排队工作。

**返回**

如果工作已经挂起，则为false。这意味着计时器正在运行或者工作已经排队。否则返回true。

### void kthread\_flush\_work（struct kthread\_work \* work）

刷新kthread\_work

**参数**

struct kthread\_work \* work

要刷新的工作

**说明**

如果工作已排队或正在执行，则等待它执行完成。

### bool kthread\_mod\_delayed\_work（struct kthread\_worker \* worker，struct kthread\_delayed\_work \* dwork，unsigned long delay）

修改或排队kthread延迟工作的延迟

**参数**

struct kthread\_worker \* worker

要使用的kthread worker

struct kthread\_delayed\_work \* dwork

要排队的kthread\_delayed\_work

unsigned long delay

排队前要等待的jiffies数

**说明**

如果dwork为空闲状态，则相当于kthread\_queue\_delayed\_work（）。否则，修改dwork的计时器，使其在延迟后过期。如果延迟为零，则保证立即排队工作。

**返回**

如果dwork已挂起并且其计时器已修改，则返回true；否则返回false。

一个特殊情况是并行取消工作。它可能由realkthread\_cancel\_delayed\_work\_sync（）或另一个kthread\_mod\_delayed\_work（）调用引起。我们让其他命令获胜，并在此处返回true。返回可用于引用计数，排队工作的数量保持不变。不管怎样，调用者应以合理的方式同步这些操作。

此函数可以从任何上下文安全调用，包括IRQ处理程序。有关详细信息，请参见\_\_kthread\_cancel\_work（）和kthread\_delayed\_work\_timer\_fn（）。

### bool kthread\_cancel\_work\_sync（struct kthread\_work \* work）

取消kthread work并等待其执行完成

**参数**

struct kthread\_work \* work

要取消的kthread work

**说明**

取消工作并等待其执行完成。即使工作重新排队，也可以使用此函数。从此函数返回时，保证work不会挂起或在任何CPU上执行。

不得使用kthread\_cancel\_work\_sync（delayed\_work-> work）来取消延迟工作。请改用kthread\_cancel\_delayed\_work\_sync（）。

调用者必须确保最后排队work的工作者在此函数返回之前不能被销毁。

**返回**

如果work已挂起，则返回true；否则返回false。

### bool kthread\_cancel\_delayed\_work\_sync（struct kthread\_delayed\_work \* dwork）

取消kthread延迟工作并等待其执行完成。

**参数**

struct kthread\_delayed\_work \* dwork

要取消的kthread延迟工作

**说明**

这是面向延迟工作的kthread\_cancel\_work\_sync（）。

**返回**

如果dwork已挂起，则返回true；否则返回false。

### void kthread\_flush\_worker（struct kthread\_worker \* worker）

清除kthread\_worker上的所有当前工作

**参数**

struct kthread\_worker \* worker

要清除的工作者

**说明**

等待工作者上的所有正在执行或挂起的工作完成。

### void kthread\_destroy\_worker（struct kthread\_worker \* worker）

销毁kthread\_worker

**参数**

struct kthread\_worker \* worker

要销毁的工作者

**说明**

清除并销毁worker。简单的清除就足够了，因为kthread worker API仅用于简单的场景。不需要多步状态机。

### void kthread\_associate\_blkcg（struct cgroup\_subsys\_state \* css）

将blkcg关联到当前kthread

**参数**

struct cgroup\_subsys\_state \* css

cgroup信息

**说明**

当前线程必须是kthread。该线程代表其他线程运行作业。在某些情况下，我们希望作业附加原始线程的cgroup信息，而不是当前线程的cgroup信息。此函数将原始线程的cgroup信息存储在当前kthread上下文中，以供以后检索。

### struct cgroup\_subsys\_state \* kthread\_blkcg（void）

获取当前kthread关联的blkcg css

**参数**

void

无参数

**说明**

当前线程必须是kthread。

## 引用计数

### struct refcount\_struct

用于引用计数的原子类型的变体

**定义**

struct refcount\_struct {

atomic\_t refs;

};

**成员**

refs

计数器字段

**说明**

计数器饱和于UINT\_MAX，并且一旦达到这个值就不会再动了。这避免了计数器包装并导致“虚假”的使用后释放错误。

### void refcount\_set（refcount\_t \* r，int n）

设置引用计数的值

**参数**

refcount\_t \* r

引用计数

int n

要将引用计数设置为的值

### unsigned int refcount\_read(const refcount\_t \* r)

获取引用计数的值

**参数**

const refcount\_t \* r

引用计数

返回是引用计数的值

### bool refcount\_add\_not\_zero\_checked（int i，refcount\_t \*r）

将一个值添加到refcount中，除非它为0

**参数**

int i

要添加到refcount中的值

refcount\_t \* r

引用计数

**说明**

饱和时为REFCOUNT\_SATURATED和WARN。

不提供内存订购，假定调用者已保证对象内存稳定（RCU等）。它确实提供了控制依赖性，从而为将来的存储排序。请参阅顶部的注释。

不推荐在正常引用计数用例中使用此功能，在这些情况下，应使用refcount\_inc（）或其变体之一来增加引用计数。

**返回**

如果传递的引用计数为0，则为false，否则为true

### void refcount\_add\_checked（int i，refcount\_t \* r）

将一个值添加到refcount

**参数**

int i

要添加到引用计数中的值

refcount\_t \* r

引用计数

**说明**

类似于atomic\_add（），但饱和时为REFCOUNT\_SATURATED和WARN。

不提供内存订购，假定调用者已保证对象内存稳定（RCU等）。它确实提供了控制依赖性，从而为将来的存储排序。请参阅顶部的注释。

不推荐在正常引用计数用例中使用此功能，在这些情况下，应使用refcount\_inc（）或其变体之一来增加引用计数。

### bool refcount\_inc\_not\_zero\_checked（refcount\_t \* r）

增加引用计数，除非它为0

**参数**

refcount\_t \* r

要增加的引用计数

**说明**

类似于atomic\_inc\_not\_zero（），但饱和时为REFCOUNT\_SATURATED和WARN。

不提供内存订购，假定调用者已保证对象内存稳定（RCU等）。它确实提供了控制依赖性，从而为将来的存储排序。请参阅顶部的注释。

**返回**

如果增量成功，则为true，否则为false

### void refcount\_inc\_checked（refcount\_t \* r）

增加引用计数

**参数**

refcount\_t \* r

要增加的引用计数

**说明**

类似于atomic\_inc（），但饱和时为REFCOUNT\_SATURATED和WARN。

不提供内存订购，假定调用者已保留了对象的引用。

如果refcount为0，则会发出警告，因为这表示可能存在使用后释放的情况。

### bool refcount\_sub\_and\_test\_checked（int i，refcount\_t \* r）

从refcount中减去并测试是否为0

**参数**

int i

要从refcount中减去的数量

refcount\_t \* r

引用计数

**说明**

类似于atomic\_dec\_and\_test（），但会显示警告，返回false，并最终泄漏，当REFCOUNT\_SATURATED饱和时，将无法减少。

提供释放内存排序，以使先前的加载和存储在之前完成，并在成功时提供获取排序，以使free（）必须在之后。

不推荐在正常引用计数用例中使用此功能，在这些情况下，应删除refcount\_dec（）或其变体之一来减少引用计数。

**返回**

如果结果refcount为0，则为true，否则为false

### bool refcount\_dec\_and\_test\_checked（refcount\_t \* r）

递减refcount并测试是否为0

**参数**

refcount\_t \* r

引用计数

**说明**

类似于atomic\_dec\_and\_test（），它会在REFCOUNT\_SATURATED饱和时发出警告，并无法减少。

提供释放内存排序，以使先前的加载和存储在之前完成，并在成功时提供获取排序，以使free（）必须在之后。

**返回**

如果结果refcount为0，则为true，否则为false

### void refcount\_dec\_checked（refcount\_t \* r）

递减refcount

**参数**

refcount\_t \* r

引用计数

**说明**

类似于atomic\_dec（），如果饱和时为REFCOUNT\_SATURATED并发出警告。

提供释放内存排序，以使先前的加载和存储在之前完成。

### bool refcount\_dec\_if\_one（refcount\_t \* r）

如果为1，则递减refcount

**参数**

refcount\_t \* r

引用计数

**说明**

没有atomic\_t对应项，它会尝试1->0的转换并返回成功。

像所有减量操作一样，它提供释放内存顺序并提供控制依赖性。

它可以像try-delete运算符一样使用；提供了此明确情况并且在通用情况下不提供cmpxchg，因为那会允许实现不安全操作。

**返回**

如果结果refcount为0，则为true，否则为false

### bool refcount\_dec\_not\_one（refcount\_t \* r）

如果不是1，则递减refcount

**参数**

refcount\_t \* r

引用计数

**说明**

没有atomic\_t对应项，它会递减，除非其值为1，在这种情况下，它将返回false。

通常会这样做atomic\_add\_unless（var，-1，1）

**返回**

如果递减操作成功，则为true，否则为false

### bool refcount\_dec\_and\_mutex\_lock（refcount\_t \* r，struct mutex \* lock）

如果能够将refcount减到0，则返回保持互斥锁

**参数**

refcount\_t \*r

引用计数

struct mutex \*lock

要锁定的互斥锁

**说明**

与atomic\_dec\_and\_mutex\_lock()类似，当饱和时，会发出underflow的警告并不能够减少到RECOUNT\_SATURATED。

提供释放内存顺序，先前的加载和存储已完成，并提供控制依赖关系，free()必须在之后进行。请参考顶部的注释。

**返回**

如果能够将引用计数减少到0，则返回true和保持互斥锁；否则返回false

### bool refcount\_dec\_and\_lock(refcount\_t \*r, spinlock\_t \*lock)

减小引用计数到0就保持自旋锁

**参数**

refcount\_t \*r

引用计数

spinlock\_t \*lock

要锁定的自旋锁

**说明**

与atomic\_dec\_and\_lock()类似，当饱和时，会发出underflow的警告并不能够减少到RECOUNT\_SATURATED。

提供释放内存顺序，先前的加载和存储已完成，并提供控制依赖关系，free()必须在之后进行。请参考顶部的注释。

**返回**

如果能够将引用计数减少到0，则返回true和保持自旋锁；否则返回false

### bool refcount\_dec\_and\_lock\_irqsave(refcount\_t \*r, spinlock\_t \*lock, unsigned long \*flags)

在减小引用计数到0时，会保存禁用中断的自旋锁

**参数**

refcount\_t \*r

引用计数

spinlock\_t \*lock

要锁定的自旋锁

unsigned long \*flags

如果已获取，则保存irq-flags

**说明**

与上述refcount\_dec\_and\_lock()相同，只是自旋锁使用禁用中断进行获取。

**返回**

如果能够将引用计数减少到0，则返回true和保持自旋锁；否则返回false

## 原子操作

### int arch\_atomic\_read(const atomic\_t \*v)

读取原子变量

**参数**

const atomic\_t \*v

类型为atomic\_t的指针

**说明**

原子地读取v的值。

### void arch\_atomic\_set(atomic\_t \*v, int i)

设置原子变量

**参数**

atomic\_t \*v

类型为atomic\_t的指针

int i

要设置的值

**说明**

原子地将v的值设置为i。

### void arch\_atomic\_add(int i, atomic\_t \*v)

将整数添加到原子变量

**参数**

int i

要添加的整数值

atomic\_t \*v

类型为atomic\_t的指针

**说明**

原子地将i加到v。

### void arch\_atomic\_sub(int i, atomic\_t \*v)

从原子变量中减去整数

**参数**

int i

要减去的整数值

atomic\_t \*v

类型为atomic\_t的指针

**说明**

原子地从v中减去i。

### bool arch\_atomic\_sub\_and\_test(int i, atomic\_t \*v)

从变量中减去值并测试结果

**参数**

int i

要减去的整数值

atomic\_t \*v

类型为atomic\_t的指针

**说明**

原子地从v中减去i，如果结果为零，则返回true，否则返回false。

### void arch\_atomic\_inc(atomic\_t \*v)

递增原子变量

**参数**

atomic\_t \*v

类型为atomic\_t的指针

**说明**

原子地将v增加1。

### void arch\_atomic\_dec(atomic\_t \*v)

递减原子变量

**参数**

atomic\_t \*v

类型为atomic\_t的指针

**说明**

原子地将v减1。

### bool arch\_atomic\_dec\_and\_test(atomic\_t \*v)

递减并测试

**参数**

atomic\_t \*v

类型为atomic\_t的指针

**说明**

原子地将v减1，如果结果为0，则返回true，否则返回false。

### bool arch\_atomic\_inc\_and\_test(atomic\_t \*v)

递增并测试

**参数**

atomic\_t \*v

类型为atomic\_t的指针

**说明**

原子地将v增加1，如果结果为零，则返回true，否则返回false。

### bool arch\_atomic\_add\_negative(int i, atomic\_t \*v)

添加并测试值是否为负数

**参数**

int i

要添加的整数值

atomic\_t \*v

类型为atomic\_t的指针

**说明**

原子地将i加到v中，并在结果为负数时返回true，在结果大于或等于零时返回false。

### int arch\_atomic\_add\_return(int i, atomic\_t \*v)

添加整数并返回

**参数**

int i

要添加的整数值

atomic\_t \*v

类型为atomic\_t的指针

**说明**

原子地将i加到v中，并返回i + v

### int arch\_atomic\_sub\_**r**eturn(int i, atomic\_t \*v)

减去整数并返回

**参数**

int i

要减去的整数值

atomic\_t \*v

类型为atomic\_t的指针

**说明**

原子地从v中减去i，并返回v - i

## 内核对象操作

### char \*kobject\_get\_path(struct kobject \*kobj, gfp\_t gfp\_mask)

分配内存并填充kobj的路径。

**参数**

struct kobject \*kobj

要构建路径的kobject

gfp\_t gfp\_mask

用于分配路径的分配类型

**返回**

新分配的内存，调用者必须使用kfree()释放。

### int kobject\_set\_name(struct kobject \*kobj, const char \*fmt, ...)

设置kobject的名称。

**参数**

struct kobject \*kobj

要设置名称的struct kobject

const char \*fmt

用于构建名称的格式字符串

...

可变参数

**说明**

这将设置kobject的名称。如果已将kobject添加到系统中，则必须调用kobject\_rename()以更改kobject的名称。

### void kobject\_init(struct kobject \*kobj, const struct kobj\_type \*ktype)

初始化kobject结构。

**参数**

struct kobject \*kobj

要初始化的kobject指针

const struct kobj\_type \*ktype

此kobject的ktype的指针。

**说明**

此函数将正确初始化一个 kobject，以便可以将其传递给 kobject\_add() 调用。

调用此函数后，必须通过调用 kobject\_put() 清理 kobject，而不是直接调用 kfree，以确保所有内存得到正确清理。

### int kobject\_add(struct kobject \*kobj, struct kobject \*parent, const char \*fmt, ...)

主要的 kobject 添加函数。

**参数**

struct kobject \*kobj

要添加的 kobject

struct kobject \*parent

指向 kobject 的父级的指针。

const char \*fmt

用于对 kobject 命名的格式。

...

变量参数

**说明**

此函数将设置 kobject 名称，并将其添加到 kobject 层次结构中。

如果设置了 parent，则 kobj 的父级将设置为它。如果 parent 为空，则 kobj 的父级将设置为分配给此 kobject 的 kset 关联的 kobject。如果没有将 kset 分配给 kobject，则 kobject 将位于 sysfs 树的根目录中。

如果此函数返回错误，则必须调用 kobject\_put() 以正确清理与该对象关联的内存。在任何情况下，都不应该直接使用 kfree() 调用来释放传递给此函数的 kobject，因为可能会泄漏内存。

请注意，此调用不会创建“add”uevent，调用者应该设置该对象的所有必要 sysfs 文件，然后调用 kobject\_uevent()，并使用 UEVENT\_ADD 参数，以确保用户空间正确通知此 kobject 的创建。

### int kobject\_init\_and\_add(struct kobject \*kobj, const struct kobj\_type \*ktype, struct kobject \*parent, const char \*fmt, ...)

初始化 kobject 结构并将其添加到 kobject 层次结构中。

**参数**

struct kobject \*kobj

要初始化的 kobject 指针

const struct kobj\_type \*ktype

此 kobject 的 ktype 指针。

struct kobject \*parent

此 kobject 的父级指针。

const char \*fmt

kobject 的名称。

...

可变参数

**说明**

此函数将调用 kobject\_init() 和 kobject\_add()。

### int kobject\_rename(struct kobject \*kobj, const char \*new\_name)

更改对象的名称。

**参数**

struct kobject \*kobj

需要更改名称的对象。

const char \*new\_name

对象的新名称。

**说明**

调用者有责任在同一 kobject 上的两个不同 kobject\_rename 调用之间提供互斥，并确保 new\_name 是有效的并且不会与其他 kobject 冲突。

### int kobject\_move(struct kobject \*kobj, struct kobject \*new\_parent)

将对象移动到另一个父级。

**参数**

struct kobject \*kobj

需要移动的对象。

struct kobject \*new\_parent

对象的新父级（可以为空）。

### void kobject\_del(struct kobject \*kobj)

从层次结构中取消链接 kobject。

**参数**

struct kobject \*kobj

对象。

**说明**

这是应该调用以通过 kobject\_add() 成功添加对象并删除对象的函数。

### struct kobject \*kobject\_get(struct kobject \*kobj)

增加对象的引用计数。

**参数**

struct kobject \*kobj

对象。

### void kobject\_put(struct kobject \*kobj)

减少对象的引用计数。

**参数**

struct kobject \*kobj

对象。

**说明**

减少引用计数，如果为 0，则调用 kobject\_cleanup()。

### struct kobject \*kobject\_create\_and\_add(const char \*name, struct kobject \*parent)

动态创建 struct kobject 并在 sysfs 中注册它。

**参数**

const char \*name

kobject 的名称

struct kobject \*parent

该 kobject 的父 kobject，如果有的话。

**说明**

此函数动态创建 kobject 结构并在 sysfs 中注册它。完成此结构后，请调用 kobject\_put()，这个结构将在不再使用它时被动态释放。

如果无法创建 kobject，则返回 NULL。

### int kset\_register(struct kset \*k)

初始化并添加 kset。

**参数**

struct kset \*k

kset。

### void kset\_unregister(struct kset \*k)

删除 kset。

**参数**

struct kset \*k

kset。

### struct kobject \*kset\_find\_obj(struct kset \*kset, const char \*name)

在 kset 中搜索对象。

**参数**

struct kset \*kset

我们要查找的 kset。

const char \*name

对象的名称。

**说明**

通过 kset->subsys 锁和迭代 kset->list，查找匹配的 kobject，如果找到匹配对象，则引用该对象并返回它。

### struct kset \*kset\_create\_and\_add(const char \*name, const struct kset\_uevent\_ops \*uevent\_ops, struct kobject \*parent\_kobj)

动态创建 struct kset 并将其添加到 sysfs 中。

**参数**

const char \*name

kset 的名称

const struct kset\_uevent\_ops \*uevent\_ops

kset\_uevent\_ops 的 struct kset\_uevent\_ops

struct kobject \*parent\_kobj

该 kset 的父 kobject，如果有的话。

**说明**

此函数动态创建 kset 结构并在 sysfs 中注册它。完成此结构后，请调用 kset\_unregister()，这个结构将在不再使用它时被动态释放。

如果无法创建 kset，则返回 NULL。

## 内核实用函数

### REPEAT\_BYTE

REPEAT\_BYTE（x）

将值x多次重复为无符号长整型值

**参数**

x——要重复的值

**注**

没有检查x> 0xff;较大的值产生奇怪的结果。

### ARRAY\_SIZE

ARRAY\_SIZE(arr)

获取数组arr中的元素数量

**参数**

arr——要调整大小的数组

### round\_up（x，y）

向上舍入到下一个指定的2的幂次

**参数**

x

要舍入的值

y

要舍入到的倍数（必须是2的幂次）

**说明**

将x舍入到接下来的y的倍数（必须是2的幂次）。要执行任意舍入，请使用下面的roundup（）。

### round\_down（x，y）

向下舍入到下一个指定的2的幂次

**参数**

x

要舍入的值

y

要向下舍入到的倍数（必须是2的幂次）

**说明**

将x向下舍入到接下来的y的倍数（必须是2的幂次）。要执行任意向下舍舍，请使用下面的“rounddown（）”。

### FIELD\_SIZEOF（t，f）

获取struct 字段的大小

**参数**

t

目标struct

f

目标struct 的字段

**返回**

在没有声明t的实例的结构定义中，f的大小。

### roundup（x，y）

向上舍入到下一个指定的倍数

**参数**

x

要上调的值

y

要舍入到的倍数

**说明**

将x向上舍入到接下来的y的倍数。如果y始终是2的幂次，则考虑使用更快的round\_up（）。

这里的“const”防止gcc-3.3调用\_\_divdi3

### rounddown（x，y）

向下舍入到下一个指定​​的倍数

**参数**

x

要舍入的值

y

要向下舍入到的倍数

**说明**

将x向下舍入到接下来的y的倍数。如果y始终是2的幂次，则考虑使用更快的round\_down（）。

### upper\_32\_bits

upper\_32\_bits（n）

返回数字的位32-63

**参数**

n——我们正在访问的数字

**说明**

64位或32位数量的基本右移。使用此方法可以消除“右移计数>类型宽度”的警告，当该数量为32位时。

### lower\_32\_bits

lower\_32\_bits（n）

返回数字0-31位

**参数**

n——我们正在访问的数字

可能入睡

### might\_sleep（）

可以睡觉的函数注释

**参数**

**说明**

如果在原子上下文（自旋锁，irq-handler，…）中执行，则此宏将打印堆栈跟踪。还可以使用non\_block\_start（）和non\_block\_end（）对不允许阻止的其他部分进行注释。

这是一种有用的调试帮助，可以尽早捕捉问题，并且在不应该睡眠的调用函数稍后发生睡眠时不会受到伤害。

### abs（x）

返回参数的绝对值

**参数**

x

价值。如果它是无符号类型，则首先将其转换为有符号类型。 char被视为带符号的（无论实际上是否），但是保留了宏的返回类型为char。

回报

x的绝对值。

### u32 reciprocal\_scale（u32 val，u32 ep\_ro）

将“规模”值缩放到[0，ep\_ro）范围内

**参数**

u32 val

价值

u32 ep\_ro

右打开间隔端点

**说明**

执行“倒数乘法”以便将值缩放到范围[0，ep\_ro）内，其中上部间隔端点为右打开。这很有用，例如用于访问包含ep\_ro元素的数组的索引。想像一下它是一种模量，只是结果不是模除的结果。 ;)请注意，如果初始输入是小值，则结果将返回0。

**返回**

基于val的结果，范围为[0，ep\_ro）。

### int kstrtoul（const char \* s，unsigned int base，unsigned long \* res）

将字符串转换为无符号长整数

**参数**

const char \* s

字符串的开始。字符串必须以null终止，并且也可以在其终止null之前包括单个换行符。第一个字符还可以是加号，但不能是减号。

unsigned int base

要使用的数字基数。支持的最大基数是16。如果将基数给定为0，则字符串的基数将自动检测到具有常规语义-如果以0x开头，则该数字将解析为十六进制（不区分大小写），如果它以0开头，则将将解析为八进制数。否则将解析为十进制。

unsigned long \* res

成功转换的结果的写入位置。

**说明**

返回成功的0，对于溢出的-ERANGE和对于解析错误的-EINVAL。用作过时的simple\_strtoull的替代品。必须检查返回代码。

### int kstrtol（const char \* s，unsigned int base，long \* res）

将字符串转换为长整数

**参数**

const char \* s

字符串的起始位置。该字符串必须以空字符结尾，也可以在终止空字符之前包含单个换行符。第一个字符也可能是加号或减号。

unsigned int base

要使用的数字进制。支持的最大进制是16。如果base的值为0，则将以传统语义自动检测字符串的进制——如果以0x开头，则解析为16进制（不区分大小写），如果以0开头，则解析为八进制数。否则，将解析为十进制。

long \* res

成功转换后结果的存储位置。

**说明**

返回0表示成功，-ERANGE表示溢出，-EINVAL表示解析错误。用作simple\_strtoull的替代品。必须检查返回。

### trace\_printk

trace\_printk(fmt，...)

ftrace缓冲区中的printf格式化

**参数**

fmt

打印的printf格式

...

可变参数

**注意**

\_\_trace\_printk是trace\_printk()的内部函数，ip通过trace\_printk()宏传递。

该函数允许内核开发人员调试快速路径段，对于printk不合适的情况。通过在代码中散布各种类似于printk的跟踪，开发人员可以快速地看到出现问题的地方。

这只是开发人员的调试工具。请勿在您的代码中散布trace\_printk。（使用trace\_printk()时会使用专用缓冲区分配额外的内存。）

这里做了一个小优化技巧。如果只有一个参数，不需要扫描字符串以找到printf格式。trace\_puts()就足够了。但是当trace\_printk()只有一个参数时，我们如何利用使用trace\_puts()呢？通过将参数字符串化并检查其大小，我们可以告诉是否有参数。当没有参数时，\_\_stringify((\_\_VA\_ARGS\_\_))将变成“(）0”，大小为3。否则，大小会变大。我们只需要将一个字符串**定义**为此，然后取其大小并将其与3进行比较。如果更大，使用do\_trace\_printk()，否则将其优化为trace\_puts()。然后让gcc优化其余部分。

### trace\_puts

trace\_puts(str)

将字符串写入ftrace缓冲区

**参数**

str

要记录的字符串

**注意**

\_\_trace\_bputs是trace\_puts的内部函数，并通过trace\_puts宏传递ip。

这类似于trace\_printk()，但适用于那些处理打印格式仍然太多的快速路径，开发人员希望其出现“Heisenbug”效应的时间尽可能少。

该函数允许内核开发人员调试快速路径段，对于printk不合适的情况。通过在代码中散布各种类似于printk的跟踪，开发人员可以快速地看到出现问题的地方。

这只是开发人员的调试工具。请勿在您的代码中散布trace\_puts。（使用trace\_puts()时会使用专用缓冲区分配额外的内存。）

**返回**

如果没有写入任何内容，则返回0，如果返回字符串，则返回正的#（使用\_\_trace\_bputs时为1，使用\_\_trace\_puts时为strlen(str)）。

### min(x，y)

返回两个相同或兼容类型值的最小值

**参数**

x

第一个值

y

第二个值

### max(x，y)

返回两个相同或兼容类型值的最大值

**参数**

x

第一个值

y

第二个值

### min3(x，y，z)

返回三个值中的最小值

**参数**

x

第一个值

y

第二个值

z

第三个值

### max3(x，y，z)

返回三个值中的最大值

**参数**

x

第一个值

y

第二个值

z

第三个值

### min\_not\_zero(x，y)

返回不为零的最小值，除非两者都为零

**参数**

x

值1

y

值2

### clamp(val，lo，hi)

返回在给定范围内夹住的值，并进行严格类型检查

**参数**

val

当前值

lo

最小允许值

hi

最大允许值

**说明**

此宏对lo/hi进行严格的类型检查，以确保它们与val具有相同的类型。查看不必要的指针比较。

### min\_t(type，x，y)

使用指定类型返回两个值的最小值

**参数**

type

要使用的数据类型

x

第一个值

y

第二个值

### max\_t(type，x，y)

使用指定类型返回两个值的最大值

**参数**

type

要使用的数据类型

x

第一个值

y

第二个值

### clamp\_t(type，val，lo，hi)

使用给定类型将值夹在给定范围内

**参数**

type

要使用的变量类型

val

当前值

lo

最小允许的值

hi

最大允许值

**说明**

该宏不进行类型检查，并使用类型为type的临时变量进行所有比较。

### clamp\_val（val，lo，hi）

使用val的类型返回限制在给定范围内的值

**参数**

val

当前值

lo

最小允许值

hi

最大允许值

**说明**

该宏不进行类型检查，并使用输入参数val的任何类型的临时变量。当val为无符号类型而lo和hi是将分配有符号整数类型的文字时，这会很有用。

### swap（a，b）

交换a和b的值

**参数**

a

第一个值

b

第二个值

### container\_of（ptr，type，member）

将结构的成员强制转换为包含结构

**参数**

ptr

成员的指针。

type

包含此嵌入式结构的容器结构的类型。

member

结构中的成员名称。

### container\_of\_safe（ptr，type，member）

将结构的成员强制转换为包含结构

**参数**

ptr

成员的指针。

type

包含此嵌入式结构的容器结构的类型。

member

结构中的成员名称。

**说明**

如果IS\_ERR\_OR\_NULL（ptr），则返回未更改的ptr。

### \_\_visible int printk（const char \* fmt，...）

打印内核消息

**参数**

const char \* fmt

格式字符串

...

可变参数

**说明**

这是printk（）。它可以从任何上下文调用。我们希望它起作用。

我们尝试抓取console\_lock。如果成功，很容易-我们记录输出并调用控制台驱动程序。如果我们未能获取信号量，我们将输出放入日志缓冲区并返回。当前控制台\_sem的持有者将在console\_unlock（）中注意到新的输出;并在释放锁之前将其发送到控制台。

这种延迟的打印的一个影响是，调用printk（）然后更改console\_loglevel的代码可能会中断。这是因为在实际打印发生时检查console\_loglevel。

另请参见printf（3）

请参阅vsnprintf（）文档以获取C99格式字符串扩展。

### void console\_lock（void）

锁定控制台系统以进行独占使用。

**参数**

void

没有参数

**说明**

获取保证调用者具有对控制台系统和控制台驱动程序列表的独占访问权限的锁。

可以睡觉，什么都不返回

### int console\_trylock（void）

尝试锁定控制台系统以进行独占使用。

**参数**

void

没有参数

**说明**

尝试获取保证调用者具有对控制台系统和控制台驱动程序列表的独占访问权限的锁。

在成功时返回1，在无法获取锁时返回0。

### void console\_unlock（void）

解锁控制台系统

**参数**

void

没有参数

**说明**

释放调用方在控制台系统和控制台驱动程序列表上保持的console\_lock。如果此时保留了console\_lock，则可能缓冲了printk（）的控制台输出。如果是这种情况，“console\_unlock（）;”将在释放锁之前发射输出。如果有等待输出，我们将唤醒/dev/kmsg和syslog（）用户。

可以从任何上下文调用console\_unlock（）。

### void console\_conditional\_schedule（void）

如有必要，请让出CPU

**参数**

void

没有参数

**说明**

如果控制台代码当前允许睡眠，并且该CPU应将CPU交给另一个任务，则在此处执行。必须在console\_lock（）内调用;

### bool printk\_timed\_ratelimit(unsigned long \* caller\_jiffies，unsigned int interval\_msecs)

调用方控制的printk速率限制

**参数**

unsigned long \* caller\_jiffies

指向caller状态的指针

unsigned int interval\_msecs

打印之间的最小间隔

**说明**

printk\_timed\_ratelimit（）返回true，如果从上次返回true的时间以来已经过去了interval\_msecs毫秒。

### int kmsg\_dump\_register（struct kmsg\_dumper \* dumper）

注册内核日志转储程序。

**参数**

struct kmsg\_dumper \* dumper

指向kmsg\_dumper结构的指针

**说明**

向系统添加内核日志转储程序。必须设置结构中的转储回调，并返回成功的零，并在否则返回-EINVAL或-EBUSY。

### int kmsg\_dump\_unregister（struct kmsg\_dumper \* dumper）

注销kmsg转储器。

**参数**

struct kmsg\_dumper \* dumper

指向kmsg\_dumper结构的指针

**说明**

从系统中删除转储设备。成功时返回零，否则返回-EINVAL。

### bool kmsg\_dump\_get\_line（struct kmsg\_dump\_iter \* iter，bool syslog，char \* line，size\_t size，size\_t \* len）

检索一个kmsg日志行

**参数**

struct kmsg\_dump\_iter \* iter

kmsg转储迭代器

bool syslog

包括“<4>”前缀

char \* line

将行复制到的缓冲区

size\_t size

缓冲区的最大大小

size\_t \* len

放入缓冲区的行的长度

**说明**

将记录开始到提供的缓冲区中。连续调用将返回向年轻的消息移动到缓冲区末尾的下一个可用记录。

FALat表明kmsg缓冲区的开头，其中包含最旧的kmsg记录，并复制一个rSE表示没有更多记录可读。

### bool kmsg\_dump\_get\_buffer(struct kmsg\_dump\_iter \*iter, bool syslog, char \*buf, size\_t size, size\_t \*len\_out)

复制 kmsg 日志行

**参数**

struct kmsg\_dump\_iter \*iter

kmsg 转储迭代器

bool syslog

包括“<4>”前缀

char \*buf

将行复制到的缓冲区

size\_t size

缓冲区的最大长度

size\_t \*len\_out

放入缓冲区的行的长度

**说明**

从 kmsg 缓冲区的末尾开始，并将尽可能多的最新的 kmsg 记录填充到提供的缓冲区中。如果缓冲区足够大，则可以使用单个调用复制所有可用的 kmsg 记录。

连续调用将使用下一个可用的较旧记录块填充缓冲区，不包括较早检索到的记录。

返回为 FALSE 表示没有更多的记录可读取。

### void kmsg\_dump\_rewind(struct kmsg\_dump\_iter \*iter)

重置迭代器

**参数**

struct kmsg\_dump\_iter \*iter

kmsg 转储迭代器

**说明**

重置转储器的迭代器，以便可以在相同的转储程序中多次调用 kmsg\_dump\_get\_line() 和 kmsg\_dump\_get\_buffer()。

### void panic(const char \*fmt, ...)

停止系统

**参数**

const char \*fmt

要打印的文本字符串

...

可变参数

**说明**

显示消息，然后执行清理。

此函数永远不会返回。

### void add\_taint(unsigned flag, enum lockdep\_ok lockdep\_ok)

如果尚未设置，则添加脏标志。

**参数**

unsigned flag

TAINT\_\* 常量之一。

enum lockdep\_ok lockdep\_ok

锁调试是否仍然正常。

**说明**

如果出现严重问题，则需要 lockdebug\_ok = false，但对于一些值得注意但不会损坏的情况，可以将其设置为 true。

### bool notrace rcu\_is\_watching(void)

查看 RCU 是否认为当前 CPU 处于空闲状态

**参数**

void

没有参数

**说明**

如果 RCU 正在监视运行 CPU，则返回 true，这意味着此 CPU 可以安全地进入 RCU 读侧临界区。换句话说，如果当前 CPU 在其空闲循环中，并且既不在中断处理程序也不在 NMI 处理程序中，则返回 true。

### void call\_rcu\_sched(struct rcu\_head \* head, rcu\_callback\_t func)

在 sched 优先期之后排队调用 RCU。

**参数**

struct rcu\_head \* head

用于排队 RCU 更新的结构。

rcu\_callback\_t func

经过宽限期后要调用的实际回调函数

**说明**

回调函数将在完整的宽限期之后的某个时间被调用，换句话说，在当前执行的所有 RCU 读侧临界区都完成之后。 call\_rcu\_sched() 假定读侧临界区在启用前置换或自愿前置换时结束。 RCU 读侧临界区由以下内容进行限定

rcu\_read\_lock\_sched() 和 rcu\_read\_unlock\_sched()，或

禁用前置换的任何内容。

这些可能是嵌套的。

有关内存顺序保证的更详细信息，请参见 call\_rcu() 的说明。

### void call\_rcu\_bh(struct rcu\_head \* head, rcu\_callback\_t func)

在较快的宽限期之后排队调用 RCU。

**参数**

struct rcu\_head \* head

用于排队 RCU 更新的结构。

rcu\_callback\_t func

经过宽限期后要调用的实际回调函数

**说明**

回调函数将在完整的宽限期之后的某个时间被调用，换句话说，在当前执行的所有 RCU 读侧临界区都完成之后。 call\_rcu\_bh() 假定读侧临界区在完成软中断处理程序时结束。这意味着进程上下文中的读侧临界区不能被软中断打断。此接口在大多数读侧临界区在软中断上下文中的情况下使用。 RCU 读侧临界区由以下内容进行限定

如果在中断上下文中，则为 rcu\_read\_lock() 和 rcu\_read\_unlock()，或

如果在进程上下文中，则为 rcu\_read\_lock\_bh() 和 rcu\_read\_unlock\_bh()。

这些可能是嵌套的。

有关内存顺序保证的更详细信息，请参见 call\_rcu() 的说明。

### void synchronize\_sched(void)

等待 rcu-sched 宽限期已过。

**参数**

void

没有参数

**说明**

控制将在完整的 rcu-sched 宽限期之后的某个时间返回给调用者，换句话说，在当前执行的所有 rcu-sched 读侧临界区都完成之后。 这些读侧临界区由 rcu\_read\_lock\_sched() 和 rcu\_read\_unlock\_sched() 进行限定，可能是嵌套的。请注意，可以使用 preempt\_disable()、local\_irq\_disable() 等代替 rcu\_read\_lock\_sched()。

这意味着所有 preempt\_disable 代码序列（包括 NMI 和非线程化硬件中断处理程序）在进入时正在进行中的这个保证都将完成，然后这个基元将返回。但是，这并不保证软中断处理程序将完成，因为在某些内核中，这些处理程序可以在进程上下文中运行并且可以阻止。

请注意，此保证意味着进一步的内存排序保证。在具有多个 CPU 的系统上，当 synchronize\_sched() 返回时，保证每个 CPU 自上次 RCU-sched 读侧临界区结束以来已执行完整的内存屏障。此外，具有扩展自 synchronize\_sched() 返回以外的 RCU 读侧临界区的每个 CPU 都保证在 synchronize\_sched() 的开始和该 RCU 读侧临界区的开始之间执行完整的内存屏障。请注意，这些保证包括处于离线状态、空闲状态或用户模式下执行的 CPU，以及在内核中执行的 CPU。

此外，如果 CPU A 调用了 synchronize\_sched() 函数，并在 CPU B 上返回了它的调用，则在 synchronize\_sched() 函数执行期间 CPU A 和 CPU B 都保证已执行完整的内存屏障 - 即使 CPU A 和 CPU B 是同一 CPU（但仅在系统具有多个 CPU 的情况下）。

## void synchronize\_rcu\_bh(void)

等待 rcu\_bh 优雅时期结束。

**参数**

void

无参数

**说明**

在完整的 rcu\_bh 优雅时期经过后，控制权将返回给调用者。换句话说，所有当前正在执行的 rcu\_bh 读侧临界区块都已完成。 RCU 读侧关键部分由 rcu\_read\_lock\_bh() 和 rcu\_read\_unlock\_bh() 限定，并且可以嵌套。

有关内存排序保证的更多详细信息，请参见 synchronize\_sched() 的说明。

### unsigned long get\_state\_synchronize\_rcu(void)

快照当前 RCU 状态

**参数**

void

无参数

**说明**

返回一个 cookie，该 cookie 可用于稍后调用 cond\_synchronize\_rcu() 来确定是否在此期间经过了完整的优雅时期。

### void cond\_synchronize\_rcu(unsigned long oldstate)

有条件地等待 RCU 优雅时期

**参数**

unsigned long oldstate

早期调用 get\_state\_synchronize\_rcu() 的返回

**说明**

如果自早期调用 get\_state\_synchronize\_rcu() 后已经经过了完整的 RCU 优雅时期，则只需返回即可。否则，请调用 synchronize\_rcu() 等待完整的优雅时期。

是的，此函数不考虑计数器包装。但是计数器包装是无害的。如果计数器包装，我们等待了超过20亿个优雅周期（在64位系统上还要多得多！），因此等待一个额外的优雅周期应该没问题。

### unsigned long get\_state\_synchronize\_sched(void)

快照当前 RCU-sched 状态

**参数**

void

无参数

**说明**

返回一个 cookie，该 cookie 可用于稍后调用 cond\_synchronize\_sched() 来确定是否在此期间经过了完整的优雅时期。

### void cond\_synchronize\_sched(unsigned long oldstate)

有条件地等待 RCU-sched 优雅时期

**参数**

unsigned long oldstate

早期调用 get\_state\_synchronize\_sched() 的返回

**说明**

如果自早期调用 get\_state\_synchronize\_sched() 后已经经过了完整的 RCU-sched 优雅时期，则只需返回即可。否则，请调用 synchronize\_sched() 等待完整的优雅时期。

是的，此函数不考虑计数器包装。但是计数器包装是无害的。如果计数器包装，我们等待了超过20亿个优雅周期（在64位系统上还要多得多！），因此等待一个额外的优雅周期应该没问题。

### void rcu\_barrier\_bh(void)

等待所有正在飞行的 call\_rcu\_bh() 回调完成。

**参数**

void

无参数

### void rcu\_barrier\_sched(void)

等待正在进行的 call\_rcu\_sched() 回调。

**参数**

void

无参数

### void call\_rcu(struct rcu\_head \*head, rcu\_callback\_t func)

将 RCU 回调队列排队，以在优雅时期后调用回调函数。

**参数**

struct rcu\_head \*head

用于排队 RCU 更新的struct 。

rcu\_callback\_t func

在经过优雅时期后实际调用的回调函数

**说明**

回调函数将在完整的优雅期间过后的某个时间调用，换句话说，在所有现有的 RCU 读侧临界区块完成后调用。但是，回调函数可能会与在调用 call\_rcu() 之后开始的 RCU 读侧临界区块并发执行。 RCU 读侧关键部分由 rcu\_read\_lock() 和 rcu\_read\_unlock() 限定，并且可以嵌套。

请注意，所有 CPU 必须同意优雅期间是否已经超过了所有现有的 RCU 读侧临界区块。在具有多个 CPU 的系统上，这意味着在调用“func()”时，每个 CPU 都保证在其上一个 RCU 读侧临界区块的开始之前，自其上一个 RCU 读侧临界区块结束以来已执行完整的内存屏障。这也意味着，在开始“func()”之后继续执行 RCU 读侧临界区块的每个 CPU 必须在 call\_rcu() 之后在其 RCU 读侧临界区块开始之前执行内存屏障。请注意，这些保证包括处于离线、空闲或用户模式下执行以及在内核中执行的 CPU。

此外，如果 CPU A 调用了 call\_rcu()，CPU B 调用了相应的 RCU 回调函数“func()”，则在调用 call\_rcu() 和调用“func()”之间的时间间隔内，CPU A 和 CPU B 都保证执行完整的内存屏障 - 即使 CPU A 和 CPU B 是同一 CPU（但仅在系统具有多个 CPU 的情况下）。

### void synchronize\_rcu(void)

等待一个优雅期间经过。

**参数**

void

无参数

**说明**

在完整的优雅时期经过后，控制权将返回调用者。换句话说，在从 synchronize\_rcu() 返回后，调用者可能会与使用新的 RCU 读侧临界区块并发执行。 RCU 读侧临界区块由 rcu\_read\_lock() 和 rcu\_read\_unlock() 限定，并且可以嵌套。

请参阅 synchronize\_sched() 的说明，以获取有关内存顺序保证的详细信息。但请注意，仅适用于内存顺序保证。例如，synchronize\_rcu() 不保证等待任何由 preempt\_disable() 保护的代码，而仅保证等待 RCU 读侧临界区，也就是由 rcu\_read\_lock() 保护的代码区段。

### void rcu\_barrier(void)

等待所有正在运行的 call\_rcu() 回调函数完成。

**参数**

void

无参数

**说明**

请注意，此原语未必会等待 RCU 优化周期完成。例如，如果系统中没有任何 RCU 回调在队列中等待，则 rcu\_barrier() 有权立即返回，无需等待任何东西，更不用说等待 RCU 优化周期。

### int rcu\_read\_lock\_sched\_held(void)

可能我们正在 RCU-sched 读侧临界区中？

**参数**

void

无参数

**说明**

如果选择了 CONFIG\_DEBUG\_LOCK\_ALLOC，则在 RCU-sched 读侧临界区返回非零值。在没有 CONFIG\_DEBUG\_LOCK\_ALLOC 的情况下，假设我们正在 RCU-sched 读侧临界区中，除非可以证明我们不在其中。注意，禁用抢占（包括禁用 IRQ）计为 RCU-sched 读侧临界区。这对于要求在 RCU-sched 读侧临界区内调用它们的函数的调试检查非常有用。

检查 debug\_lockdep\_rcu\_enabled()，以防止启动期间和在禁用 lockdep 时出现误报。

请注意，如果 CPU 从 RCU 的角度处于空闲循环中（即我们在 rcu\_idle\_enter() 和 rcu\_idle\_exit() 之间的部分中），则即使 CPU 做了 rcu\_read\_lock()，rcu\_read\_lock\_held() 返回 false 也是如此。原因是 RCU 忽略处于这种部分中的 CPU，将其视为处于扩展静止状态，因此无论其调用了什么 RCU 原语，这样的 CPU 实际上永远不处于 RCU 读侧临界区。必须是这种情况——我们需要在空闲时保留一个无 RCU 窗口，以便 CPU 可能进入低功耗模式。这样我们就可以向启动优化周期的其他 CPU 感知到扩展静止状态。否则，我们会延迟所有的优化周期，因为我们在空闲任务中运行。

类似地，如果当前 CPU 离线，我们会避免声称持有 SRCU 读锁。

### void rcu\_expedite\_gp(void)

加速未来的 RCU 优化周期

**参数**

void

无参数

**说明**

调用此函数后，未来调用 synchronize\_rcu() 等函数将像调用相应的 synchronize\_rcu\_expedited() 函数一样。

### void rcu\_unexpedite\_gp(void)

取消以前对 rcu\_expedite\_gp() 的调用

**参数**

void

无参数

**说明**

撤消之前对 rcu\_expedite\_gp() 的调用。如果所有对 rcu\_expedite\_gp() 的先前调用都由后续对 rcu\_unexpedite\_gp() 的调用取消，而且 rcu\_expedited sysfs/boot 参数未设置，则所有后续对 synchronize\_rcu() 等函数的调用都将返回其正常的非加速行为。

### int rcu\_read\_lock\_held(void)

可能我们正在 RCU 读侧临界区中？

**参数**

void

无参数

**说明**

如果选择了 CONFIG\_DEBUG\_LOCK\_ALLOC，则在 RCU 读侧临界区返回非零值。在没有 CONFIG\_DEBUG\_LOCK\_ALLOC 的情况下，假设我们正在 RCU 读侧临界区中，除非可以证明我们不在其中。这对于要求在 RCU 读侧临界区内调用它们的函数的调试检查非常有用。

检查 debug\_lockdep\_rcu\_enabled()，以防止启动期间和在禁用 lockdep 时出现误报。

请注意，必须在相同的上下文中调用 rcu\_read\_lock() 和相应的 rcu\_read\_unlock()，例如，如果匹配的 rcu\_read\_lock() 是从 irq 处理程序内部调用的，则在进程上下文中调用 rcu\_read\_unlock() 是非法的。

请注意，如果 CPU 从 RCU 的角度处于空闲或离线状态，则不允许使用 rcu\_read\_lock()，因此也要检查这些状态。

### int rcu\_read\_lock\_bh\_held(void)

可能我们正在 RCU-bh 读侧临界区中？

**参数**

void

无参数

**说明**

检查是否禁用了底半部，这涵盖了 CONFIG\_PROVE\_RCU 和非这种情况的情况。请注意，如果有人使用了 rcu\_read\_lock\_bh()，但随后启用了 BH，则锁定依赖项（如果启用）将显示该情况。这对于要求在 RCU 读侧临界区内调用它们的函数的调试检查非常有用。

检查 debug\_lockdep\_rcu\_enabled()，以防止启动期间出现误报。

请注意，必须在相同的上下文中调用 rcu\_read\_lock() 和相应的 rcu\_read\_unlock()，例如，如果匹配的 rcu\_read\_lock() 是从 irq 处理程序内部调用的，则在进程上下文中调用 rcu\_read\_unlock() 是非法的。

### void wakeme\_after\_rcu(struct rcu\_head \* head)

回调函数，在收到优化周期信号后唤醒任务

**参数**

struct rcu\_head \* head

在 rcu\_synchronize 结构内的 rcu\_head 成员指针

**说明**

在优化周期过去后立即唤醒相应的任务。

### void init\_rcu\_head\_on\_stack(struct rcu\_head \* head)

为 debugobjects 初始化栈上的 rcu\_head

**参数**

struct rcu\_head \* head

指向要初始化的rcu\_head结构的指针

**说明**

该函数通知debugobjects，已在堆栈上分配了一个新的rcu\_head结构作为自动变量。这个函数不适用于静态定义或在堆上动态分配的rcu\_head结构。对于!CONFIG\_DEBUG\_OBJECTS\_RCU\_HEAD内核构建，此函数没有任何影响。

### void destroy\_rcu\_head\_on\_stack(struct rcu\_head \* head)

为debugobjects销毁在栈上的rcu\_head

**参数**

struct rcu\_head \* head

指向要初始化的rcu\_head结构的指针

**说明**

此函数通知debugobjects，一个在栈上的rcu\_head结构即将超出范围。与init\_rcu\_head\_on\_stack()一样，对于静态定义或在堆上动态分配的rcu\_head结构，此函数不是必需的。而且，与init\_rcu\_head\_on\_stack()一样，此函数对于!CONFIG\_DEBUG\_OBJECTS\_RCU\_HEAD内核构建没有影响。

### void call\_rcu\_tasks(struct rcu\_head \* rhp, rcu\_callback\_t func)

将RCU加入任务基础优雅周期的调用队列

**参数**

struct rcu\_head \* rhp

用于排队RCU更新的结构。

rcu\_callback\_t func

在优雅周期之后调用的实际回调函数

**说明**

回调函数将在完整的优雅周期经过一段时间后调用，换句话说，在所有当前执行的RCU读侧关键段完成后。call\_rcu\_tasks()假定读侧关键段在自愿的上下文切换（而不是抢占！ ），cond\_resched\_rcu\_qs()，进入idle或转换为usermode执行时结束。因此，不存在类似于rcu\_read\_lock()和rcu\_read\_unlock()的读侧原语，因为此原语旨在确定所有任务都已通过安全状态，而不是进行数据结构同步。

有关内存排序保证的更详细信息，请参阅call\_rcu()的说明。

### void synchronize\_rcu\_tasks(void)

等待rcu-tasks优雅周期已经过。

**参数**

void

没有参数

**说明**

在完整的rcu-tasks优雅周期经过一段时间后，控制将返回调用者，换句话说，在所有当前执行的rcu-tasks读侧临界区域已经过时。这些读侧关键段由schedule()，cond\_resched\_tasks\_rcu\_qs()，空闲执行，用户空间执行，调用synchronize\_rcu\_tasks()和（理论上）cond\_resched()限定。

这是一个非常专业的基元，仅供追踪和其他需要操纵函数序言和分析挂钩的情况使用。synchronize\_rcu\_tasks()函数（尚未）不适用于多个CPU的重度使用。

请注意，此保证还意味着进一步的内存排序保证。在具有多个CPU的系统上，当synchronize\_rcu\_tasks()返回时，保证每个CPU都已执行自上次RCU-tasks读侧关键区域结束以来的完整内存屏障。此外，有一个rcu-tasks读侧关键部分的每个CPU超出返回synchronize\_rcu\_tasks()，在开始synchronize\_rcu\_tasks()之后，在该RCU-tasks读侧关键部分开始之前执行了完整的内存屏障。请注意，这些保证包括脱机，空闲或在用户模式下执行以及在内核中执行的CPU。

此外，如果CPU A调用了synchronize\_rcu\_tasks()，返回到CPU B的调用者，则在synchronize\_rcu\_tasks()的执行期间，CPU A和CPU B都保证已经执行了完整的内存屏障-即使CPU A和CPU B是相同的CPU（但仅当系统具有多个CPU时）。

### void rcu\_barrier\_tasks(void)

等待正在进行的call\_rcu\_tasks()回调。

**参数**

void

没有参数

**说明**

虽然当前的实现保证等待，但例如如果没有挂起的回调，则不必如此。

## 设备资源管理

### void \*\_\_devres\_alloc\_node(dr\_release\_t release, size\_t size, gfp\_t gfp, int nid, const char \*name)

分配设备资源数据

**参数**

dr\_release\_t release

与devres将相关联的释放函数

size\_t size

分配大小

gfp\_t gfp

分配标志

int nid

NUMA节点

const char \*name

资源名称

**说明**

分配大小为字节的devres。分配的区域为零，然后与释放相关联。返回的指针可以传递给其他devres\_\*()函数。

**返回**

成功时返回指向分配的devres的指针，失败时返回NULL。

### void devres\_for\_each\_res(struct device \*dev, dr\_release\_t release, dr\_match\_t match, void \*match\_data, void (\*fn)(struct device\*, void\*, void\*), void \*data)

资源迭代器

**参数**

struct device \*dev

从中迭代资源的设备

dr\_release\_t release

查找与该释放函数相关联的资源

dr\_match\_t match

匹配函数（可选）

void \*match\_data

匹配函数的数据

void (\*fn)(struct device \*, void \*, void \*)

对每个匹配的资源调用的函数。

void \*data

fn的数据，fn的第3个参数

**说明**

对于与release相关联的每个devres，调用fn，对于匹配函数返回1的devres。

**返回**

无

### void devres\_free（void \*res）

释放设备资源数据

**参数**

void \*res

要释放的devres数据指针

**说明**

使用devres\_alloc（）创建的devres将被释放。

### void devres\_add（struct device \*dev，void \*res）

注册设备资源

**参数**

struct device \*dev

要添加资源的设备

void \*res

要注册的资源

**说明**

将devres res注册到dev上。 res应该使用devres\_alloc（）进行分配。在驱动程序卸载时，将调用相关的释放函数，devres将自动释放。

### void \*devres\_find（struct device \*dev，dr\_release\_t release，dr\_match\_t match，void \*match\_data）

查找设备资源

**参数**

struct device \*dev

要查询资源的设备

dr\_release\_t release

查找与此释放函数相关联的资源

dr\_match\_t match

匹配函数（可选）

void \*match\_data

匹配函数的数据

**说明**

找到最新的与释放相关联且匹配返回1的设备的devres。如果match为NULL，则认为匹配所有内容。

**返回**

找到的devres指针，如果未找到则为NULL。

### void \*devres\_get（struct device \*dev，void \*new\_res，dr\_match\_t match，void \*match\_data）

查找devres，如果不存在，则原子地添加一个

**参数**

struct device \*dev

要查找或添加devres的设备

void \*new\_res

指向新初始化的devres的指针，以添加如果未找到

dr\_match\_t match

匹配函数（可选）

void \*match\_data

匹配函数的数据

**说明**

找到与new\_res具有相同释放函数且匹配返回1的设备的最新devres。如果找到，new\_res将被释放；否则，new\_res将被原子地添加。

**返回**

找到或添加的devres指针。

### void \*devres\_remove（struct device \*dev，dr\_release\_t release，dr\_match\_t match，void \*match\_data）

查找设备资源并删除它

**参数**

struct device \*dev

要找到资源的设备

dr\_release\_t release

查找与此释放函数相关联的资源

dr\_match\_t match

匹配函数（可选）

void \*match\_data

匹配函数的数据

**说明**

找到最新的与释放相关联且匹配返回1的设备的devres。如果match为NULL，则认为匹配所有内容。如果找到，将原子地删除资源并返回。

**返回**

成功删除的devres指针，如果未找到则为NULL。

### int devres\_destroy（struct device \*dev，dr\_release\_t release，dr\_match\_t match，void \*match\_data）

查找设备资源并销毁它

**参数**

struct device \*dev

要找到资源的设备

dr\_release\_t release

查找与此释放函数相关联的资源

dr\_match\_t match

匹配函数（可选）

void \*match\_data

匹配函数的数据

**说明**

找到最新的与释放相关联且匹配返回1的设备的devres。如果match为NULL，则认为匹配所有。如果找到，将原子地删除资源并释放。

请注意，资源的释放函数不会被调用，只有devres分配的数据将被释放。调用者需要负责释放其他数据。

**返回**

如果找到并释放devres，则为0；如果未找到，则为-ENOENT。

### int devres\_release（struct device \*dev，dr\_release\_t release，dr\_match\_t match，void \*match\_data）

查找设备资源并销毁它，调用释放函数

**参数**

struct device \*dev

要找到资源的设备

dr\_release\_t release

查找与此释放函数相关联的资源

dr\_match\_t match

匹配函数（可选）

void \*match\_data

匹配函数的数据

**说明**

找到最新的与释放相关联且匹配返回1的设备的devres。如果match为NULL，则认为匹配所有内容。如果找到，将原子地删除资源，调用释放函数并释放资源。

**返回**

如果找到并释放devres，则为0；如果未找到，则为-ENOENT。

### void \*devres\_open\_group（struct device \*dev，void \*id，gfp\_t gfp）

打开新的devres组

**参数**

struct device \*dev

要为其打开devres组的设备

void \*id

分隔符ID

gfp\_t gfp

分配标志

**说明**

为dev以id打开一个新的devres组。对于id，建议使用将不会用于另一个组的对象的指针。如果id为NULL，则创建一个按地址唯一的ID。

**返回**

新组的ID，失败为NULL。

### void devres\_close\_group（struct device \*dev，void \*id）

关闭devres组

**参数**

struct device \*dev

要关闭devres组的设备

void \*id

目标组的ID，可以为NULL

**说明**

关闭由id标识的组。如果id为NULL，则选择最新打开的组。

### void devres\_remove\_group（struct device \*dev，void \*id）

删除devres组

**参数**

struct device \*dev

要删除组的设备

void \*id

要删除的组的ID，可以为NULL

**说明**

删除由id标识的组。如果id为NULL，则选择最新打开的组。请注意，删除组不会影响任何其他资源。

### int devres\_release\_group（struct device \*dev，void \*id）

释放devres组中的资源

**参数**

struct device \*dev

需要释放的设备

void \*id

目标组的ID，可以为NULL

**说明**

释放由id标识的组中的所有资源。如果id为NULL，则选择最近打开的组。选择的组及正确嵌套在所选组中的组均将被移除。

**返回**

释放的非组资源数量。

### int devm\_add\_action(struct device \*dev, void (\*action)(void\*), void \*data)

将自**定义**操作添加到受管理资源的列表中

**参数**

struct device \*dev

拥有该操作的设备

void (\*action)(void \*)

应该调用的函数

void \*data

传递给操作实现的数据指针

**说明**

这将添加一个自定义操作到受管理资源的列表中，以便作为标准资源解除的一部分执行它。

### void devm\_remove\_action(struct device \*dev, void (\*action)(void\*), void \*data)

删除先前添加的自定义操作

**参数**

struct device \*dev

拥有该操作的设备

void (\*action)(void \*)

实现该操作的函数

void \*data

传递给操作实现的数据指针

**说明**

删除由devm\_add\_action()先前添加的操作实例。操作和数据应与现有条目之一匹配。

### void devm\_release\_action(struct device \*dev, void (\*action)(void\*), void \*data)

释放以前添加的自定义操作

**参数**

struct device \*dev

拥有该操作的设备

void (\*action)(void \*)

实现该操作的函数

void \*data

传递给操作实现的数据指针

**说明**

释放并删除由devm\_add\_action()先前添加的操作实例。操作和数据应与现有条目之一匹配。

### void \*devm\_kmalloc(struct device \*dev, size\_t size, gfp\_t gfp)

资源管理kmalloc

**参数**

struct device \*dev

分配内存的设备

size\_t size

分配大小

gfp\_t gfp

分配gfp标志

**说明**

被管理的kmalloc。使用此函数分配的内存在驱动程序分离时将自动释放。像所有其他devres资源一样，保证对齐方式是unsigned long long。

**返回**

成功时分配的内存指针，失败时为NULL。

### char \*devm\_kstrdup(struct device \*dev, const char \*s, gfp\_t gfp)

分配资源受控空间并将现有字符串复制到其中。

**参数**

struct device \*dev

分配内存的设备

const char \*s

要复制的字符串

gfp\_t gfp

在调用devm\_kmalloc()时用于分配内存的GFP掩码

**返回**

成功时分配的字符串指针，失败时为NULL。

### char \*devm\_kvasprintf(struct device \*dev, gfp\_t gfp, const char \*fmt, va\_list ap)

分配资源管理空间并将一个字符串格式化到其中。

**参数**

struct device \*dev

分配内存的设备

gfp\_t gfp

在调用devm\_kmalloc()时用于分配内存的GFP掩码

const char \*fmt

printf()-样式格式字符串

va\_list ap

格式字符串的参数

**返回**

成功时分配的字符串指针，失败时为NULL。

### char \*devm\_kasprintf(struct device \*dev, gfp\_t gfp, const char \*fmt, ...)

分配资源管理空间并将一个字符串格式化到其中。

**参数**

struct device \*dev

分配内存的设备

gfp\_t gfp

在调用devm\_kmalloc()时用于分配内存的GFP掩码

const char \*fmt

printf()-样式格式字符串

...

格式字符串的参数

**返回**

成功时分配的字符串指针，失败时为NULL。

### void devm\_kfree(struct device \*dev, const void \*p)

资源受控kfree

**参数**

struct device \*dev

此内存所属的设备

const void \*p

要释放的内存

**说明**

释放使用devm\_kmalloc()分配的内存。

### void \*devm\_kmemdup(struct device \*dev, const void \*src, size\_t len, gfp\_t gfp)

资源管理kmemdup

**参数**

struct device \*dev

此内存所属的设备

const void \*src

要复制的内存区域

size\_t len

内存区域长度

gfp\_t gfp

要使用的GFP标志

**说明**

使用资源管理kmalloc复制一段内存区域

### unsigned long devm\_get\_free\_pages(struct device \*dev, gfp\_t gfp\_mask, unsigned int order)

资源管理\_\_get\_free\_pages

**参数**

struct device \*dev

要分配内存的设备

gfp\_t gfp\_mask

分配gfp标志

unsigned int order

分配大小为(1 << order)页

**说明**

管理的get\_free\_pages。使用此函数分配的内存在驱动程序分离时将自动释放。

**返回**

成功时分配的内存地址，失败时为0。

### void devm\_free\_pages(struct device \*dev, unsigned long addr)

资源受控的free\_pages

**参数**

struct device \*dev

此内存所属的设备

unsigned long addr

要释放的内存

**说明**

释放使用devm\_get\_free\_pages()分配的内存。与free\_pages不同的是，无需提供顺序。

### void \_\_percpu \*\_\_devm\_alloc\_percpu(struct device \*dev, size\_t size, size\_t align)

资源管理的alloc\_percpu

**参数**

struct device \*dev

要为其分配per-CPU内存的设备

size\_t size

要分配的per-CPU内存的大小

size\_t align

要分配的per-CPU内存的对齐

**说明**

管理的alloc\_percpu。使用此函数分配的per-CPU内存在驱动程序分离时将自动释放。

**返回**

成功时分配的内存指针，失败时为NULL。

### void devm\_free\_percpu(struct device \*dev, void \_\_percpu \*pdata)

资源管理的free\_percpu

**参数**

struct device \*dev

此内存所属设备

void \_\_percpu \*pdata

要释放的单核内存

**说明**

释放用devm\_alloc\_percpu()分配的内存。

# 设备驱动的基础架构

## 基本设备驱动模型结构

### struct bus\_type

设备的总线类型

**定义**

struct bus\_type {

const char \*name;

const char \*dev\_name;

struct device \*dev\_root;

const struct attribute\_group \*\*bus\_groups;

const struct attribute\_group \*\*dev\_groups;

const struct attribute\_group \*\*drv\_groups;

int (\*match)(struct device \*dev, struct device\_driver \*drv);

int (\*uevent)(struct device \*dev, struct kobj\_uevent\_env \*env);

int (\*probe)(struct device \*dev);

int (\*remove)(struct device \*dev);

void (\*shutdown)(struct device \*dev);

int (\*online)(struct device \*dev);

int (\*offline)(struct device \*dev);

int (\*suspend)(struct device \*dev, pm\_message\_t state);

int (\*resume)(struct device \*dev);

int (\*num\_vf)(struct device \*dev);

int (\*dma\_configure)(struct device \*dev);

const struct dev\_pm\_ops \*pm;

const struct iommu\_ops \*iommu\_ops;

struct subsys\_private \*p;

struct lock\_class\_key lock\_key;

bool need\_parent\_lock;

};

**成员**

name

总线的名称。

dev\_name

用于子系统enum设备的名称形如(“foo``u``”, dev->id)。

dev\_root

要用作父设备的默认设备。

bus\_groups

总线的默认属性。

dev\_groups

总线上设备的默认属性。

drv\_groups

总线上设备驱动的默认属性。

match

当添加此总线的新设备或驱动程序时，也许会调用多次。如果给定的设备可以由给定的驱动程序处理，则应返回正值，否则返回零。如果不能确定驱动程序支持设备，则也可返回错误代码。如果返回-EPROBE\_DEFER，则将设备排队进行延迟探测。

uevent

在添加设备、删除设备或生成uevents以添加环境变量时调用。

probe

当添加到此总线的新设备或驱动程序时，回调特定驱动程序的probe以初始化匹配的设备。

remove

当从此总线中删除设备时调用。

shutdown

在关闭时间调用以使设备排除干扰。

online

用于将设备重新上线（脱机后）。

offline

将设备脱机以进行热拔插调用。

suspend

当此总线上的设备想要进入睡眠模式时调用。

resume

当此总线上的设备退出睡眠模式时调用。

num\_vf

用于找出此总线上的设备支持多少个虚拟功能。

dma\_configure

用于在此总线上的设备上设置DMA配置。

pm

此总线的电源管理操作，回调特定设备驱动程序的pm-ops。

iommu\_ops

适用于此总线的IOMMU特定操作，用于将IOMMU驱动程序实现连接到总线并允许驱动程序执行总线特定的设置。

p

驱动程序核心的私有数据，只能由驱动程序核心触碰。

lock\_key

用于锁验证器的锁类键。

need\_parent\_lock

在此总线上探测或删除设备时，设备核心应锁定设备的父对象。

**说明**

总线是处理器和一个或多个设备之间的通道。为了设备模型，即使是内部、虚拟的“平台”总线，所有设备也通过总线连接。总线可以插入彼此。例如，USB控制器通常是PCI设备。设备模型表示总线和它们控制的设备之间的实际连接。总线由bus\_type结构表示。它包含名称、默认属性、总线方法、PM操作和驱动程序核心的私有数据。

### enum probe\_type

设备驱动程序探测类型，尝试。设备驱动程序可以选择特殊处理各自的探测程序。这告诉核心要期望和优先使用什么。

**常量**

PROBE\_DEFAULT\_STRATEGY

由能够同步或异步进行探测的驱动程序使用。

PROBE\_PREFER\_ASYNCHRONOUS

针对“慢”设备的驱动程序，探测顺序不是引导系统的关键，可以选择异步执行它们的探测。

PROBE\_FORCE\_SYNCHRONOUS

使用此注释的驱动程序需要它们的探测程序与驱动程序和设备注册同步运行（除了处理-EPROBE\_DEFER处理之外—重探测总是异步完成）。

**说明**

请注意，最终目标是将内核切换为默认使用异步探测，因此将驱动程序注释为PROBE\_PREFER\_ASYNCHRONOUS是一种临时措施，允许我们在验证其余驱动程序时加快引导进程。

### struct device\_driver

基本设备驱动程序结构

**定义**

struct device\_driver {

const char \*name;

struct bus\_type \*bus;

struct module \*owner;

const char \*mod\_name;

bool suppress\_bind\_attrs;

enum probe\_type probe\_type;

const struct of\_device\_id \*of\_match\_table;

const struct acpi\_device\_id \*acpi\_match\_table;

int (\*probe) (struct device \*dev);

int (\*remove) (struct device \*dev);

void (\*shutdown) (struct device \*dev);

int (\*suspend) (struct device \*dev, pm\_message\_t state);

int (\*resume) (struct device \*dev);

const struct attribute\_group \*\*groups;

const struct dev\_pm\_ops \*pm;

void (\*coredump) (struct device \*dev);

struct driver\_private \*p;

};

**成员**

name

设备驱动程序的名称。

bus

该驱动程序的设备所属的总线。

owner

模块所有者。

mod\_name

用于内置模块。

suppress\_bind\_attrs

通过 sysfs 禁用绑定/解除绑定。

probe\_type

要使用的探测类型（同步或异步）。

of\_match\_table

打开的固件表。

acpi\_match\_table

ACPI 匹配表。

probe

调用查询特定设备是否存在，此驱动程序是否可以与其一起工作，并将驱动程序绑定到特定设备。

remove

当设备从系统中移除时调用以解除设备与该驱动程序的绑定。

shutdown

在关闭时调用以停止设备。

suspend

调用以使设备进入睡眠模式。通常为低功耗状态。

resume

调用以使设备退出睡眠模式。

groups

由驱动程序核心自动创建的默认属性。

pm

匹配此驱动程序的设备的电源管理操作。

coredump

写入 sysfs 条目时调用。设备驱动程序应调用 dev\_coredump API，从而产生 uevent。

p

驱动核心的私有数据，除驱动核心外，其他人不能触及。

**说明**

设备驱动程序模型跟踪系统已知的所有驱动程序。这种跟踪的主要原因是使驱动程序核心能够将驱动程序与新设备相匹配。然而，一旦驱动程序成为系统中的已知对象，许多其他事情就成为可能。设备驱动程序可以导出独立于任何特定设备的信息和配置变量。

### struct subsys\_interface

设备功能接口

**定义**

struct subsys\_interface {

const char \*name;

struct bus\_type \*subsys;

struct list\_head node;

int (\*add\_dev)(struct device \*dev, struct subsys\_interface \*sif);

void (\*remove\_dev)(struct device \*dev, struct subsys\_interface \*sif);

};

**成员**

name

设备功能名称

subsys

要附加到的设备的子系统

node

在子系统中注册的函数列表

add\_dev

设备连接到设备功能处理程序

remove\_dev

设备连接到设备功能处理程序

**说明**

附加到子系统的简单接口。多个接口可以连接到一个子系统及其设备。与驱动程序不同，他们不专门声明或控制设备。接口通常表示子系统/设备类的特定功能。

### struct class

设备类别

**定义**

struct class {

const char \*name;

struct module \*owner;

const struct attribute\_group \*\*class\_groups;

const struct attribute\_group \*\*dev\_groups;

struct kobject \*dev\_kobj;

int (\*dev\_uevent)(struct device \*dev, struct kobj\_uevent\_env \*env);

char \*(\*devnode)(struct device \*dev, umode\_t \*mode);

void (\*class\_release)(struct class \*class);

void (\*dev\_release)(struct device \*dev);

int (\*shutdown\_pre)(struct device \*dev);

const struct kobj\_ns\_type\_operations \*ns\_type;

const void \*(\*namespace)(struct device \*dev);

void (\*get\_ownership)(struct device \*dev, kuid\_t \*uid, kgid\_t \*gid);

const struct dev\_pm\_ops \*pm;

struct subsys\_private \*p;

};

**成员**

name

类的名称。

owner

模块所有者。

class\_groups

此类的默认属性。

dev\_groups

属于该类的设备的默认属性。

dev\_kobj

表示此类并将其链接到层次结构中的 kobject。

dev\_uevent

在添加设备、从此类中删除设备或生成 uevents 以添加环境变量的其他一些事情时调用。

devnode

回调以提供 devtmpfs。

class\_release

调用释放这个类。

dev\_release

调用以释放设备。

shutdown\_pre

在驱动程序关闭之前的关闭时间调用。

ns\_type

回调，以便 sysfs 可以确定命名空间。

namespace

设备的命名空间属于此类。

get\_ownership

允许类为属于该类的设备指定 sysfs 目录的 uid/gid。通常绑定到设备的命名空间。

pm

此类的默认设备电源管理操作。

p

驱动核的私有数据，除驱动核外，其他人不能触及。

**说明**

类是设备的高级视图，它抽象出低级实现细节。驱动程序可能会看到 SCSI 磁盘或 ATA 磁盘，但在类级别上，它们都只是磁盘。类允许用户空间根据设备的作用而不是它们的连接方式或工作方式来使用设备。

### devm\_alloc\_percpu

devm\_alloc\_percpu (dev, type)

资源管理的 alloc\_percpu

**参数**

dev

为其分配每个 CPU 内存的设备

type

为每个 CPU 分配内存的类型

**说明**

托管 alloc\_percpu。使用此函数分配的每个 CPU 内存在驱动程序分离时自动释放。

**返回**

成功时指向已分配内存的指针，失败时指向 NULL。

### struct device\_connection

设备连接描述符

**定义**

struct device\_connection {

const char \*endpoint[2];

const char \*id;

struct list\_head list;

};

**成员**

endpoint

连接在一起的两个设备的名称

id

连接的唯一标识符

list

列表头，私有，仅供内部使用

### enum device\_link\_state

设备连接状态。

**常量**

DL\_STATE\_NONE

未跟踪驱动程序的存在。

DL\_STATE\_DORMANT

供应/消费驱动程序都不存在。

DL\_STATE\_AVAILABLE

供应驱动程序存在，但消费者不存在。

DL\_STATE\_CONSUMER\_PROBE

消费者正在探测（存在供应驱动程序）。

DL\_STATE\_ACTIVE

供应和消费驱动程序均存在。

DL\_STATE\_SUPPLIER\_UNBIND

供应驱动程序正在解除绑定。

### struct device\_link

设备连接表示。

**定义**

struct device\_link {

struct device \*supplier;

struct list\_head s\_node;

struct device \*consumer;

struct list\_head c\_node;

enum device\_link\_state status;

u32 flags;

bool rpm\_active;

struct kref kref;

#ifdef CONFIG\_SRCU;

struct rcu\_head rcu\_head;

#endif;

};

**成员**

v成员

supplier

链路供应商端的设备。

s\_node

挂钩到供应商设备的消费者链接列表。

consumer

链路消费者端的设备。

c\_node

挂钩到消费者设备的供应商链接列表。

status

链接的状态（关于驱动程序的存在）。

flags

链接标志。

rpm\_active

消费者设备是否处于运行时 PM 活动状态。

kref

计算重复添加相同的链接。

rcu\_head

用于延迟执行 SRCU 回调的 RCU 头。

### enum dl\_dev\_state

设备驱动程序存在跟踪信息。

**常量**

DL\_DEV\_NO\_DRIVER

设备未附加驱动程序。

DL\_DEV\_PROBING

正在探测驱动程序。

DL\_DEV\_DRIVER\_BOUND

驱动程序已绑定到设备。

DL\_DEV\_UNBINDING

驱动程序正在从设备解除绑定。

### struct dev\_links\_info

与设备链接相关的设备数据。

**定义**

struct dev\_links\_info {

struct list\_head suppliers;

struct list\_head consumers;

enum dl\_dev\_state status;

};

**成员**

suppliers

链接至供应商设备的链表；

consumers

链接至消费者设备的链表。

状态

驱动程序状态信息。

### struct device

基本设备结构

**定义**

struct device {

struct device \*parent;

struct device\_private \*p;

struct kobject kobj;

const char \*init\_name;

const struct device\_type \*type;

struct mutex mutex;

struct bus\_type \*bus;

struct device\_driver \*driver;

void \*platform\_data;

void \*driver\_data;

struct dev\_links\_info links;

struct dev\_pm\_info power;

struct dev\_pm\_domain \*pm\_domain;

#ifdef CONFIG\_GENERIC\_MSI\_IRQ\_DOMAIN;

struct irq\_domain \*msi\_domain;

#endif;

#ifdef CONFIG\_PINCTRL;

struct dev\_pin\_info \*pins;

#endif;

#ifdef CONFIG\_GENERIC\_MSI\_IRQ;

struct list\_head msi\_list;

#endif;

#ifdef CONFIG\_NUMA;

int numa\_node;

#endif;

const struct dma\_map\_ops \*dma\_ops;

u64 \*dma\_mask;

u64 coherent\_dma\_mask;

u64 bus\_dma\_mask;

unsigned long dma\_pfn\_offset;

struct device\_dma\_parameters \*dma\_parms;

struct list\_head dma\_pools;

struct dma\_coherent\_mem \*dma\_mem;

#ifdef CONFIG\_DMA\_CMA;

struct cma \*cma\_area;

#endif;

struct dev\_archdata archdata;

struct device\_node \*of\_node;

struct fwnode\_handle \*fwnode;

dev\_t devt;

u32 id;

spinlock\_t devres\_lock;

struct list\_head devres\_head;

struct klist\_node knode\_class;

struct class \*class;

const struct attribute\_group \*\*groups;

void (\*release)(struct device \*dev);

struct iommu\_group \*iommu\_group;

struct iommu\_fwspec \*iommu\_fwspec;

bool offline\_disabled:1;

bool offline:1;

bool of\_node\_reused:1;

};

**成员**

parent

设备的“父”设备，它所连接到的设备。在大多数情况下，父设备是某种总线或主机控制器。如果父设备为NULL，则该设备是顶级设备，通常不是您想要的。

p

保存设备驱动程序内核部分的私有数据。有关详细信息，请参阅struct device\_private的注释。

kobj

一个顶层，抽象的类，其他类都是从其派生的。

init\_name

设备的初始名称。

type

设备类型。这标识设备类型和携带类型特定信息。

mutex

同步调用其驱动程序的互斥锁。

bus

设备所在的总线类型。

driver

分配此设备的驱动程序。

platform\_data

特定于设备的平台数据。

driver\_data

驱动程序特定信息的私有指针。

links

该设备的供应商和消费者链接。

power

用于设备电源管理。有关详细信息，请参见文档/driver-api/pm/devices.rst。

pm\_domain

提供回调，在系统挂起，休眠，系统恢复，以及在运行时PM转换期间执行，以及子系统级别和驱动程序级别的回调。

msi\_domain

此设备使用的通用MSI域。

pins

用于设备引脚管理。有关详细信息，请参见文档/driver-api/pinctl.rst。

msi\_list

承载MSI说明符

numa\_node

该设备靠近的NUMA节点。

dma\_ops

此设备的DMA映射操作。

dma\_mask

Dma掩码（如果dma’ble设备）。

coherent\_dma\_mask

像dma\_mask一样，但对于alloc\_coherent映射，因为并非所有硬件都支持64位地址进行一致的分配，因此需要更小的掩码。

bus\_dma\_mask

上游桥或总线的掩码，其施加的DMA限制比设备本身支持的更小。

dma\_pfn\_offset

DMA内存范围相对于RAM的偏移量

dma\_parms

低级驱动程序可能会设置这些参数，以便让IOMMU代码了解段限制。

dma\_pools

Dma池（如果dma可用的设备）。

dma\_mem

用于一致内存覆盖的内部。

cma\_area

用于DMA分配的连续内存区域。

archdata

用于特定架构的附加内容。

of\_node

关联设备树节点。

fwnode

由平台固件提供的关联设备节点。

devt

用于创建sysfs“dev”。

id

设备实例。

devres\_lock

用于保护设备资源的自旋锁。

devres\_head

设备的资源列表。

knode\_class

用于将设备添加到类列表的节点。

class

设备的类。

groups

可选属性组。

release

在所有引用消失后释放设备的回调函数。这应该由设备分配器（即发现设备的总线驱动程序）设置。

iommu\_group

设备所属的IOMMU组。

iommu\_fwspec

由固件提供的IOMMU特定属性。

offline\_disabled

如果设置，则设备永久在线。

offline

在成功调用总线类型的的offline()之后设置。

of\_node\_reused

如果设备树节点与祖先设备共享，则设置。

**示例**

对于嵌入式和基于SOC的硬件所典型的定制板上的设备，Linux通常使用platform\_data指向说明设备及其如何连接的板特定结构。这可以包括可用的端口、芯片变量、哪些GPIO引脚扮演什么附加角色等。这缩小了“板支持包”（BSP）并最小化驱动程序中的特定于板的#ifdef。

**说明**

在Linux系统中，每个设备在最低级别上由struct device的实例表示。设备结构包含设备模型核心模拟系统所需的信息。然而，大多数子系统跟踪它们托管的设备的其他信息。因此，设备很少由裸的设备结构表示；相反，像kobject结构一样，该结构通常嵌入在设备的更高级别表示中。

### module\_driver(\_\_driver, \_\_register, \_\_unregister, ...)

用于不在模块init/exit中执行任何特殊操作的驱动程序的辅助宏。这消除了许多样板文件。每个模块仅可以使用此宏一次，并调用它将替换module\_init()和module\_exit()。

**参数**

\_\_driver

驱动程序名称。

\_\_register

此驱动程序类型的注册函数。

\_\_unregister

此驱动程序类型的注销函数。

...

要传递给\_\_register和\_\_unregister的其他参数。

**说明**

使用此宏构建特定于总线的宏，不要单独使用它。

### builtin\_driver(\_\_driver, \_\_register, ...)

用于不在init中执行任何特殊操作且没有退出的驱动程序的辅助宏。这消除了一些样板文件。每个驱动程序仅可以使用此宏一次，并调用它将替换device\_initcall（或在某些情况下是传统的\_\_initcall）。这旨在是上面的module\_driver()的直接平行，但没有不用于内置情况的\_\_exit内容。

**参数**

\_\_driver

驱动程序名称。

\_\_register

此驱动程序类型的注册函数。

...

要传递给\_\_register的其他参数

**说明**

使用此宏构建特定于总线的宏，不要单独使用它。

## 设备驱动库

### void driver\_init(void)

初始化驱动模型。

**参数**

void

没有参数。

**说明**

调用驱动程序模型初始化函数以初始化其子系统。从init / main.c早期调用。

### int driver\_for\_each\_device(struct device\_driver \*drv, struct device \*start, void \*data, int (\*fn)(struct device\*, void\*))

绑定到驱动程序的设备的迭代器。

**参数**

struct device\_driver \*drv

正在迭代的驱动程序。

struct device \*start

要开始的设备。

void \*data

要传递给回调的数据。

int (\*)(struct device \*, void \*) fn

调用每个设备的功能。

**说明**

遍历drv的设备列表，为每个设备调用fn 。

### struct device \*driver\_find\_device(struct device\_driver \* drv, struct device \* start, void \* data, int (\*match) (struct device \*dev, void \*data)

用于定位特定设备的设备迭代器。

**参数**

struct device\_driver \* drv

设备的驱动程序。

struct device \* start

要开始的设备。

void \* data

要传递给match函数的数据。

int (\*)(struct device \*dev, void \*data) match

检查设备的回调函数。

**说明**

这类似于上面的driver\_for\_each\_device()函数，但它返回一个对设备的引用，后续可以“查找”，由match回调确定。

如果回调返回0，则设备不匹配，如果返回非零，则此函数将返回给调用方，而不会迭代更多的设备。

### int driver\_create\_file(struct device\_driver \*drv, const struct driver\_attribute \*attr)

为驱动程序创建sysfs文件。

**参数**

struct device\_driver \*drv

驱动程序。

const struct driver\_attribute \*attr

驱动程序属性说明符。

### void driver\_remove\_file(struct device\_driver \*drv, const struct driver\_attribute \*attr)

删除驱动程序的sysfs文件。

**参数**

struct device\_driver \*drv

驱动程序。

const struct driver\_attribute \*attr

驱动程序属性说明符。

### int driver\_register(struct device\_driver \*drv)

注册驱动程序与总线

**参数**

struct device\_driver \*drv

要注册的驱动程序

**说明**

我们将大部分工作交给bus\_add\_driver() 调用，因为我们大多数要做的事情都涉及总线结构。

### void driver\_unregister(struct device\_driver \*drv)

从系统中移除驱动程序。

**参数**

struct device\_driver \*drv

驱动程序。

**说明**

同样，我们将大部分工作交给总线层调用。

### struct device\_driver \*driver\_find(const char \*name, struct bus\_type \*bus)

按名称在总线上定位驱动程序。

**参数**

const char \*name

驱动程序的名称。

struct bus\_type \*bus

要扫描驱动程序的总线。

**说明**

调用kset\_find\_obj() 来迭代总线上驱动程序列表，以按名称查找驱动程序。如果找到，则返回驱动程序。

此例程不提供锁定，以防止调用程序在使用该驱动程序时注销或卸载该驱动程序。调用方负责防止这种情况发生。

### struct device\_link \*device\_link\_add(struct device \*consumer, struct device \*supplier, u32 flags)

在两个设备之间创建链接。

**参数**

struct device \*consumer

链接的消费方。

struct device \*supplier

链接的供应方。

u32 flags

链接标记。

**说明**

调用者负责链接创建与运行时 PM 的正确同步。首先，设置 DL\_FLAG\_PM\_RUNTIME 标志将导致运行时 PM 框架考虑链接。其次，如果除此之外还设置了 DL\_FLAG\_RPM\_ACTIVE 标志，则供应商设备将被迫进入活动元状态并在创建链接时进行引用计数。如果没有设置 DL\_FLAG\_PM\_RUNTIME，DL\_FLAG\_RPM\_ACTIVE 将被忽略。

如果设置了 DL\_FLAG\_AUTOREMOVE\_CONSUMER，当消费者设备驱动程序与其解除绑定时，该链接将自动删除。DL\_FLAG\_AUTOREMOVE\_CONSUMER 和 DL\_FLAG\_STATELESS 集的组合无效，将导致返回 NULL。

创建链接的副作用是通过将消费者设备和所有依赖于它的设备移动到这些列表的末尾来重新排序 dpm\_list 和 devices\_kset 列表（当此功能启用时，未注册的设备不会发生这种情况）称为）。

调用此函数时需要注册供应商设备，否则将返回 NULL。然而，消费者设备不需要注册

### void device\_link\_del(struct device\_link \*link)

删除两个设备之间的无状态链接。

**参数**

struct device\_link \*link

要删除的设备链接。

**说明**

调用者必须确保在运行时PM中与此函数同步。如果链接被多次添加，则需要多次删除。热插拔设备需要特别注意它们的链接在删除时会被清除，因此不再允许调用device\_link\_del()。

### void device\_link\_remove(void \*consumer, struct device \*supplier)

删除两个设备之间的无状态链接。

**参数**

void \*consumer

链接的消费方。

struct device \*supplier

链接的供应方。

**说明**

调用程序必须确保该函数与运行时电源管理同步。

### const char \*dev\_driver\_string(const struct device \*dev)

返回设备的驱动程序名称，如果可能的话

**参数**

const struct device \*dev

要获取名称的struct 设备

**说明**

如果设备绑定了驱动程序，将返回设备的驱动程序名称。如果设备未绑定驱动程序，将返回所连接总线的名称。如果它没有连接到总线，则返回一个空字符串。

### int devm\_device\_add\_group(struct device \*dev，const struct attribute\_group \*grp)

给定一个设备，创建一个受管理的属性组

**参数**

struct device \*dev

要为其创建组的设备

const struct attribute\_group \*grp

要创建的属性组

**说明**

此功能首次创建组。如果要创建的属性文件已存在，则它会显式警告和报错。

成功返回0或失败时返回错误代码。

### void devm\_device\_remove\_group(struct device \*dev，const struct attribute\_group \*grp)

从设备中删除受管组

**参数**

struct device \*dev

要从中删除组的设备

const struct attribute\_group \*grp

要删除的组

**说明**

此功能从设备中删除一组属性。之前必须已经为此组创建了属性，否则它将失败。

### int devm\_device\_add\_groups(struct device \*dev，const struct attribute\_group \*\*groups)

创建一组受管理的属性组

**参数**

struct device \*dev

要为其创建组的设备

const struct attribute\_group \*\*groups

要创建的属性组，以NULL结尾

**说明**

此函数创建一组受管理属性组。如果创建组时发生错误，则会删除所有先前创建的组，将一切恢复到调用此函数时的原始状态。如果要创建的属性文件已存在，则它会显式警告和报错。

成功返回0或从sysfs\_create\_group返回错误代码。

### void devm\_device\_remove\_groups(struct device \*dev，const struct attribute\_group \*\*groups)

删除受管理组的列表

**参数**

struct device \*dev

要从中删除组的设备

const struct attribute\_group \*\*groups

要删除的组的以NULL结尾列表

**说明**

如果groups不为NULL，则从设备中删除指定的组。

### int device\_create\_file(struct device \*dev，const struct device\_attribute \*attr)

为设备创建sysfs属性文件。

**参数**

struct device \*dev

设备。

const struct device\_attribute \*attr

设备属性说明符。

### void device\_remove\_file(struct device \*dev，const struct device\_attribute \*attr)

移除sysfs属性文件。

**参数**

struct device \*dev

设备。

const struct device\_attribute \*attr

设备属性说明符。

### bool device\_remove\_file\_self(struct device \*dev，const struct device\_attribute \*attr)

从自身方法中移除sysfs属性文件。

**参数**

struct device \*dev

设备。

const struct device\_attribute \*attr

设备属性说明符。

**说明**

有关详细信息，请参见kernfs\_remove\_self（）。

### int device\_create\_bin\_file(struct device \*dev，const struct bin\_attribute \*attr)

为设备创建sysfs二进制属性文件。

**参数**

struct device \*dev

设备。

const struct bin\_attribute \*attr

设备二进制属性说明符。

### void device\_remove\_bin\_file(struct device \*dev，const struct bin\_attribute \*attr)

移除sysfs二进制属性文件

**参数**

struct device \*dev

设备。

const struct bin\_attribute \*attr

设备二进制属性说明符。

### void device\_initialize(struct device \*dev)

初始化设备结构。

**参数**

struct device \*dev

设备。

**说明**

通过初始化其字段，为其他层使用准备设备。它是device\_register（）的前半部分，如果由该函数调用，则还可以单独调用，以便可以使用dev的字段。特别是，可以使用get\_device（）/put\_device（）对dev进行引用计数，以便在调用此函数后使用它。

调用方必须通过0初始化dev中的所有字段，除非显式设置为其他值。最简单的方法是使用kzalloc（）来分配包含dev的结构。

**注**

使用put\_device（）放弃您的引用，而不是直接释放dev，一旦您调用了此函数。

### int dev\_set\_name（struct device \*dev，const char \*fmt，...）

设置设备名称

**参数**

struct device \*dev

设备

const char \*fmt

设备名称的格式字符串

...

可变参数

### int device\_add(struct device \*dev)

将设备添加到设备层次结构中。

**参数**

struct device \*dev

设备。

**说明**

这是device\_register（）的第2部分，但可以单独调用，前提是已单独调用了device\_initialize（）。

通过kobject\_add（）将此设备添加到k对象层次结构中，将其添加到设备的全局和兄弟列表中，然后将其添加到驱动程序模型的其他相关子系统中。

不要为任何设备结构调用此例程或device\_register()超过一次。驱动程序模型核心不适用于取消注册并重新启动的设备。 （除其他事项外，很难保证已删除dev先前版本的所有引用。）分配并注册一个全新的struct device。

注意此后无论如何都不要直接释放dev，即使它返回了错误！始终使用put\_device（）放弃引用。

### int device\_register（struct device \*dev）

向系统注册设备。

**参数**

struct device \* dev

指向设备结构的指针。

**说明**

这将在两个干净的步骤中进行-初始化设备并将其添加到系统中。这两个步骤可以分别调用，但这是最简单和最常见的。即，仅在加入层次结构之前有明确**定义**的需要使用和引用计数设备时才调用这两个帮助程序。

有关更多信息，请参见device\_initialize（）和device\_add（）的内核文档。

注意此后无论如何都不要直接释放dev，即使它返回了错误！始终使用put\_device（）放弃在此函数中初始化的引用。

### struct device \* get\_device（struct device \* dev）

增加设备的引用计数。

**参数**

struct device \* dev设备。

**说明**

这只是将调用转发到kobject\_get（），尽管我们确实注意到为传递一个空指针而提供。

### void put\_device（struct device \* dev）

递减引用计数。

**参数**

struct device \* dev

疑问中的设备。

void device\_del（struct device \* dev）

从系统中删除设备。

**参数**

struct device \* dev设备。

**说明**

这是设备取消注册序列的第一部分。这将从这里控制的列表中删除该设备，在device\_add（）中添加到的其他驱动程序模型子系统中将其删除，并将其从kobject层次结构中删除。

注意仅在手动调用device\_add（）时才应手动调用此函数。

### void device\_unregister（struct device \* dev）

从系统中注销设备。

**参数**

struct device \* dev即将离开的设备。

**说明**

我们像device\_register（）一样分成两部分来完成此操作。首先，我们使用device\_del（）从所有子系统中删除它，然后通过put\_device（）递减引用计数。如果这是最终引用计数，则将通过device\_release（）清除设备。否则，该结构将保留，直到放弃设备的最终引用为止。

### int device\_for\_each\_child（struct device \* parent，void \* data，int（\* fn）（struct device \* dev，void \* data））

设备子迭代器。

**参数**

struct device \* parent

父结构设备。void \* data回调的数据。

int（\* fn）（struct device \* dev，void \* data）

要为每个设备调用的函数。

**说明**

迭代父设备的子设备，并为每个设备调用fn，将data传递给它。我们每次检查fn的返回。如果它返回除0之外的任何值，我们将停止并返回该值。

### int device\_for\_each\_child\_reverse（struct device \* parent，void \* data，int（\* fn）（struct device \* dev，void \* data））

以相反的顺序迭代设备子项。

**参数**

struct device \* parent

父结构设备。

void \* data

回调的数据。

### int（\* fn）（struct device \* dev，void \* data）

要为每个设备调用的函数。

**说明**

迭代父设备的子设备，并为每个设备调用fn，将data传递给它。我们每次检查fn的返回。如果它返回除0之外的任何值，我们将停止并返回该值。

### struct device \* device\_find\_child（struct device \* parent，void \* data，int（\* match）（struct device \* dev，void \* data））

设备迭代器以查找特定设备。

**参数**

struct device \* parent

父结构设备

void \* data

传递给匹配功能的数据

int（\* match）（struct device \* dev，void \* data）

用于检查设备的回调函数。

**说明**

这类似于上面的device\_for\_each\_child（）函数，但它返回一个对稍后使用的“找到”的设备的引用，由匹配回调决定。如果回调返回0，则设备不匹配，并且如果可以获取对当前设备的引用，则此函数将返回到调用方，并且不会迭代任何其他设备。注意使用后，您需要使用put\_device（）放弃引用。

### struct device \* \_\_root\_device\_register（const char \* name，struct module \* owner）

分配并注册根设备。

**参数**

const char \* name

根设备名称。

struct module \* owner

根设备的所有者模块，通常为THIS\_MODULE。

**说明**

此函数分配根设备并使用device\_register（）进行注册。为了释放返回的设备，请使用root\_device\_unregister（）。根设备是虚拟设备，允许将其他设备分组为/sys/devices下的设备。使用此功能来分配根设备，然后将其用作应出现在/sys/devices/{name}下的任何设备的父设备。

在/sys/devices/{name}目录中还将包含指向sysfs中所有者目录的“module”符号链接。

成功时返回struct device指针，错误时返回ERR\_PTR()。

**注意**

您可能需要使用root\_device\_register()。

### void root\_device\_unregister(struct device \*dev)

注销并释放根设备。

**参数**

struct device \*dev

正在消失的设备

**说明**

此函数注销并清理由root\_device\_register()创建的设备。

### struct device \* device\_create\_vargs(struct class \* class, struct device \* parent, dev\_t devt, void \* drvdata, const char \* fmt, va\_list args)

创建设备并在sysfs中注册。

**参数**

struct class \* class

此设备应注册到的struct类的指针

struct device \* parent

指向此新设备的父struct device的指针，如果有

dev\_t devt

要添加的字符设备的dev\_t

void \* drvdata

要添加到设备用于回调的数据

const char \* fmt

该设备名称的字符串

va\_list args

设备名称的va\_list

**说明**

此函数可由字符设备类使用。将在sysfs中创建struct device，将其注册到指定的类别中。

如果dev\_t不是0,0，则创建“dev”文件，显示设备的dev\_t。如果传递了指向父struct device的指针，则新创建的结构device将成为sysfs中该设备的子项。将从调用中返回指向struct device的指针。可以使用此指针创建可能需要的任何其他sysfs文件。

成功时返回struct device指针，错误时返回ERR\_PTR()。

注意

传递给此函数的struct class必须先使用class\_create()调用创建。

### struct device \*device\_create(struct class \*class, struct device \*parent, dev\_t devt, void \*drvdata, const char \*fmt, ...)

创建设备并在sysfs中注册。

**参数**

struct class \*class

此设备应注册到的struct类的指针

struct device \*parent

指向此新设备的父struct device的指针，如果有

dev\_t devt

要添加的字符设备的dev\_t

void \*drvdata

要添加到设备用于回调的数据

const char \*fmt

该设备名称的字符串

...

可变参数

**说明**

此函数可由字符设备类使用。将在sysfs中创建struct device，将其注册到指定的类别中。

如果dev\_t不是0,0，则创建“dev”文件，显示设备的dev\_t。如果传递了指向父struct device的指针，则新创建的struct device将成为sysfs中该设备的子项。将从调用中返回指向struct device的指针。可以使用此指针创建可能需要的任何其他sysfs文件。

成功时返回struct device指针，错误时返回ERR\_PTR()。

**注意**

传递给此函数的struct class必须先使用class\_create()调用创建。

### struct device \*device\_create\_with\_groups(struct class \*class, struct device \*parent, dev\_t devt, void \*drvdata, const struct attribute\_group \*\*groups, const char \*fmt, ...)

创建设备并在sysfs中注册。

**参数**

struct class \*class

此设备应注册到的struct类的指针

struct device \*parent

指向此新设备的父struct device的指针，如果有

dev\_t devt

要添加的字符设备的dev\_t

void \*drvdata

要添加到设备用于回调的数据

const struct attribute\_group \*\*groups

要创建的属性组的NULL终止列表

const char \*fmt

该设备名称的字符串

...

可变参数

**说明**

此函数可由字符设备类使用。将在sysfs中创建struct device，将其注册到指定的类别中。还将自动创建在groups参数中指定的其他属性。

如果dev\_t不是0,0，则创建“dev”文件，显示设备的dev\_t。如果传递了指向父struct device的指针，则新创建的struct device将成为sysfs中该设备的子项。将从调用中返回指向struct device的指针。可以使用此指针创建可能需要的任何其他sysfs文件。

成功时返回struct device指针，错误时返回ERR\_PTR()。

**注意**

传递给此函数的struct class必须先使用class\_create()调用创建。

### void device\_destroy(struct class \*class, dev\_t devt)

删除使用device\_create()创建的设备。

**参数**

struct class \*class

此设备曾注册的struct class的指针

dev\_t devt

已注册先前设备的dev\_t

**说明**

此调用取消注册并清理使用device\_create()创建的设备。

### int device\_rename(struct device \*dev, const char \*new\_name)

重命名设备

**参数**

struct device \*dev

待重命名的struct 指针

const char \*new\_name

设备的新名称

**说明**

调用者有责任在同一设备上互斥地提供两个不同的device\_rename调用，以确保new\_name有效且不会与其他设备冲突。

重命名设备存在多个级别的竞态条件，符号链接和其他内容不能原子替换，您将得到一个“move”事件，但很难将事件连接到旧设备和新设备。设备节点根本没有重命名，内核现在甚至没有支持该功能。

同时，在重命名期间，您的目标名称可能会被另一个驱动程序占用，从而创建冲突。或者，旧名称直接从您重命名后被占用-然后您会在甚至看到“move”事件之前获取与同一DEVPATH的事件。这只是个混乱，任何新的内容都不应该依赖于内核设备重命名。除此之外，它现在甚至没有实现其他与（驱动程序核心方面非常简单的）网络设备有关的事情。

我们当前将更改udev中的网络重命名，以完全禁止在与内核使用相同的命名空间中重命名设备，因为我们无法妥善解决在交换多个接口名称时出现的问题，其中包括竞争。这意味着，eth [0-9] \*的重命名将仅允许到eth [0-9] \*之外的某个名称，出于上述原因。

在注册任何内容之前，为驱动程序编写“真实”名称，或添加其他属性以便于用户空间查找设备，或使用udev添加符号链接-但永远不要以后重命名内核设备，这是一团糟。我们甚至不想涉及它，并尝试在核心中实现缺失的部分。我们确实在驱动程序核心混乱中有其他问题需要解决。 :)

**注意**

不要调用此函数。目前，网络层调用此函数，但这将发生变化。Kay Sievers的以下文本提供了一些见解

### int device\_move(struct device \*dev，struct device \*new\_parent，enum dpm\_order dpm\_order)

将设备移动到新的父级

**参数**

struct device \*dev

要移动的struct 指针

struct device \*new\_parent

设备的新父级（可以为NULL）

enum dpm\_order dpm\_order

如何重新排序dpm\_list

### void set\_primary\_fwnode(struct device \*dev，struct fwnode\_handle \*fwnode)

更改给定设备的主要固件节点。

**参数**

struct device \*dev

要处理的设备。

struct fwnode\_handle \*fwnode

设备的新主要固件节点。

**说明**

将设备的固件节点指针设置为fwnode，但如果设备存在次要固件节点，则保留它。

### void device\_set\_of\_node\_from\_dev(struct device \*dev，const struct device \*dev2)

重用另一个设备的设备树节点

**参数**

struct device \*dev

正在设置其设备树节点的设备

const struct device \*dev2

正在重复使用其设备树节点的设备

**说明**

在首先放弃对旧节点的任何引用后，获取新设备树节点的另一个引用。

### void register\_syscore\_ops(struct syscore\_ops \*ops)

注册一组系统核心操作。

**参数**

struct syscore\_ops \*ops

要注册的系统核心操作。

### void unregister\_syscore\_ops(struct syscore\_ops \*ops)

取消注册一组系统核心操作。

**参数**

struct syscore\_ops \*ops

要取消注册的系统核心操作。

### int syscore\_suspend(void)

执行所有已注册的系统核心挂起回调。

**参数**

void

无参数

**说明**

此函数在一个CPU上启用且禁用中断运行。

### void syscore\_resume(void)

执行所有已注册的系统核心恢复回调。

**参数**

void

无参数

**说明**

此函数在一个CPU上启用且禁用中断运行。

### struct class \*\_\_class\_create(struct module \*owner，const char \*name，struct lock\_class\_key \*key)

创建一个struct 类结构

**参数**

struct module \*owner

指向要“拥有”此struct 类的模块的指针

const char \*name

此类名称的字符串指针。

struct lock\_class\_key \*key

用于此类的lock\_class\_key；由互斥锁调试使用

**说明**

用于创建可以在调用device\_create（）中使用的struct class指针。

成功返回struct class指针，或返回ERR\_PTR（）以获取错误。

请注意，此处创建的指针在完成后要通过调用class\_destroy（）来销毁。

### void class\_destroy(struct class \*cls)

销毁一个struct classstruct

**参数**

struct class \*cls

指向要销毁的结构class的指针

**说明**

请注意，要销毁的指针必须是使用class\_create()调用创建的。

### void class\_dev\_iter\_init(struct class\_dev\_iter \*iter, struct class \*class, struct device \*start, const struct device\_type \*type)

初始化类设备迭代器

**参数**

struct class\_dev\_iter \*iter

要初始化的类迭代器

struct class \*class

我们要迭代的类

struct device \*start

要迭代的设备（如果有）

const struct device\_type \*type

要迭代的设备类型，全部为NULL

**说明**

初始化类迭代器iter，使其迭代类的设备。如果设置了start，则列表迭代将从那里开始，否则如果为NULL，则从列表的开头开始迭代。

### struct device \*class\_dev\_iter\_next（struct class\_dev\_iter \*iter）

迭代到下一个设备

**参数**

struct class\_dev\_iter \*iter

要进行的类迭代器

**说明**

将iter前进到下一个设备并将其返回。如果迭代完成，则返回NULL。

返回的设备被引用并且在迭代器前进到下一个设备或退出之前不会被释放。调用方可以自由地对设备执行任何操作，包括调用回到类代码中。

### void class\_dev\_iter\_exit（struct class\_dev\_iter \*iter）

完成迭代

**参数**

struct class\_dev\_iter \*iter

要完成的类迭代器

**说明**

完成迭代。无论迭代是到达结尾还是未完成，始终调用此函数。

### int class\_for\_each\_device（struct class \*class，struct device \*start，void \*data，int（\*fn）（struct device\*，void\*））

设备迭代器

**参数**

struct class \*class

我们正在迭代的类

struct device \*start

列表中要开始的设备（如果有）

void \*data

回调使用的数据

int（\*fn）（struct device\*，void\*）

用于每个设备调用的函数

**说明**

迭代类的设备列表，并为每个设备调用fn，向其传递数据。如果设置了start，则列表迭代将从那里开始，否则如果为NULL，则从列表的开头开始迭代。

我们每次检查fn的返回。如果它返回除0以外的任何值，则退出并返回该值。

fn允许执行任何操作，包括回调到类代码。没有锁定限制。

### struct device \*class\_find\_device（struct class \*class，struct device \*start，const void \*data，int（\*match）（struct device\*，const void\*））

设备迭代器，用于查找特定的设备

**参数**

struct class \*class

我们正在迭代的类

struct device \*start

要开始的设备

const void \*data

要匹配的数据

int（\*match）（struct device\*，const void\*）

用于检查设备的函数

**说明**

这类似于上面的class\_for\_each\_dev（）函数，但它返回对后续使用“找到”的设备的引用，由match回调确定。

如果回调返回0，则设备不匹配，如果返回非零，则此函数将返回给调用方，而不迭代更多设备。

请注意，在使用后，您需要使用put\_device（）放弃引用。

match允许执行任何操作，包括回调到类代码。没有锁定限制。

### struct class\_compat \*class\_compat\_register（const char \*name）

注册兼容性类

**参数**

const char \*name

类的名称

**说明**

兼容性类是一种临时的用户空间兼容性解决方案，用于将一组类设备转换为总线设备。

### void class\_compat\_unregister（struct class\_compat \*cls）

注销兼容性类

**参数**

struct class\_compat \*cls

要注销的类

**说明**

注销兼容性类。

### int class\_compat\_create\_link（struct class\_compat \*cls，struct device \*dev，struct device \*device\_link）

创建兼容性类设备链接到总线设备

**参数**

struct class\_compat \*cls

兼容性类

struct device \*dev

目标总线设备

struct device \*device\_link

应创建“设备”链接的可选设备

### void class\_compat\_remove\_link（struct class\_compat \*cls，struct device \*dev，struct device \*device\_link）

删除兼容性类设备链接到总线设备

**参数**

struct class\_compat \*cls

兼容性类

struct device \*dev

目标总线设备

struct device \*device\_link

先前创建的“设备”链接的可选设备

### void unregister\_node（struct node \*node）

注销节点设备

**参数**

struct node \*node

节点消失

**说明**

注销节点设备node。在调用此函数之前，必须注销节点上的所有设备。

request\_firmware

### int request\_firmware（const struct firmware \*\*firmware\_p，const char \*name，struct device \*device）

发送固件请求并等待

**参数**

const struct firmware \*\*firmware\_p

固件图像的指针

const char \*name

固件文件的名称

struct device \*device

正在加载固件的设备

**说明**

将使用firmware\_p返回名称为name的固件图像，供device设备使用。

应该从允许睡眠的用户上下文中调用。

名称将在uevent环境中用作$FIRMWARE，并应足够独特，以不与此设备或任何其他设备的任何其他固件映像混淆。

调用者必须持有设备的引用计数。

此函数可以安全地在设备的挂起和恢复回调中调用。

### firmware\_request\_nowarn

int firmware\_request\_nowarn(const struct firmware \*\*firmware, const char \*name, struct device \*device)

请求可选的固件模块

**参数**

const struct firmware \*\*firmware

指向固件映像的指针

const char \*name

固件文件的名称

struct device \*device

正在加载固件的设备

**说明**

此函数的行为类似于request\_firmware()，但不在未找到文件时产生警告消息。如果直接文件系统查找失败，则启用sysfs回退机制。但是，它仍然会抑制使用它找不到固件文件的失败。因此，由驱动程序检查此调用的返回并决定何时通知用户出现错误。

### request\_firmware\_direct

int request\_firmware\_direct(const struct firmware \*\*firmware\_p, const char \*name, struct device \*device)

直接加载固件，不使用用户模式帮助程序

**参数**

const struct firmware \*\*firmware\_p

指向固件映像的指针

const char \*name

固件文件的名称

struct device \*device

正在加载固件的设备

**说明**

此函数的工作方式与request\_firmware()几乎相同，但即使无法直接从fs加载固件，它也不会回退到用户模式帮助程序。因此，它对于加载不始终存在的可选固件非常有用，而不需要额外的udev超时时间。

### int firmware\_request\_cache(struct device \* device, const char \* name)

将固件缓存以便暂停时可以使用

**参数**

struct device \* device

为其应缓存固件的设备

const char \* name

固件文件的名称

**说明**

有一些设备具有优化功能，使得设备在系统重启时无需加载固件。这种优化在从挂起中恢复时仍可能需要固件。在这些情况下，可以使用此函数确保在从挂起恢复时存在固件。此助手与使用request\_firmware\_into\_buf()或request\_firmware\_nowait()且未设置uevent的驱动程序不兼容。

### request\_firmware\_into\_buf

int request\_firmware\_into\_buf(const struct firmware \*\*firmware\_p, const char \*name, struct device \*device, void \*buf, size\_t size)

将固件加载到先前分配的缓冲区中

**参数**

const struct firmware \*\*firmware\_p

指向固件映像的指针

const char \*name

固件文件的名称

struct device \*device

正在加载固件和DMA区域的设备

void \*buf

将固件加载到其中的缓冲区的地址

size\_t size

缓冲区的大小

**说明**

此函数的工作方式与request\_firmware()几乎相同，但它不分配用于保存固件数据的缓冲区。相反，固件直接加载到buf指向的缓冲区中，并且固件\_p数据成员指向buf。

此函数也不缓存固件。

### void release\_firmware(const struct firmware \* fw)

释放与固件映像相关联的资源

**参数**

const struct firmware \* fw

要释放的固件资源

### int request\_firmware\_nowait(struct module \* module, bool uevent, const char \* name, struct device \* device, gfp\_t gfp, void \* context, void (\*cont) (const struct firmware \*fw, void \*context)

request\_firmware的异步版本

**参数**

struct module \* module

请求固件的模块

bool uevent

如果这个标志非零，那么发送uevent以复制固件映像，否则固件副本必须手动完成。

const char \* name

固件文件的名称

struct device \* device

正在加载固件的设备

gfp\_t gfp

分配标志

void \* context

将传递给cont，如果固件请求失败，则fw可能为NULL。

void (\*)(const struct firmware \*fw, void \*context) cont

在固件请求结束时将异步调用函数。

**说明**

调用者必须持有设备的引用计数。

user contexts的request\_firmware()的异步变体

如果gfp是GFP\_KERNEL，则尽可能短地睡眠，因为它可能会增加在->:c:func:probe()方法中请求固件的内置设备驱动程序的内核引导时间。

如果gfp是GFP\_ATOMIC，则无法睡眠。

### int transport\_class\_register(struct transport\_class \* tclass)

注册初始传输类

**参数**

struct transport\_class \* tclass

指针，指向要初始化的传输类结构

**说明**

传输类包含用于标识它的嵌入式类。调用者应使用零初始化此结构，然后必须使用实际传输类唯一名称初始化通用类。有一个名为DECLARE\_TRANSPORT\_CLASS()的宏可以执行此操作（声明的类仍必须注册）。

成功返回0，否则返回错误。

### void transport\_class\_unregister(struct transport\_class \*  tclass )

取消先前注册的类

**参数**

struct transport\_class \*tclass

要取消注册的传输类

**说明**

必须在释放传输类的内存之前调用。

### int anon\_transport\_class\_register(struct anon\_transport\_class \*atc)

注册一个匿名类

**参数**

struct anon\_transport\_class \*atc

要注册的匿名传输类

**说明**

匿名传输类包含传输类和容器。匿名类的想法是它实际上永远不会与任何设备属性相关联（因此节省容器存储空间）。因此它只能用于触发事件。使用prezero，然后使用DECLARE\_ANON\_TRANSPORT\_CLASS（）初始化匿名传输类存储。

### void anon\_transport\_class\_unregister(struct anon\_transport\_class \*atc)

注销匿名类

**参数**

struct anon\_transport\_class \*atc

要注销的匿名传输类的指针

**说明**

必须在释放匿名传输类的内存之前调用。

### void transport\_setup\_device(struct device \*dev)

声明一个新的设备进行传输类关联，但尚未使其可见。

**参数**

struct device \*dev

表示要添加的实体的通用设备

**说明**

通常，dev代表HBA系统中的某个组件（要么是HBA本身，要么是横跨HBA总线的远程设备）。此例程仅是一个触发点，以查看任何一组传输类是否希望与添加的设备关联。这将为类设备分配存储空间并对其进行初始化，但尚未将其添加到系统或添加属性（您可以使用transport\_add\_device完成此操作）。如果您不需要单独的设置和添加操作，请使用transport\_register\_device（参见transport\_class.h）。

### int transport\_add\_device(struct device \*dev)

声明一个新的设备进行传输类关联

**参数**

struct device \*dev

表示要添加的实体的通用设备

**说明**

通常，dev代表HBA系统中的某个组件（要么是HBA本身，要么是横跨HBA总线的远程设备）。此例程仅是一个触发点，用于将设备添加到系统并注册其属性。

### void transport\_configure\_device(struct device \*dev)

配置已经设置的设备

**参数**

struct device \*dev

表示要配置的设备的通用设备

**说明**

配置的想法只是提供设置过程中的一个点，以允许传输类在设置后从设备中提取信息。这在SCSI中使用，因为我们必须有一个设置设备才能开始使用HBA，但在发送初始查询之后，我们使用configure从设备中提取设备参数。设备不需要被添加即可进行配置。

### void transport\_remove\_device(struct device \*dev)

删除设备的可见性

**参数**

struct device \*dev

要删除的通用设备

**说明**

此调用会删除设备的可见性（对从sysfs中的用户），但不会销毁它。要完全消除设备，还必须调用transport\_destroy\_device。如果您不需要分别进行删除和销毁操作，请使用transport\_unregister\_device（参见transport\_class.h），它将为您执行两个调用。

### void transport\_destroy\_device(struct device \*dev)

销毁已删除的设备

**参数**

struct device \*dev

要从传输类中消除的设备。

**说明**

此调用触发与传输类dev相关联的存储的消除。注意它实际上只是释放对classdev的引用。直到最后引用计数变为零，内存才不会被释放。还请注意，classdev保留对dev的引用计数，因此只要传输类设备保留，dev也将保留。

### int device\_bind\_driver(struct device \*dev)

将驱动程序绑定到一个设备。

**参数**

struct device \*dev

设备。

**说明**

允许手动将驱动程序绑定到设备。调用者必须已经设置了dev->驱动程序。

请注意，这不会修改总线引用计数。在调用之前，请验证是否已计入（从驱动程序的probe（）方法没有其他努力是可以的）。

必须在设备锁定的情况下调用此函数。

### void wait\_for\_device\_probe(void)

**参数**

void

没有参数

**说明**

等待设备探测完成。

### int device\_attach(struct device \*dev)

尝试将设备附加到驱动程序。

**参数**

struct device \*dev

设备。

**说明**

遍历总线拥有的驱动程序列表，并为每个对调用driver\_probe\_device（）。如果找到兼容对，中断并返回。如果设备绑定到驱动程序，则返回 1；否则 0 如果没有找到匹配的驱动程序；-ENODEV 如果设备未注册。

如果为USB接口调用，则必须保持dev->parent锁定。

### int driver\_attach(struct device\_driver \*drv)

尝试将驱动程序绑定到设备。

**参数**

struct device\_driver \*drv

驱动程序。

**说明**

遍历总线上的设备列表，并尝试将驱动程序与它们匹配。如果driver\_probe\_device()返回0并且dev->driver已设置，那么我们找到了一个兼容的设备对。

### void device\_release\_driver(struct device \*dev)

手动从驱动程序分离设备。

**参数**

struct device \*dev

设备。

**说明**

手动从驱动程序中分离设备。当针对USB接口调用此函数时，必须持有dev->parent锁。

如果要使用dev->parent锁调用此函数，请确保在先解除设备的消费者绑定或者在dev->parent锁下可以获取其锁。

### struct platform\_device \*platform\_device\_register\_resndata(struct device \*parent， const char \*name，int id，const struct resource \*res，unsigned int num，const void \*data，size\_t size)

添加具有资源和平台特定数据的平台级设备

**参数**

struct device \*parent

我们正在添加的设备的父设备

const char \*name

我们正在添加的设备的基本名称

int id

实例ID

const struct resource \*res

需要为设备分配的资源集

unsigned int num

资源数量

const void \*data

此平台设备的平台特定数据

size\_t size

平台特定数据的大小

**说明**

成功返回struct platform\_device指针，错误返回ERR\_PTR()。

### struct platform\_device \*platform\_device\_register\_simple(const char \*name，int id，const struct resource \*res，unsigned int num)

添加平台级设备及其资源

**参数**

const char \*name

我们正在添加的设备的基本名称

int id

实例ID

const struct resource \*res

需要为设备分配的资源集

unsigned int num

资源数量

**说明**

此函数创建一个简单的平台设备，需要最少的资源和内存管理。提供释放用于释放为设备分配的内存的预设释放函数允许使用此类设备的驱动程序在等待对设备的最后引用被删除之前卸载。

此接口主要用于以前的直接探测硬件的驱动程序。因为这种驱动程序自己创建sysfs设备节点，而不是让系统基础架构处理这样的设备enum任务，所以它们不完全符合Linux驱动程序模型。特别是，当这些驱动程序作为模块构建时，它们无法“热插拔”。

成功返回struct platform\_device指针，错误返回ERR\_PTR()。

### struct platform\_device \*platform\_device\_register\_data(struct device \*parent， const char \*name，int id，const void \*data，size\_t size)

添加带有平台特定数据的平台级设备

**参数**

struct device \*parent

我们正在添加的设备的父设备

const char \*name

我们正在添加的设备的基本名称

int id

实例ID

const void \*data

此平台设备的平台特定数据

size\_t size

平台特定数据的大小

**说明**

此函数创建一个简单的平台设备，需要最少的资源和内存管理。提供释放用于释放为设备分配的内存的预设释放函数允许使用此类设备的驱动程序在等待对设备的最后引用被删除之前卸载。成功返回struct platform\_device指针，错误返回ERR\_PTR()。

### struct resource \*platform\_get\_resource(struct platform\_device \*dev， unsigned int type， unsigned int num)

获取设备的资源

**参数**

struct platform\_device \*dev

平台设备

unsigned int type

资源类型

unsigned int num

资源索引

### struct resource \* platform\_get\_resource(struct platform\_device \* dev， unsigned int type， unsigned int num)

获取设备的资源

**参数**

struct platform\_device \* dev

平台设备

unsigned int type

资源类型

unsigned int num

资源索引

### int platform\_get\_irq(struct platform\_device \*dev， unsigned int num)

获取设备的IRQ

**参数**

struct platform\_device \*dev

平台设备

unsigned int num

IRQ编号索引

### int platform\_irq\_count(struct platform\_device \*dev)

计算平台设备使用的IRQ数量

**参数**

struct platform\_device \*dev

平台设备

**返回**

平台设备使用的IRQ数量或EPROBE\_DEFER。

### struct resource \*platform\_get\_resource\_byname(struct platform\_device \*dev， unsigned int type， const char \*name)

按名称获取设备的资源

**参数**

struct platform\_device \*dev

平台设备

unsigned int type

资源类型

const char \*name

资源名称

### int platform\_get\_irq\_byname(struct platform\_device \*dev， const char \*name)

按名称获取设备的IRQ

**参数**

struct platform\_device \*dev

平台设备

const char \*name

IRQ名称

### int platform\_add\_devices(struct platform\_device \*\*devs， int num)

添加平台设备的数量

**参数**

struct platform\_device \*\*devs

要添加的平台设备数组

int num

数组中平台设备的数量

### void platform\_device\_put(struct platform\_device \*pdev)

销毁平台设备

**参数**

struct platform\_device \*pdev

要释放的平台设备

**说明**

释放与平台设备相关联的所有内存。此函数只能在错误情况下外部调用。所有其他用途都是错误。

struct platform\_device \*platform\_device\_alloc(const char \*name, int id)

创建平台设备

**参数**

const char \*name

我们要添加的设备的基本名称

int id

实例ID

**说明**

创建一个平台设备对象，该对象可以附加其他对象，并且在释放时附加的对象将被释放。

### int platform\_device\_add\_resources(struct platform\_device \*pdev, const struct resource \*res, unsigned int num)

向平台设备添加资源

**参数**

struct platform\_device \*pdev

由platform\_device\_alloc分配的平台设备，以添加资源

const struct resource \*res

需要为设备分配的资源集

unsigned int num

资源数量

**说明**

将资源的副本添加到平台设备中。与资源相关联的内存将在释放平台设备时释放。

### int platform\_device\_add\_data(struct platform\_device \*pdev, const void \*data, size\_t size)

向平台设备添加特定于平台的数据

**参数**

struct platform\_device \*pdev

由platform\_device\_alloc分配的平台设备，以添加资源

const void \*data

平台设备的特定于平台的数据

size\_t size

平台特定数据的大小

**说明**

将平台特定数据的副本添加到平台设备的platform\_data指针中。与平台数据相关联的内存将在释放平台设备时释放。

### int platform\_device\_add\_properties(struct platform\_device \* pdev，const struct property\_entry \* properties)

向平台设备添加内置属性

**参数**

struct platform\_device \* pdev

要添加属性的平台设备

const struct property\_entry \*属性

要添加的属性的以null结尾的数组

**说明**

该函数将深度复制properties并将副本附加到平台设备上。与属性相关联的内存将在释放平台设备时释放。

### int platform\_device\_add(struct platform\_device \*pdev)

将平台设备添加到设备层次结构中

**参数**

struct platform\_device \*pdev

要添加的平台设备

**说明**

这是platform\_device\_register()的第2部分，但可以单独调用iff pdev由platform\_device\_alloc()分配。

### void platform\_device\_del(struct platform\_device \*pdev)

删除平台级设备

**参数**

struct platform\_device \*pdev

要删除的平台设备

**说明**

请注意，此函数还将释放设备所拥有的所有基于内存和端口的资源（dev->resource）。此函数只能在错误情况下外部调用。所有其他用途都是错误。

### int platform\_device\_register(struct platform\_device \*pdev)

添加平台级设备

**参数**

struct platform\_device \*pdev

要添加的平台设备

### void platform\_device\_unregister(struct platform\_device \*pdev)

注销平台级设备

**参数**

struct platform\_device \*pdev

要注销的平台设备

**说明**

注销分为2个步骤。首先，我们释放所有资源并将其从子系统中删除，然后通过调用platform\_device\_put()降低参考计数。

### struct platform\_device \*platform\_device\_register\_full(const struct platform\_device\_info \*pdevinfo)

添加具有资源和特定于平台的数据的平台级设备

**参数**

const struct platform\_device\_info \*pdevinfo

用于创建设备的数据

**说明**

在成功时返回struct platform\_device指针，或在出错时返回ERR\_PTR()。

### int \_\_platform\_driver\_register(struct platform\_driver \*drv, struct module \*owner)

为平台级设备注册驱动程序

**参数**

struct platform\_driver \*drv

平台驱动程序结构

struct module \*owner

拥有模块/驱动程序

### void platform\_driver\_unregister(struct platform\_driver \*drv)

为平台级设备注销驱动程序

**参数**

struct platform\_driver \*drv

平台驱动程序结构

### int \_\_platform\_driver\_probe(struct platform\_driver \*drv, int (\*probe)(struct platform\_device\*), struct module \*module)

为非可热插拔设备注册驱动程序

**参数**

struct platform\_driver \*drv

平台驱动程序结构

int (\*probe)(struct platform\_device \*)

驱动程序探测例程，可能来自\_\_init部分

struct module \*module

将成为驱动程序所有者的模块

**说明**

当您知道设备不是可热插拔的并且已经注册并且想要在驱动程序绑定到设备后从内存中删除其运行一次的probe()基础设施时，请使用此项。驱动器已绑定到设备。

这种情况下的一种典型用途是针对集成在SoC处理器中的控制器驱动程序，其中控制器设备已配置为板设置的一部分。

请注意，这与延迟探测不兼容。

如果驱动程序已注册并绑定到设备，则返回零，否则返回负错误代码并且驱动程序未注册。

### struct platform\_device \*\_\_platform\_create\_bundle(struct platform\_driver \*driver，int (\*probe)(struct platform\_device\*)，struct resource \*res，unsigned int n\_res，const void \*data，size\_t size，struct module \*module)

注册驱动程序并创建相应的设备

**参数**

struct platform\_driver \*driver

平台驱动程序结构

int (\*probe)(struct platform\_device \*)

驱动程序探测例程，可能来自\_\_init部分

struct resource \*res

需要为设备分配的资源集

unsigned int n\_res

资源数量

const void \*data

平台设备的平台特定数据

size\_t size

平台特定数据的大小

struct module \*module

将作为驱动程序所有者的模块

**说明**

用于直接探测硬件并注册单个平台设备和相应平台驱动程序的旧式模块中。成功时返回 struct platform\_device 指针，错误时返回 ERR\_PTR()。

### int \_\_platform\_register\_drivers(struct platform\_driver \*const \*drivers, unsigned int count, struct module \*owner)

注册平台驱动程序数组

**参数**

struct platform\_driver \* const \*drivers

要注册的驱动程序数组

unsigned int count

要注册的驱动程序的数量

struct module \*owner

拥有驱动程序的模块

**说明**

注册由数组指定的平台驱动程序。在注册驱动程序失败时，所有先前注册的驱动程序都将取消注册。调用此API的调用者应使用 platform\_unregister\_drivers() 按相反顺序取消注册驱动程序。

**返回**

成功时为 0，失败时为负错误代码。

### void platform\_unregister\_drivers(struct platform\_driver \*const \*drivers, unsigned int count)

取消注册平台驱动程序数组

**参数**

struct platform\_driver \* const \*drivers

要取消注册的驱动程序数组

unsigned int count

要取消注册的驱动程序的数量

**说明**

取消注册由数组指定的平台驱动程序。这通常用于与之前调用 platform\_register\_drivers() 补充使用。驱动程序按照它们注册的相反顺序取消注册。

### int bus\_for\_each\_dev(struct bus\_type \*bus, struct device \*start, void \*data, int (\*fn)(struct device\*, void\*))

设备迭代器

**参数**

struct bus\_type \*bus

总线类型

struct device \*start

开始迭代的设备

void \*data

回调数据

int (\*fn)(struct device \*, void \*)

要为每个设备调用的函数

**说明**

迭代总线的设备列表，并为每个设备调用 fn，将 data 传递给它。如果 start 不为 NULL，则从该设备开始迭代。

我们每次都会检查 fn 的返回。如果它返回除 0 以外的任何值，则我们会退出并返回该值。

**注意**

返回非零值的设备不会被保留，其引用计数也不会增加。如果调用者需要保留此数据，它应该这样做，并在提供的回调中增加引用计数。

### struct device \*bus\_find\_device(struct bus\_type \*bus, struct device \*start, const void \*data, int (\*match)(struct device \*dev, const void \*data))

设备迭代器，用于定位特定设备。

**参数**

struct bus\_type \*bus

总线类型

struct device \*start

要开始迭代的设备

const void \*data

传递给 match 函数的数据

int (\*match)(struct device \*dev, const void \*data)

检查设备的回调函数

**说明**

与上面的 bus\_for\_each\_dev() 函数类似，但它返回一个“找到”的设备引用，以供以后使用，由 match 回调确定。

如果回调返回 0，则设备不匹配，如果回调返回非零，则此函数将返回到调用者并且不会迭代其他设备。

### struct device \* bus\_find\_device\_by\_name(struct bus\_type \* bus, struct device \* start, const char \* name)

设备迭代器，用于定位特定名称的特定设备

**参数**

struct bus\_type \* bus

总线类型

struct device \* start

要开始迭代的设备

const char \* name

要匹配的设备名称

**说明**

类似于上面的 bus\_find\_device() 函数，但它自动处理按名称搜索，无需编写另一个 strcmp 匹配函数。

### struct device \*subsys\_find\_device\_by\_id(struct bus\_type \*subsys, unsigned int id, struct device \*hint)

通过特定的enum号查找设备

**参数**

struct bus\_type \*subsys

子系统

unsigned int id

在struct device 中的索引“id”

struct device \*hint

要首先检查的设备

**说明**

检查提示的下一个对象，如果它匹配，则直接返回它，否则退回到完整列表搜索。无论如何，都会取得返回对象的参考。

### int bus\_for\_each\_drv(struct bus\_type \*bus, struct device\_driver \*start, void \*data, int (\*fn)(struct device\_driver\*, void\*))

驱动程序迭代器

**参数**

struct bus\_type \*bus

所处理的总线

struct device\_driver \*start

要开始迭代的驱动程序

void \*data

要传递给回调的数据

int (\*fn)(struct device\_driver \*, void \*)

对每个驱动程序调用的函数

**说明**

这与上面的设备迭代器几乎相同。我们迭代属于总线的每个驱动程序，并为每个驱动程序调用 fn。如果 fn 返回的值不为 0，则我们中断并返回它。如果 start 不为 NULL，则将其用作列表的头。

**注意**

我们不返回返回非零值的驱动程序，也不保留该驱动程序的引用计数。如果调用方需要知道该信息，它必须在回调中设置它。它还必须确保增加引用计数，以便在返回给调用方之前不会消失。

### int bus\_rescan\_devices（struct bus\_type \*bus）

重新扫描总线上的设备以寻找可能的驱动程序

**参数**

struct bus\_type \*bus

要扫描的总线。

**说明**

此函数将查找未连接驱动程序的总线上的设备，并通过对未绑定设备调用device\_attach（）来重新扫描现有驱动程序，以查看是否与任何驱动程序匹配。

### int device\_reprobe（struct device \*dev）

移除设备的驱动程序并探测新的驱动程序

**参数**

struct device \*dev

要重新探测的设备

**说明**

此函数会分离给定设备的附加驱动程序（如果有），并重新启动驱动程序探测过程。如果在设备生命周期中更改了探测标准并且应该相应更改驱动程序附件，则使用它。

### int bus\_register（struct bus\_type \*bus）

注册一个驱动程序核心子系统

**参数**

struct bus\_type \*bus

要注册的总线

**说明**

一旦获得该内容，我们就使用kobject基础设施注册总线，然后注册它具有的子系统属于子系统的设备和驱动程序。

### void bus\_**unregister**（struct bus\_type \*bus）

从系统中删除总线

**参数**

struct bus\_type \*bus

总线。

**说明**

取消注册子系统和总线本身。最后，我们调用bus\_put（）来释放引用计数

### void subsys\_dev\_iter\_init（struct subsys\_dev\_iter \*iter、struct bus\_type \*subsys、struct device \*start、const struct device\_type \*type）

初始化subsys设备迭代器

**参数**

struct subsys\_dev\_iter \*iter

要初始化的subsys迭代器

struct bus\_type \*subsys

我们要迭代的subsys

struct device \*start

要迭代的设备（如果有）

const struct device\_type \*type

要迭代的设备的device\_type，全部为NULL

**说明**

初始化subsys迭代器iter，以便迭代subsys的设备。如果设置了start，则列表迭代将从start开始，否则如果它为NULL，则从列表开头开始迭代。

### struct device \*subsys\_dev\_iter\_next（struct subsys\_dev\_iter \*iter）

迭代到下一个设备

**参数**

struct subsys\_dev\_iter \*iter

要继续的subsys迭代器

**说明**

将iter继续到下一个设备并返回它。如果迭代完成，则返回NULL。

返回的设备已被引用，直到迭代器进入下一个设备或退出为止都不会被释放。调用者可以自由地对设备做任何想做的事情，包括回调到子系统代码。

### void subsys\_dev\_iter\_exit（struct subsys\_dev\_iter \*iter）

完成迭代

**参数**

struct subsys\_dev\_iter \*iter

要完成的subsys迭代器

**说明**

完成迭代。无论迭代运行到末尾还是没有，都要始终调用此函数。

### int subsys\_system\_register（struct bus\_type \*subsys、const struct attribute\_group \*\*groups）

在/sys/devices/system/注册子系统

**参数**

struct bus\_type \*subsys

系统子系统

const struct attribute\_group \*\*groups

根设备的默认属性

**说明**

所有“系统”子系统都有一个/sys/devices/system/<name>根设备，名称为子系统。根设备可以携带子系统范围的属性。所有注册的设备都在此单个根设备下，并以附加一个简单的enum号码的子系统命名。注册的设备不是显式命名的；设备中只需设置‘id’即可。

不要为任何新内容使用此接口，它仅存在于与错误想法的兼容性中。新的子系统应使用简单的子系统；并且应将子系统范围的属性添加到子系统目录本身，而不是放置在/sys/devices/system/<name>的创建虚假根设备中。

### int subsys\_virtual\_register（struct bus\_type \*subsys、const struct attribute\_group \*\*groups）

在/sys/devices/virtual/注册子系统

**参数**

struct bus\_type \*subsys

虚拟子系统

const struct attribute\_group \*\*groups

根设备的默认属性

**说明**

所有的“虚拟”子系统都有一个名为子系统的/sys/devices/system/<name>根设备。根设备可以携带子系统范围的属性。所有注册的设备都在此单个根设备下。没有设备命名的限制。这是用于需要sysfs接口的内核软件构造。

## 设备驱动程序DMA管理

### int dma\_alloc\_from\_dev\_coherent（struct device \* dev、ssize\_t size、dma\_addr\_t \* dma\_handle、void \*\* ret）

从设备一致池中分配内存

**参数**

struct device \* dev

我们从中分配内存的设备

ssize\_t size

所请求的内存区域的大小

dma\_addr\_t \* dma\_handle

这将填充正确的dma句柄

void \*\* ret

此指针将填充分配区域的虚拟地址。

**说明**

此函数仅应从每个设备的一致内存池中调用，以支持从此设备分配内存。

如果dma\_alloc\_coherent应继续从一般内存区域分配，则返回0，否则返回！0以返回ret。

### int dma\_release\_from\_dev\_coherent（struct device \*dev、int order、void \* vaddr）

向设备一致内存池释放内存

**参数**

struct device \* dev

分配内存的设备

int order

分配的页面顺序

void \* vaddr

已分配页面的虚拟地址

**说明**

该函数检查内存是否是从设备一致内存池中分配，并释放该内存。

如果成功释放了内存，则返回1；如果调用者应该继续从通用内存池释放内存，则返回0。

### int dma\_mmap\_from\_dev\_coherent(struct device \* dev, struct vm\_area\_struct \* vma, void \* vaddr, size\_t size, int \* ret)

映射来自设备一致内存池的内存

**参数**

struct device \* dev

分配内存的设备

struct vm\_area\_struct \* vma

用户空间内存的vm\_area

void \* vaddr

通过dma\_alloc\_from\_dev\_coherent返回的cpu地址

size\_t size

已分配内存缓冲区的大小

int \* ret

remap\_pfn\_range()的结果

**说明**

该函数检查内存是否是从设备一致内存池中分配，并将该内存映射到提供的vma中。

如果vaddr属于设备一致内存池，则返回1，调用者应该返回ret；如果应该从通用区域映射内存，则返回0。

### void \* dmam\_alloc\_coherent(struct device \* dev, size\_t size, dma\_addr\_t \* dma\_handle, gfp\_t gfp)

分配管理的dma\_alloc\_coherent()

**参数**

struct device \* dev

要为其分配一致内存的设备

size\_t size

分配的大小

dma\_addr\_t \* dma\_handle

用于分配的DMA句柄的输出参数

gfp\_t gfp

分配标志

**说明**

管理的dma\_alloc\_coherent()。使用此函数分配的内存将在驱动程序分离时自动释放。

**返回**

成功时分配的内存指针，失败时为NULL。

### void dmam\_free\_coherent(struct device \*dev, size\_t size, void \*vaddr, dma\_addr\_t dma\_handle)

管理的dma\_free\_coherent()

**参数**

struct device \*dev

要为其释放一致内存的设备

size\_t size

分配的大小

void \*vaddr

要释放的内存的虚拟地址

dma\_addr\_t dma\_handle

要释放的内存的DMA句柄

**说明**

管理的dma\_free\_coherent()。

### void \*dmam\_alloc\_attrs(struct device \*dev, size\_t size, dma\_addr\_t \*dma\_handle, gfp\_t gfp, unsigned long attrs)

管理的dma\_alloc\_attrs()

**参数**

struct device \*dev

要为其分配不一致内存的设备

size\_t size

分配的大小

dma\_addr\_t \*dma\_handle

用于分配的DMA句柄的输出参数

gfp\_t gfp

分配标志

unsigned long attrs

DMA\_ATTR\_\*命名空间中的标志。

**说明**

管理的dma\_alloc\_attrs()。使用此函数分配的内存将在驱动程序分离时自动释放。

**返回**

成功时分配的内存指针，失败时为NULL。

### int dmam\_declare\_coherent\_memory(struct device \* dev, phys\_addr\_t phys\_addr, dma\_addr\_t device\_addr, size\_t size, int flags)

管理的dma\_declare\_coherent\_memory()

**参数**

struct device \* dev

要声明一致性内存的设备

phys\_addr\_t phys\_addr

要声明的一致内存的物理地址

dma\_addr\_t device\_addr

要声明的一致内存的设备地址

size\_t size

要声明的一致性内存大小

int flags

标志

**说明**

管理的dma\_declare\_coherent\_memory()。

**返回**

成功时为0，失败时为-errno。

### void dmam\_release\_declared\_memory(struct device \* dev)

管理的dma\_release\_declared\_memory()。

**参数**

struct device \* dev

要为其释放已声明的一致性内存的设备

**说明**

管理的dma\_release\_declared\_memory()。

## 设备驱动程序PnP支持

### int pnp\_register\_protocol(struct pnp\_protocol \*protocol)

向PnP层添加PnP协议

**参数**

struct pnp\_protocol \*protocol

指向相应pnp\_protocol结构的指针

Ex protocols: ISAPNP, PNPBIOS, etc

### void pnp\_unregister\_protocol(struct pnp\_protocol \*protocol)

从PnP层删除PnP协议

**参数**

struct pnp\_protocol \*protocol

指向相应pnp\_protocol结构的指针

### struct pnp\_dev \*pnp\_request\_card\_device(struct pnp\_card\_link \*clink, const char \*id, struct pnp\_dev \*from)

在指定的卡下查找PnP设备

**参数**

struct pnp\_card\_link \*clink

卡链接的指针，不能为NULL

const char \*id

指向PnP ID结构的指针，解释查找设备的规则

struct pnp\_dev \*from

搜索起始位置。如果为NULL，则从头开始。

### void pnp\_release\_card\_device(struct pnp\_dev \*dev)

当驱动程序不再需要设备时调用此函数

**参数**

struct pnp\_dev \*dev

指向PnP设备结构的指针

### int pnp\_register\_card\_driver(struct pnp\_card\_driver \*drv)

向PnP层注册PnP卡驱动程序

**参数**

struct pnp\_card\_driver \*drv

指向要注册的驱动程序的指针

### void pnp\_unregister\_card\_driver(struct pnp\_card\_driver \*drv)

从PnP层注销PnP卡驱动程序

**参数**

struct pnp\_card\_driver \*drv

指向要注销的驱动程序的指针

### struct pnp\_id \*pnp\_add\_id(struct pnp\_dev \*dev, const char \*id)

向指定设备添加一个EISA ID

**参数**

struct pnp\_dev \*dev

指向所需设备的指针

const char \*id

指向EISA ID字符串的指针

### int pnp\_start\_dev(struct pnp\_dev \*dev)

启动PnP设备的低级函数

**参数**

struct pnp\_dev \*dev

指向所需设备的指针

**说明**

假设资源已经分配

### int pnp\_stop\_dev(struct pnp\_dev \*dev)

禁用PnP设备的低级函数

**参数**

struct pnp\_dev \*dev

指向所需设备的指针

**说明**

不释放资源

### int pnp\_activate\_dev(struct pnp\_dev \*dev)

激活PnP设备供使用

**参数**

struct pnp\_dev \*dev

指向所需设备的指针

**说明**

不验证或设置资源，因此要小心。

### int pnp\_disable\_dev(struct pnp\_dev \*dev)

禁用设备

**参数**

struct pnp\_dev \*dev

指向所需设备的指针

请提供正确的 PnP 协议，以便其他设备可以使用资源

### int pnp\_is\_active(struct pnp\_dev \*dev)

根据当前资源确定设备是否激活

**参数**

struct pnp\_dev \*dev

指向所需 PnP 设备的指针

## 用户空间 IO 设备

### void uio\_event\_notify(struct uio\_info \*info)

触发中断事件

**参数**

struct uio\_info \*info

UIO 设备功能

### int \_\_uio\_register\_device(struct module \*owner, struct device \*parent, struct uio\_info \*info)

注册新的用户空间 IO 设备

**参数**

struct module \*owner

创建新设备的模块

struct device \*parent

父设备

struct uio\_info \*info

UIO 设备功能

**说明**

成功时返回零或负错误代码。

### void uio\_unregister\_device(struct uio\_info \*info)

注销工业 IO 设备

**参数**

struct uio\_info \*info

UIO 设备功能

### struct uio\_mem

UIO 内存区域的说明

**定义**

struct uio\_mem {

const char \*name;

phys\_addr\_t addr;

unsigned long offs;

resource\_size\_t size;

int memtype;

void \_\_iomem \*internal\_addr;

struct uio\_map \*map;

};

**成员**

name

用于标识的内存区域的名称

addr

设备内存的地址舍入为页面大小（由于 addr 可以是逻辑、虚拟或物理的，因此使用 phys\_addr\_t 可以始终足够大以处理任何地址类型）

offs

设备内存在页面内的偏移量

size

IO 的大小（页面大小的倍数）

memtype

addr 指向的内存类型

internal\_addr

addr 的 ioremap 版本，供驱动内部使用

map

仅供 UIO 核心使用。

### struct uio\_port

UIO 端口区域的说明

**定义**

struct uio\_port {

const char \*name;

unsigned long start;

unsigned long size;

int porttype;

struct uio\_portio \*portio;

};

**成员**

name

用于标识的端口区域的名称

start

端口区域的起始位置

size

端口区域的大小

porttype

端口类型（参见下面的 UIO\_PORT\_\*）

portio

仅供 UIO 核心使用。

### struct uio\_info

UIO 设备功能

**定义**

struct uio\_info {

struct uio\_device \*uio\_dev;

const char \*name;

const char \*version;

struct uio\_mem mem[MAX\_UIO\_MAPS];

struct uio\_port port[MAX\_UIO\_PORT\_REGIONS];

long irq;

unsigned long irq\_flags;

void \*priv;

irqreturn\_t (\*handler)(int irq, struct uio\_info \*dev\_info);

int (\*mmap)(struct uio\_info \*info, struct vm\_area\_struct \*vma);

int (\*open)(struct uio\_info \*info, struct inode \*inode);

int (\*release)(struct uio\_info \*info, struct inode \*inode);

int (\*irqcontrol)(struct uio\_info \*info, s32 irq\_on);

};

**成员**

uio\_dev

该信息所属的 UIO 设备

name

设备名称

version

设备驱动程序版本

mem

可映射内存区域的列表，size==0 为列表的结尾

port

端口区域的列表，size==0 为列表的结尾

irq

中断号或 UIO\_IRQ\_CUSTOM

irq\_flags

request\_irq() 的标志

priv

可选私有数据

handler

设备的中断处理程序

mmap

该 uio 设备的 mmap 操作

open

该 uio 设备的打开操作

release

该 uio 设备的释放操作

irqcontrol

往 /dev/uioX 写入 0/1 时禁用/启用 IRQsX

# 设备电源管理

## 设备电源管理基础知识

版权所有© 2010-2011 Rafael J. Wysocki <rjw@sisk.pl>，Novell Inc.

版权所有© 2010 Alan Stern <stern@rowland.harvard.edu>

版权所有© 2016 Intel Corporation

Linux 中的大部分代码都是设备驱动程序，因此 Linux 中的大部分电源管理（PM）代码也是特定于驱动程序的。大多数驱动程序将做很少的工作；而其他一些驱动程序，特别是面向小电池平台（如手机）的驱动程序，则会做很多工作。

本文概述了驱动程序如何与系统范围的电源管理目标交互，重点强调了所有连接到驱动程序模型核心的模型和接口。请将其作为特定驱动程序所需的领域专业知识的基础。

### 设备电源管理的两个模型

驱动程序将使用以下一个或两个模型将设备置入低功耗状态

系统休眠模型：

驱动程序可以作为进入像“挂起”（也称为“挂起到 RAM”）或（大多数带有磁盘的系统的）“休眠”（也称为“挂起到磁盘”）等系统范围的低功耗状态的一部分进入低功耗状态。

这是通过实现各种角色特定的挂起和恢复方法来协作完成的，以清除电源和软件子系统，然后重新激活它们，而不会丢失数据。

某些驱动程序可以管理硬件唤醒事件，从而使系统离开低功耗状态。可以使用相关的 /sys/devices/.../power/wakeup 文件（对于以太网驱动程序，也可以使用 ethtool 使用的 ioctl 接口来实现该功能）启用或禁用此功能。启用它可能会消耗一些功率，但可以让整个系统更频繁地进入低功耗状态。

运行时电源管理模型：

设备在运行时也可以置于低功耗状态，原则上独立于其他电源管理活动。但是，设备通常不独立于彼此（例如，除非所有子设备都已挂起，否则父设备无法挂起）。此外，根据设备所在的总线类型，可能需要对设备执行某些总线特定的操作。在运行时置于低功耗状态的设备在全局系统转换（挂起或休眠）期间可能需要特殊处理。

出于这些原因，运行时电源管理涉及设备驱动程序本身，以及适当的子系统（总线类型、设备类型或设备类别）驱动程序和 PM 核心。与系统睡眠电源管理情况一样，它们需要通过实现各种特定角色的暂停和恢复方法进行协作，以便硬件能够干净地关闭和重新激活，而不会丢失数据或服务。

除了它们非常特定于系统和经常特定于设备之外，关于这些低功耗状态并没有太多可说的。此外，如果足够多的设备已被置于低功耗状态（在运行时），则效果可能非常类似于进入某些系统范围的低功耗状态（系统睡眠）...并且存在协同作用，使得使用运行时 PM 的几个驱动程序可以将系统置于更深的节能选项状态中。

大多数已挂起的设备都已静止所有 I/O不存在更多的 DMA 或 IRQ（除了唤醒事件），不再读取或写入任何数据，上游驱动程序的请求也不再接受。然而，给定的总线或平台可能具有不同的要求。

硬件唤醒事件的例子包括实时时钟的警报、网络唤醒 LAN 数据包、键盘或鼠标活动以及媒体插入或移除（针对 PCMCIA、MMC/SD、USB 等）。

### 进入系统睡眠状态的接口

为子系统（总线类型、设备类型、设备类别）和设备驱动程序提供编程接口，使它们能够参与所关注设备的电源管理。这些接口涵盖了系统睡眠和运行时电源管理。

#### 设备电源管理操作

在子系统级别和设备驱动程序级别上实现设备电源管理操作是通过定义和填充类型为 struct dev\_pm\_ops 的对象来完成的，该类型定义在 include/linux/pm.h 中。其中包含的方法的角色将在下文中解释。目前，记住最后三个方法是特定于运行时电源管理的，而其余方法则用于系统范围的电源转换。

至少对于某些子系统，还存在已弃用的“旧”或“遗留”接口用于电源管理操作。这种方法不使用 struct dev\_pm\_ops 对象，并且仅适用于以有限的方式实现系统睡眠电源管理方法。因此，在本文档中不进行说明，请直接参考源代码了解更多信息。

#### 子系统级方法

挂起和恢复设备的核心方法存在于由 struct dev\_pm\_ops 指向的 struct dev\_pm\_domain 的 ops 成员，或由 struct bus\_type、struct device\_type 和 struct class 的 pm 成员中。它们主要是针对为平台和总线编写基础设施的人员感兴趣，例如 PCI 或 USB，或设备类型和设备类别驱动程序的编写者。它们还与设备驱动程序的编写者相关，其子系统（PM 域、设备类型、设备类别和总线类型）不提供所有电源管理方法。

总线驱动程序根据硬件及其使用驱动程序的方式实现这些方法；PCI 的实现方式与 USB 不同，等等。很少有人编写子系统级驱动程序；大多数驱动程序代码是建立在特定于总线的框架代码之上的“设备驱动程序”。

有关这些驱动程序调用的更多信息，请参见后面的说明；它们会按照驱动程序模型树中的父子顺序针对每个设备按阶段调用。

#### /sys/devices/.../power/wakeup 文件

驱动程序模型中的所有设备对象都包含控制系统唤醒事件处理的字段（硬件信号，可以强制系统退出睡眠状态）。这些字段由总线或设备驱动程序代码使用 device\_set\_wakeup\_capable() 和 device\_set\_wakeup\_enable() 初始化，定义在 include/linux/pm\_wakeup.h 中。

power.can\_wakeup 标志只记录设备（及其驱动程序）是否能够物理支持唤醒事件。device\_set\_wakeup\_capable() 例程会影响此标志。power.wakeup 字段是一个指向 struct wakeup\_source 类型的对象的指针，用于控制该设备是否应使用其系统唤醒机制，并通知 PM 核心由设备发出的系统唤醒事件。这个对象仅对能够唤醒的设备存在（即其 can\_wakeup 标志已设置），并且是由 device\_set\_wakeup\_capable() 创建（或删除）的。

设备是否能够发出唤醒事件是硬件问题，内核负责跟踪它。相反，唤醒能力设备是否应发出唤醒事件是一项政策决策，并由用户空间通过 sysfs 属性进行管理power/wakeup 文件。用户空间可以向该文件中写入“启用”或“禁用”字符串，以指示是否可以相应地信号系统唤醒。如果 power.wakeup 对于给定设备存在，则该文件仅存在，并由 device\_set\_wakeup\_capable() 创建（或删除）。从文件中读取将返回相应的字符串。

功率/唤醒文件中的初始值对于大多数设备而言是“禁用的”；主要例外是已使用ethtool设置了WoL（wake-on-LAN）功能的电源按钮、键盘和以太网适配器。对于那些不自己生成唤醒请求而仅转发从一个总线到另一个总线的唤醒请求的设备（例如PCI Express端口），也应默认为“启用”。

仅当power.wakeup对象存在且相应的power/wakeup文件包含“启用”字符串时，device\_may\_wakeup（）例程才返回true。这些信息被子系统（如PCI总线类型代码）用于查看是否启用设备的唤醒机制。如果设备唤醒机制由驱动程序直接启用或禁用，他们也应使用device\_may\_wakeup（）来决定在系统睡眠转换期间做什么。但是，在任何情况下，不应该期望设备驱动程序直接调用device\_set\_wakeup\_enable（）。

需要注意的是，系统唤醒在概念上不同于由运行时电源管理使用的“远程唤醒”，尽管它可能由同一物理机制支持。远程唤醒是一项功能，允许处于低功率状态的设备触发特定的中断，以发出信号，说明它们应该被置于全功率状态。这些中断可能会或可能不会用于信号系统唤醒事件，这取决于硬件设计。在某些系统中，不可能从系统睡眠状态触发它们。在任何情况下，对于支持它的所有设备和驱动程序，远程唤醒都应始终启用运行时电源管理。

#### /sys/devices/.../power/control文件

驱动程序模型中的每个设备都有一个标志，用于控制它是否受到运行时电源管理的影响。这个标志runtim\_auto由总线类型（或通常是子系统）代码使用pm\_runtime\_allow（）或pm\_runtime\_forbid（）进行初始化；默认情况下，允许运行时电源管理。

用户空间可以通过将“on”或“auto”写入设备的power/control sysfs文件来调整设置。写入“auto”会调用pm\_runtime\_allow（），设置标志并允许驱动程序对设备进行运行时电源管理。写入“on”会调用pm\_runtime\_forbid（），清除标志，将设备返回到全功率状态（如果它处于低功率状态），并防止设备进行运行时电源管理。用户空间可以通过读取该文件来检查runtime\_auto标志的当前值。

设备的runtime\_auto标志对系统范围的电源转换的处理没有影响。特别是，设备可能（在大多数情况下应该和会）在系统全局转换到睡眠状态期间进入低功率状态，即使其runtime\_auto标志清除。

有关运行时电源管理框架的更多信息，请参阅 I / O 设备的运行时电源管理框架。

### 调用驱动程序以进入和离开系统睡眠状态

当系统进入睡眠状态时，会要求每个设备的驱动程序通过将其置于与目标系统状态兼容的状态来暂停设备。这通常是“关闭”的某个版本，但细节是系统特定的。此外，启用唤醒的设备通常会部分功能性地保持以便唤醒系统。

当系统离开低功率状态时，将要求设备的驱动程序通过将其返回到全功率状态来恢复设备。挂起和恢复操作始终配对进行，而且都是多阶段操作。

对于简单的驱动程序，挂起可能使用类代码消减设备，然后在挂起\_noirq期间将其硬件尽可能地关闭。然后，匹配的恢复调用将完全重新初始化硬件，然后重新激活其类I/O队列。

更具备能源感知的驱动程序可能准备好设备以触发系统唤醒事件。

#### 调用顺序保证

为了确保需要与设备通信的桥和类似链接在设备挂起或恢复时可用，设备层次结构是从下往上遍历以挂起设备。从上到下的顺序用于恢复这些设备。

设备层次结构的顺序由设备注册的顺序定义在其父项之前，子项永远不能注册、探测或恢复；而且在父项之后也无法删除或挂起它。

策略是设备层次结构应该匹配硬件总线拓扑结构。[或者至少控制总线，对于使用多个总线的设备。]特别是，如果设备的父项正在挂起（即已被PM核心选择为下一个要挂起的设备），或者父项已经挂起，并且在所有其他设备都已挂起之后，则设备注册可能会失败。设备驱动程序必须准备应对这种情况。

#### 系统电源管理阶段

暂停或恢复系统需要分几个阶段完成。不同的阶段用于暂停到空闲、浅度（待机）和深度（“挂起到RAM”）睡眠状态以及休眠状态（“挂起到磁盘”）。在下一阶段开始之前，每个阶段都涉及为每个设备执行回调函数。并非所有总线或类别支持所有这些回调函数，也不是所有驱动程序都使用所有回调函数。各个阶段总是在任务被冻结之后和被解冻之前运行。此外， \*\_noirq 阶段在 IRQ 处理程序被禁用的时候运行（除了标记有 IRQF\_NO\_SUSPEND 标志的处理程序）。

所有阶段都使用 PM 域、总线、类型、类别或驱动程序回调函数（即，在 dev->pm\_domain->ops、dev->bus->pm、dev->type->pm、dev->class->pm 或 dev->driver->pm 中定义的方法）。这些回调函数被 PM 核心视为相互排斥的。此外，PM 域回调函数始终优先于所有其他回调函数，例如，类型回调函数优先于总线、类别和驱动程序回调函数。为了准确，以下规则用于确定在给定阶段执行哪个回调函数

1）如果 dev->pm\_domain 存在，则 PM 核心将选择提供的 dev->pm\_domain->ops 回调函数执行。

2）否则，如果 dev->type 和 dev->type->pm 都存在，则将选择提供的 dev->type->pm 回调函数执行。

3）否则，如果 dev->class 和 dev->class->pm 都存在，则将选择提供的 dev->class->pm 回调函数执行。

4）否则，如果 dev->bus 和 dev->bus->pm 都存在，则将选择提供的 dev->bus->pm 回调函数执行。

这使得 PM 域和设备类型可以在必要时覆盖总线类型或设备类别提供的回调函数。

PM 域、类型、类别和总线回调函数又可以启动存储在 dev->driver->pm 中的设备或驱动程序特定方法，但并不必须这样做。

如果执行所选子系统回调方法不存在，则 PM 核心将从 dev->driver->pm 集合中执行相应的方法。

#### 进入系统挂起

当系统进入冻结、待机或睡眠状态时，各个阶段为准备、暂停、暂停晚期、暂停\_noirq。

1）准备阶段旨在通过防止注册新设备来防止竞态；如果随意注册新子设备，PM 核心永远不会知道设备的所有子项是否已经暂停。与此相反，从 PM 核心的角度来看，设备可以随时注销。与其他与挂起有关的阶段不同，在准备阶段期间，按照自上而下的方式遍历设备层次结构。

在 - > prepare 回调方法返回之后，在设备以下不得注册新的子设备。该方法也可以以某种方式为即将到来的系统电源转换准备设备或驱动程序，但不应将设备置于低功耗状态。此外，如果设备支持运行时电源管理，则 - >准备回调方法在必要时不得更新其状态，以便稍后可以从运行时挂起恢复它。

对于支持运行时电源管理的设备，可以使用准备回调的返回来向 PM 核心指示它可以安全地使设备保持在运行时挂起状态（如果已经运行时挂起），前提是设备的所有后代也必须保持在运行时挂起状态。也就是说，如果准备回调返回正数且所有后代设备都返回了正数，且它们（包括设备本身）都处于运行时挂起状态，则 PM 核心将跳过挂起、挂起晚期和挂起\_noirq 阶段以及所有这些设备的后续设备恢复的相应阶段。在这种情况下， - >完成回调将是 - >准备回调之后的下一个被调用，并完全负责适当地将设备置于一致的状态。

请注意，即使设备已禁用运行时 PM，此直接完成程序也适用。因此，如果设备具有系统 - 睡眠回调但不支持运行时 PM，则其准备回调永远不能返回正数。这是因为所有这样的设备最初都设置为运行时挂起，其运行时 PM 已禁用。

此功能还可以通过使用 DPM\_FLAG\_NO\_DIRECT\_COMPLETE 和 DPM\_FLAG\_SMART\_PREPARE 驱动程序电源管理标志来由设备驱动程序控制。[通常，它们在通过将它们传递给涉及的设备的 dev\_pm\_set\_driver\_flags() 助手函数时，在探测设备的同时设置它们。]如果设置了第一个标志，则 PM 核心将不会对给定设备和其任何祖先应用上述直接完成程序。第二个标志在设置时，通知中间层代码（总线类型、设备类型、PM 域、类别），它应考虑由驱动程序提供的 - >准备回调的返回，并且只有在驱动程序的回调已经返回了正数的情况下，才会从自己的 - >准备回调返回正数。

2）->suspend方法应暂停设备以停止其执行I / O。它们还可能保存设备寄存器并将其放入适当的低功耗状态，具体取决于设备所在的总线类型，并且它们可能启用唤醒事件。

然而，对于支持运行时功率管理的设备，子系统（尤其是总线类型和PM领域）提供的->suspend方法必须遵循关于在调用其驱动程序的->suspend方法之前可以对设备进行哪些操作的附加规则。也就是说，如果需要，它们可以通过为它们调用pm\_runtime\_resume（）来从运行时挂起中恢复设备，但它们在那时不得以任何其他方式更新设备的状态（以防驱动程序需要在其->suspend方法中从运行时挂起中恢复设备）。实际上，PM核心通过在发出->prepare回调之前调用pm\_runtime\_get\_noresume（）（并在发出->complete回调后调用pm\_runtime\_put（））来防止子系统或驱动程序在这些时间将设备置于运行时挂起状态。

3）对于许多设备来说，将挂起分为“静止设备”和“保存设备状态”两个阶段是方便的，在这种情况下，suspend\_late意味着后者。它始终在有关设备的运行时电源管理已被禁用后执行。

4）挂起noirq阶段发生在禁用IRQ处理程序之后，这意味着在运行回调方法时将不会调用驱动程序的中断处理程序。->suspend\_noirq方法应保存以前未保存的设备寄存器的值，并最终将设备放入适当的低功耗状态。

大多数子系统和设备驱动程序不需要实现此回调。然而，像PCI这样允许设备共享中断向量的总线类型通常需要它；否则，驱动程序可能会在将自己的设备设置为低功率后通过接收到其他设备生成的共享中断而在挂起阶段遇到错误。

在这些阶段结束时，驱动程序应停止所有I / O事务（DMA、IRQ），保存足够的状态以便他们可以重新初始化或恢复先前的状态（硬件所需），并将设备置于低功耗状态。在许多平台上，他们将关闭一个或多个时钟源；有时他们也会关闭电源或降低电压。[支持运行时PM的驱动程序可能已经执行了这些步骤中的一些或全部。]

如果device\_may\_wakeup（）返回true，则应准备好设备，以便在系统处于睡眠状态时生成硬件唤醒信号以触发系统唤醒事件。例如，enable\_irq\_wake（）可能会识别连接到开关或其他外部硬件的GPIO信号，pci\_enable\_wake（）对PCI PME信号执行类似操作。

如果其中任何回调返回错误，则系统不会进入所需的低功耗状态。相反，PM核心将通过恢复所有已挂起的设备来撤消其操作。

#### 退出系统挂起

当从冻结、待机或内存睡眠中恢复时，阶段为resume\_noirq、resume\_early、resume、complete。

1）->resume\_noirq回调方法应在调用驱动程序的中断处理程序之前执行任何需要的操作。这通常意味着撤消suspend\_noirq阶段的操作。如果总线类型允许设备共享中断向量，例如PCI，则该方法应将设备及其驱动程序置于状态，以便驱动程序可以识别设备是否是传入中断的源，如果有的话，则正确地处理它们。

例如，PCI总线类型的->pm.resume\_noirq（）将设备置于完全电源状态（PCI术语中的D0）并恢复设备的标准配置寄存器。然后，它调用设备驱动程序的->pm.resume\_noirq（）方法执行特定于设备的操作。

2）->resume\_early方法应为执行resume方法准备好设备。这通常涉及撤消前面的suspend\_late阶段的操作。

3）->resume方法应将设备恢复到其操作状态，以便它可以执行正常的I / O操作。这通常涉及撤消挂起阶段的操作

4）最后一步应撤消准备阶段的操作。因此，与其他恢复相关的阶段不同，完成阶段在从下到上遍历设备层次结构时进行。

但是，请注意，当->resume回调发生时，可能会在设备下注册新的子级；不必等到完成阶段运行。

此外，如果前面的->prepare回调返回正数，则设备可能已经在整个系统挂起和恢复期间保持在运行时挂起状态中（它们的->suspend，->suspend\_late，->suspend\_noirq，->resume\_noirq，->resume\_early和->resume回调可能已被跳过）。在这种情况下，->complete回调将完全负责在系统挂起之后，如有必要，将设备置于一致状态。[例如，它可能需要为此对设备排队运行时恢复请求。]要检查是否存在这种情况，->complete回调可以查阅设备的power.direct\_complete标志。如果在运行->complete回调时设置了该标志，则使用了直接完成机制，并且可能需要采取特殊操作才能使设备正常工作。

在这些阶段结束时，驱动程序应该与暂停之前一样正常运行可以使用DMA和IRQ执行I / O操作，并且相关时钟已关闭。

但是，这里的细节可能再次是特定于平台的。例如，某些系统支持多个“运行”状态，恢复结束时的模式可能不是之前的模式。这意味着某些时钟或电源供应的可用性已更改，这可能会影响驱动程序的工作方式。

驱动程序需要能够处理硬件，因为所有暂停方法都被调用后已被重置，例如通过完全重新初始化。这可能是最困难的部分，并且最受NDA文件和芯片勘误保护。如果硬件状态自暂停以来没有改变，那将是最简单的，但是只有当目标系统休眠转换为挂起到空闲时才能保证。对于其他系统休眠状态，情况可能不是这种情况（对于ACPI定义的系统休眠状态（例如S3），通常不是这种情况）。

驱动程序还必须准备好在系统断电时注意到设备已经被移除，如果物理上可能。 PCMCIA，MMC，USB，Firewire，SCSI甚至IDE是常见的Linux平台看到这种移除的例子。如何通知和处理这些移除的详细信息当前是特定于总线的，并且通常涉及单独的线程。

这些回调可能会返回错误值，但PM核心会忽略此类错误，因为除了将它们打印到系统日志中外，它无法做其他事情。

#### 进入休眠状态

休眠系统比将其放入睡眠状态更加复杂，因为它涉及创建和保存系统映像。因此，休眠需要更多的阶段，并具有不同的回调集。这些阶段始终在任务被冻结并且已经释放了足够的内存后运行。

休眠的一般流程是消除所有设备（“冻结”），在一切稳定的情况下创建系统内存映像，重新激活所有设备（“解冻”），将映像写入永久存储并最终关闭系统（“关机”）。用于完成此操作的阶段是准备，冻结，冻结晚期，冻结noirq，解冻noirq，解冻早期，解冻，完成，准备，关闭电源，关闭电源晚期，关闭电源noirq。

1. 准备阶段在上面的“进入系统挂起”部分中讨论。
2. ->freeze方法应使设备处于静止状态，以便不生成IRQ或DMA，并且可能需要保存设备寄存器的值。但是，不必将设备放在低功耗状态中，并且为节省时间最好不要这样做。此外，设备不应准备生成唤醒事件。
3. 冻结晚期阶段类似于早期说明的挂起晚期阶段，但是设备不应处于低功耗状态，并且不应允许生成唤醒事件。
4. 冻结noirq阶段类似于早期讨论的挂起noirq阶段，但是再次，设备不应处于低功耗状态，并且不应允许生成唤醒事件。

此时创建系统映像。在此期间，所有设备都应处于非活动状态，而内存内容应保持不变，以便该图像形成系统状态的原子快照。

1. 解冻noirq阶段类似于先前讨论的恢复noirq阶段。主要区别在于，它的方法可以假定设备处于与冻结noirq阶段结束时相同的状态。
2. 解冻早期阶段类似于上文说明的恢复早期阶段。如果必要，其方法应撤消先前的冻结晚期的操作。
3. 解冻阶段类似于早期讨论的恢复阶段。其方法应将设备恢复到操作状态，以便必要时可以用于保存映像。
4. 完成阶段在上面的“离开系统挂起”部分中讨论。

此时保存系统映像，然后需要为即将到来的系统关闭设备。这与在将系统置于挂起到空闲，浅度或深度睡眠状态之前暂停它们非常相似，而且阶段也类似。

1. 准备阶段在上面讨论过。
2. 关闭电源阶段类似于挂起阶段。
3. 关闭电源晚期阶段类似于挂起晚期阶段。
4. 关闭电源noirq阶段类似于挂起noirq阶段。

->poweroff，->poweroff\_late和->poweroff\_noirq回调应分别执行与->suspend，->suspend\_late和->suspend\_noirq回调基本相同的操作。值得注意的区别是它们不需要存储设备寄存器值，因为在冻结，冻结晚期或冻结noirq阶段时应该已经存储了寄存器。此外，在许多机器上，固件将关闭整个系统，因此回调无需将设备放入低功耗状态。

#### 离开休眠状态

从休眠状态恢复比从保存了主存内容的睡眠状态恢复更加复杂，因为它需要载入系统映像到内存并在将控制权传回映像内核之前还原休眠前的记忆内容。

尽管原则上系统映像可以由引导加载程序加载到内存并还原休眠前的记忆内容，但实际上这是不可能的，因为引导加载程序不够智能，也没有确立的传递必要信息的协议。因此，引导加载程序将一份新的内核实例，称为“还原内核”，加载到内存中，并以通常的方式将控制权传递给它。然后还原内核读取系统映像，还原休眠前的记忆内容，并将控制权传递给映像内核。因此，在恢复休眠状态时涉及两个不同的内核实例。实际上，还原内核可能完全不同于映像内核配置不同，版本甚至不同。这对于设备驱动程序及其子系统有重要影响。

为了能够将系统映像加载到内存中，还原内核需要包含至少一部分设备驱动程序的子集，以允许它访问包含映像的存储介质，尽管它不需要包含映像内核中的所有驱动程序。加载映像后，由引导内核管理的设备需要准备将控制权传回映像内核。这与创建系统映像所涉及的初始步骤非常相似，可以使用prepare、freeze和freeze\_noirq阶段完成。然而，这些阶段受到还原内核中的驱动程序控制的设备只有这些。其他设备仍然处于引导加载程序离开时的状态。

如果还原休眠前的记忆内容失败，还原内核将经历上面说明的“解冻”过程，使用thaw\_noirq、thaw\_early、thaw和complete阶段，然后继续正常运行。这种情况很少发生。最常见的情况是成功还原休眠前的记忆内容，并将控制权传递给映像内核，然后映像内核将负责将系统恢复为工作状态。

为了实现这一点，映像内核必须还原设备的休眠前功能。操作类似于从睡眠状态唤醒（保存内存内容），尽管涉及不同的阶段restore\_noirq、restore\_early、restore和complete。

1. restore\_noirq阶段类似于resume\_noirq阶段。
2. restore\_early阶段类似于resume\_early阶段。
3. 还原阶段类似于恢复阶段。
4. 完整阶段在上面讨论过。

与resume[\_early|\_noirq]的主要区别是restore[\_early|\_noirq]必须假设设备已被引导加载程序或还原内核访问和重新配置。因此，设备的状态可能与从freeze、freeze\_late和freeze\_noirq阶段记忆的状态不同。设备甚至可能需要重置并完全重新初始化。在许多情况下，这种差异并不重要，因此->resume[\_early|\_noirq]和->restore[\_early|\_norq]方法指针可以设置为相同的例程。然而，在确实有情况下需要使用不同的回调指针。

### 电源管理通知器

有些操作不能通过上面讨论的电源管理回调函数执行，因为回调函数发生得太晚或太早。为了处理这些情况，子系统和设备驱动程序可以注册电源管理通知器，这些通知器在任务被冻结之前和它们被解冻后调用。一般来说，PM通知器适合执行需要用户空间可用或至少不会妨碍用户空间的操作。

有关详细信息，请参阅休眠/休眠通知器。

### 设备低功耗（挂起）状态

设备低功耗状态不是标准的。一个设备可能只处理“开”和“关”，而另一个设备可能支持十几个不同版本的“开”（有多少个引擎是活动的？），再加上一个从完全“关”回到“开”的状态。

某些总线定义了有关不同挂起状态表示的规则。PCI提供了一个示例在挂起序列完成之后，非传统PCI设备可能不会执行DMA或发出IRQ，并且它发出的任何唤醒事件都将通过PME#总线信号发出。此外，有几种PCI标准设备状态，其中一些是可选的。

相比之下，集成的系统级片上处理器通常使用IRQ作为唤醒事件源（因此驱动程序会调用enable\_irq\_wake（）），并且可能能够将DMA完成视为唤醒事件（有时DMA也可能保持活动状态，仅CPU和某些外围设备会休眠）。

这里的一些细节可能是特定于平台的。系统可能有设备可以在某些睡眠状态下完全处于活动状态，例如使用 DMA 刷新的 LCD 显示器，而大部分系统处于轻度睡眠状态……同时，它的帧缓冲区甚至可以由 DSP 或其他非 Linux CPU 更新，而 Linux 控制处理器保持空闲。

此外，具体采取的操作可能取决于目标系统状态。一个目标系统状态可以让给定设备非常操作；另一个可能需要硬关机并在恢复时重新初始化。两个不同的目标系统可能会以不同的方式使用相同的设备；前面提到的 LCD 可能在一个产品的“待机”状态下处于活动状态，但使用相同的 SOC 的不同产品可能会有所不同。

### 设备电源管理域

有时设备共享参考时钟或其他功率资源。在这些情况下，通常无法单独将设备置于低功率状态。相反，共享电源资源的设备集合可以同时被置于低功率状态，通过关闭共享的功率资源。当然，它们也需要一起被置于完全电力状态，通过打开共享的功率资源。具有这种属性的设备集合通常被称为电源域。电源域也可能嵌套在另一个电源域中。嵌套域被称为父域的子域。

对设备电源管理域的支持通过struct device 的 pm\_domain 字段提供。该字段是 struct dev\_pm\_domain 类型的对象指针，该类型定义在 include/linux/pm.h 中，提供一组电源管理回调，类似于在所有电源转换期间为给定设备执行的子系统级别和设备驱动程序回调，而不是相应的子系统级别回调。具体而言，如果设备的 pm\_domain 指针不为 NULL，则将执行来自其指向的对象的 suspend() 回调，而不是其子系统（例如总线类型）的 suspend() 回调和所有剩余的回调类似。换句话说，如果为给定设备定义了电源管理域回调，则总是优先于由设备子系统（例如总线类型）提供的回调。

设备电源管理域的支持仅适用于需要在许多不同的功率域配置中使用相同的设备驱动程序电源管理回调并希望避免将功率域的支持纳入子系统级别回调的平台，例如通过修改平台总线类型。其它平台不需要实现它或以任何方式考虑它。

设备可以被定义为 IRQ 安全，这向 PM 核心指示其运行时 PM 回调可以在禁用中断时调用（有关更多信息，请参见用于 I/O 设备的运行时电源管理框架）。如果 IRQ 安全设备属于 PM 域，则将禁止该域的运行时 PM，除非该域本身被定义为 IRQ 安全。但是，仅当其中所有设备均为 IRQ 安全设备时，定义 PM 域为 IRQ 安全才有意义。此外，如果 IRQ 安全域具有父域，则仅当该父域本身也为 IRQ 安全且所有 IRQ 安全父域的子域都必须是 IRQ 安全时，才允许父域的运行时 PM。

### 运行时电源管理

许多设备能够在系统仍在运行时动态关闭电源。这个特性对于未被使用的设备非常有用，在运行的系统上可以节省大量电力。这些设备通常支持一系列运行时电源状态，这些状态可能使用名称，例如“关闭”，“休眠”，“空闲”，“活动”等。这些状态在某些情况下（如 PCI）将部分受到设备使用的总线的限制，并且通常包括硬件状态，这些状态也用于系统睡眠状态。

可以在某些设备由于运行时电源管理而处于低功率状态时启动全系统的电源转换。系统睡眠 PM 回调应该识别这种情况并适当地对其进行反应，但必要的操作是子系统特定的。

在某些情况下，决策可能在子系统级别上进行，而在其他情况下，可能留给设备驱动程序来决定。在某些情况下，希望在系统范围的电源转换期间将挂起的设备留在该状态下，但在其他情况下，必须短暂地将设备放回完全电力状态，例如，以便可以禁用其系统唤醒功能。这完全取决于硬件和有关子系统和设备驱动程序的设计。

如果需要在系统进入睡眠状态的过渡期间从运行时挂起中恢复设备，则可以在设备的驱动程序或其子系统（例如总线类型或PM域）的->suspend回调（或冻结或poweroff与休眠相关的回调）中调用pm\_runtime\_resume()。然而，在调用设备驱动程序的->suspend回调（或等效的回调）之前，子系统必须不改变设备的运行时状态从它们的->prepare和->suspend回调（或等效的回调）中除外。

一些总线类型和 PM 域有一个策略，可以在它们的回调中预先从运行时挂起恢复所有设备->suspend，但如果设备的驱动程序可以处理运行时挂起的设备，那么这可能并不是真正必要的。驱动程序可以通过在探测时间设置，将其传递给 DPM\_FLAG\_SMART\_SUSPEND帮助 程序来指示这一点。如果设备在系统范围挂起 阶段开始时（或在power.driver\_flagsdev\_pm\_set\_driver\_flags()->suspend\_late->suspend\_noirqsuspend\_latepoweroff\_late休眠阶段），当运行时 PM 已被禁用时，假设它的状态在该点之后不应该改变，直到系统范围的转换结束（PM 核心本身为“noirq”，“late ”和“早期”系统范围的 PM 回调直接由它执行）。如果发生这种情况，驱动程序的系统范围恢复回调（如果存在）可能仍会在随后的系统范围恢复转换期间被调用，并且设备的运行时电源管理状态可能会在为其启用运行时 PM 之前设置为“活动”，因此驱动程序必须准备好应对其系统范围内的恢复回调的调用->runtime\_suspend（没有干预->runtime\_resume等），并且设备的最终状态必须反映“活动”运行时 PM 状态案件。

在系统范围内从睡眠状态恢复期间，最容易将设备置于全功率状态，如 中所述Documentation/power/runtime\_pm.txt。[有关此特定问题的更多信息以及有关设备运行时电源管理框架的一般信息，请参阅该文档。]

然而，通常希望在系统转换到工作状态后让设备处于挂起状态，特别是如果这些设备在之前的系统范围挂起（或类似）转换之前已经处于运行时挂起状态。设备驱动程序可以使用DPM\_FLAG\_LEAVE\_SUSPENDED标志以向 PM 核心（和中间层代码）表明他们更愿意让他们处理的特定设备保持挂起状态，因此他们可以跳过他们的系统范围的恢复回调没有问题。设备是否实际上将处于挂起状态可能取决于它们在给定系统挂起-恢复周期之前的状态以及正在进行的系统转换的类型。特别是，如果该转换是从休眠状态恢复，则设备不会保持挂起状态，因为在这种情况下，设备状态不能保证由存储在休眠图像中的信息反映出来。

设备处理中涉及的中间层代码预计会向 PM 内核指示设备是否可以通过设置其状态位而处于挂起状态， power.may\_skip\_resumePM 内核会在前面系统的“noirq”阶段检查该状态位-wide 挂起（或类似）过渡。然后中间层负责在其“noirq”恢复回调中适当地处理设备，无论设备是否挂起都会执行，但其他恢复回调（除了 ）将由 PM 自动->complete跳过如果设备真的可以处于挂起状态，则为核心。

DPM\_FLAG\_LEAVE\_SUSPENDED对于“noirq”、“late”和“early”驱动程序回调由 PM 核心直接调用的设备，如果已设置并且设备在suspend\_noirq（或类似的）期间处于运行时挂起状态，则将跳过所有系统范围的恢复回调 阶段或正在进行的转换是适当的系统挂起（而不是与休眠相关的任何事情）并且设备的唤醒设置适用于运行时 PM（即，它根本不能生成唤醒信号，或者它被允许从唤醒系统睡觉）。

## 挂起/休眠通知程序

版权© 2016 Intel Corporation

存在一些子系统或驱动程序可能需要在休眠/挂起或恢复/恢复之前执行一些操作，但它们要求系统完全正常运行，因此驱动程序和子系统的->suspend()和->resume()甚至->prepare()和->complete()回调不能用于此目的。

例如，设备驱动程序可能希望在恢复/恢复后向其设备上传固件，但无法通过从它们的->resume()或->complete()回调例程调用request\_firmware()来执行此操作（在这些点上，用户空间进程已被冻结）。解决方案可能是在进程被冻结之前将固件加载到内存中，并从->resume()例程中上传它。可以使用挂起/休眠通知程序进行此操作。

具有此类需求的子系统或驱动程序可以注册在以下事件上由PM核心调用的挂起通知程序

PM\_HIBERNATION\_PREPARE

系统将要休眠，任务将立即被冻结。这与下面的PM\_SUSPEND\_PREPARE不同，因为在这种情况下，在通知程序和“冻结”转换的PM回调被调用之间执行其他工作。

PM\_POST\_HIBERNATION

系统内存状态已从休眠映像或休眠期间发生错误中恢复。设备恢复回调已执行并已解冻任务。

PM\_RESTORE\_PREPARE

系统将要恢复睡眠映像。如果一切顺利，恢复映像内核将发出PM\_POST\_HIBERNATION通知。

PM\_POST\_RESTORE

恢复休眠时发生错误。设备恢复回调已执行并已解冻任务。

PM\_SUSPEND\_PREPARE

系统正在为挂起做准备。

PM\_POST\_SUSPEND

系统刚刚恢复或挂起时发生错误。设备恢复回调已执行并已解冻任务。

一般假定挂起通知程序对于PM\_HIBERNATION\_PREPARE所做的任何操作，在PM\_POST\_HIBERNATION时都应撤消。类似地，对于PM\_SUSPEND\_PREPARE执行的操作应为PM\_POST\_SUSPEND保留反向操作。

此外，如果挂起通知程序之一在PM\_HIBERNATION\_PREPARE或PM\_SUSPEND\_PREPARE事件中失败，则已成功执行该事件的通知程序将分别在PM\_POST\_HIBERNATION或PM\_POST\_SUSPEND中调用。

挂起和休眠通知程序在持有pm\_mutex时被调用。它们按照通常的方式定义，但它们的最后一个参数没有意义（它始终为NULL）。

要注册和/或注销挂起通知程序，请分别使用register\_pm\_notifier()和unregister\_pm\_notifier()（都在include/linux/suspend.h中定义）。如果您不需要注销通知程序，则还可以使用在include/linux/suspend.h中定义的pm\_notifier()宏。

## 设备电源管理数据类型

### struct dev\_pm\_ops

设备PM回调。

**定义**

struct dev\_pm\_ops {

int (\*prepare)(struct device \*dev);

void (\*complete)(struct device \*dev);

int (\*suspend)(struct device \*dev);

int (\*resume)(struct device \*dev);

int (\*freeze)(struct device \*dev);

int (\*thaw)(struct device \*dev);

int (\*poweroff)(struct device \*dev);

int (\*restore)(struct device \*dev);

int (\*suspend\_late)(struct device \*dev);

int (\*resume\_early)(struct device \*dev);

int (\*freeze\_late)(struct device \*dev);

int (\*thaw\_early)(struct device \*dev);

int (\*poweroff\_late)(struct device \*dev);

int (\*restore\_early)(struct device \*dev);

int (\*suspend\_noirq)(struct device \*dev);

int (\*resume\_noirq)(struct device \*dev);

int (\*freeze\_noirq)(struct device \*dev);

int (\*thaw\_noirq)(struct device \*dev);

int (\*poweroff\_noirq)(struct device \*dev);

int (\*restore\_noirq)(struct device \*dev);

int (\*runtime\_suspend)(struct device \*dev);

int (\*runtime\_resume)(struct device \*dev);

int (\*runtime\_idle)(struct device \*dev);

};

**成员**

prepare

此回调的主要作用是防止设备的新子设备在返回后被注册（驱动程序的子系统和通常内核的其余部分应该防止在 prepare() 之后再次调用探测方法已成功）。如果prepare()检测到它无法处理的情况（例如，已经在进行中的孩子的注册），它可能会返回 -EAGAIN，以便 PM 核心可以再次执行它（例如，在注册新的孩子之后）以从中恢复比赛条件。此方法针对所有类型的挂起转换执行，后跟挂起回调之一：suspend()、freeze()或 poweroff(). 如果转换是挂起到内存或待机（即，与休眠无关），则prepare()的返回值可用于指示 PM 核心让设备在运行时挂起（如果适用）。也就是说，如果prepare()返回一个正数，PM 核心将理解为声明该设备似乎是运行时暂停的，并且它可能在整个转换期间和随后的恢复期间保持该状态，如果它的所有后代也处于运行时暂停状态。如果发生这种情况，complete()将在prepare()之后直接执行，并且它必须确保系统恢复后设备的正常运行。PM核心执行子系统级prepare()在开始为其中任何设备调用挂起回调之前为所有设备，因此通常可以假定设备可以正常运行或在执行prepare ()时响应运行时恢复请求。但是，设备驱动程序此时可能不会假设任何有关用户空间可用性的信息，并且从prepare()中请求固件是无效的 （这样做已经太晚了）。在 GFP\_KERNEL 模式下从prepare()分配大量内存也是无效的。[为了解决这些限制，驱动程序可以注册挂起和休眠通知程序，以便在任务冻结之前执行。]

complete

撤消prepare()所做的更改。此方法针对所有类型的恢复转换执行，遵循恢复回调之一： resume()、thaw()、restore()。如果状态转换在驱动程序的挂起回调之前失败，也会调用：可以执行suspend()、freeze()或 poweroff()（例如，如果 PM 核心未能成功挂起的其他设备之一的挂起回调失败较早）。PM 核心在为所有设备执行了适当的恢复回调后执行子系统级complete() 。如果相应的 prepare()在挂起转换开始时返回一个正数并且设备处于运行时挂起状态（不为其执行任何挂起和恢复回调），complete()将是恢复期间为设备执行的唯一回调。在这种情况下， complete()必须准备好做任何必要的事情来确保系统恢复后设备的正常运行。为此， complete()可以检查设备的 power.direct\_complete 标志，以了解是否 (unset) 或 (set) 之前的挂起和恢复回调已为其执行。

suspend

在将系统置于睡眠状态之前执行，在该状态下主存储器的内容得以保存。要执行的确切操作取决于设备的子系统（PM 域、设备类型、类或总线类型），但通常设备必须在子系统级 suspend() 返回后处于静止状态，这样它就不会执行任何 I / O 或 DMA。在为所有设备调用子系统级prepare()之后，对所有设备执行子系统级suspend() 。

resume

在将系统从保存主存储器内容的睡眠状态唤醒后执行。执行的确切操作取决于设备的子系统，但通常希望驱动程序再次开始工作，响应硬件事件和软件请求（设备本身可能处于低功耗状态，等待运行时恢复发生). 设备在其驱动程序的resume()回调运行时的状态取决于设备所属的平台和子系统。在大多数平台上，在resume()期间对时钟等资源的可用性没有限制。在为所有设备调用子系统级resume\_noirq ( )之后，为所有设备执行子系统级 resume()。

freeze

特定于休眠，在创建休眠图像之前执行。类似于suspend()，但它不应使设备能够发出唤醒事件信号或更改其电源状态。大多数子系统（PCI 总线类型明显例外）都希望驱动程序级 freeze()将设备设置保存在内存中，以供restore() 在随后从休眠中恢复期间使用。在为所有设备调用子系统级prepare()之后，对所有设备执行子系统级freeze() 。

thaw

特定于休眠，在创建休眠图像后执行，或者在图像创建失败时执行。也在尝试从此类图像恢复主内存内容失败后执行。撤消前面的freeze()所做的更改，因此可以像调用freeze()之前一样操作设备。在为所有设备调用子系统级thaw\_noirq()之后，对所有设备执行子系统级thaw() 。如果发生转换错误，它也可以在freeze()之后直接执行。

poweroff

特定于休眠，在保存休眠图像后执行。类似于suspend()，但它不需要将设备的设置保存在内存中。在为所有设备调用子系统级prepare()之后，对所有设备执行子系统级poweroff() 。

restore

特定于休眠，在从休眠图像恢复主内存的内容后执行，类似于resume()。

suspend\_late

继续由suspend()启动的操作。对于许多设备，suspend\_late()可能指向与运行时挂起回调相同的回调例程。

resume\_early

准备执行resume()。对于许多设备， resume\_early()可能指向与运行时恢复回调相同的回调例程。

freeze\_late

继续由freeze()启动的操作。类似于 suspend\_late()，但它不应使设备能够发出唤醒事件信号或更改其电源状态。

thaw\_early

准备执行thaw()。撤消前面的freeze\_late()所做的更改。

poweroff\_late

继续由poweroff()启动的操作。类似于 suspend\_late()，但它不需要将设备的设置保存在内存中。

restore\_early

准备执行restore()，类似于resume\_early()。

suspend\_noirq

完成由suspend()启动的操作。执行挂起可能与其驱动程序的中断处理程序竞争的设备所需的任何其他操作，保证在执行suspend\_noirq()时不会运行。通常期望在子系统级suspend\_noirq()成功返回后，设备将处于低功耗状态（适用于目标系统睡眠状态） 。如果设备可以生成系统唤醒信号并启用唤醒系统，则应在那时配置它。但是，取决于平台和设备的子系统，suspend()或suspend\_late()可以允许将设备置于低功耗状态并将其配置为生成唤醒信号，在这种情况下通常不需要定义 suspend\_noirq()。

resume\_noirq

通过执行恢复可能与其驱动程序的中断处理程序竞争的设备所需的任何操作来准备 resume() 的执行，保证在 执行resume\_noirq()时不会运行。

freeze\_noirq

完成由freeze()启动的操作。执行冻结可能与其驱动程序的中断处理程序竞争的设备所需的任何其他操作，保证在执行freeze\_noirq()时不会运行。设备的电源状态不应被freeze()或freeze\_late()或freeze\_noirq()更改，并且不应将其配置为通过任何这些回调发出系统唤醒信号。

thaw\_noirq

通过执行解冻可能与其驱动程序的中断处理程序竞争的设备所需的任何操作来准备 thaw() 的执行，保证在 执行thaw\_noirq()时不会运行。

poweroff\_noirq

完成由poweroff()启动的操作。类似于 suspend\_noirq()，但它不需要将设备的设置保存在内存中。

restore\_noirq

通过执行解冻可能与其驱动程序的中断处理程序竞争的设备所需的任何操作来准备执行 restore()，保证在 执行restore\_noirq()时不会运行。类似于resume\_noirq()。

runtime\_suspend

准备设备以应对由于电源管理而无法与 CPU 和 RAM 通信的情况。这不一定意味着设备应该进入低功耗状态。例如，如果设备位于即将关闭的链路后面，则该设备可能会保持全功率。如果设备确实进入低功耗状态并且能够生成运行时唤醒事件，则应为其启用远程唤醒（即允许设备通过中断请求更改其电源状态的硬件机制）。

runtime\_resume

将设备置于完全活动状态以响应硬件生成的唤醒事件或软件的请求。如有必要，将设备置​​于全功率状态并恢复其寄存器，以便其完全运行。

runtime\_idle

设备似乎处于非活动状态，如果满足所有必要条件，它可能会进入低功耗状态。检查这些条件，如果适合让 PM 核心排队设备的挂起请求，则返回 0。

**说明**

几个设备电源状态转换对外部可见，影响挂起的I/O队列的状态和（对于触摸硬件的驱动程序）中断、唤醒、DMA和其他硬件状态。也可能存在对各种低功率模式的内部转换，对驱动程序堆栈的其余部分是透明的（例如，将不处于活动使用中的时钟隔离的驱动程序）。

上述外部可见的转换是借助此结构中包含的回调来处理的，通常涉及两个级别的回调。首先，PM核心执行PM域、设备类型、类和总线类型提供的回调。它们是预期执行设备驱动程序提供的回调的子系统级回调，尽管它们可以选择不这样做。如果执行驱动程序回调，则它们必须与子系统级回调协作，以实现适合给定系统转换、给定转换阶段和设备所属的子系统的目标。

除complete()外的所有上述回调都会返回错误代码。但是，resume()、thaw()、restore()、resume\_noirq()、thaw\_noirq()和restore\_noirq()返回的错误代码不会导致PM核心在返回它们的过渡期间中止恢复。在这些情况下返回的错误代码仅用于调试目的打印到系统日志中。仍然建议驱动程序仅在出现无法恢复的故障（即，在处理的设备拒绝恢复并变得无法使用时）时从其恢复方法返回错误代码，以便将来可以修改PM核心，以避免尝试处理无法恢复的设备及其子项。

在上述回调被执行时，允许注销设备。但是，回调例程不得尝试注销它被调用的设备，尽管它可以注销该设备的子项（例如，如果它检测到在系统处于休眠状态时插拔了一个子项）。还有与设备的运行时电源管理相关的回调。同样，通常情况下，PM核心为子系统（PM域、设备类型、类和总线类型）执行这些回调，并且预计子系统级回调会调用驱动程序回调。此外，设备驱动程序的回调要执行的确切操作通常取决于设备所属的平台和子系统。

有关设备运行时电源管理中runtime\_suspend（），runtime\_resume（）和runtime\_idle（）回调的角色，请参阅I / O设备的Runtime Power Management Framework。

### struct dev\_pm\_domain

电源管理域表示。

**定义**

struct dev\_pm\_domain {

struct dev\_pm\_ops ops;

void (\*detach)(struct device \*dev, bool power\_off);

int (\*activate)(struct device \*dev);

void (\*sync)(struct device \*dev);

void (\*dismiss)(struct device \*dev);

};

**成员**

ops

与此域关联的电源管理操作。

detach

从域中删除设备时调用。

activate

在执行总线类型和驱动程序的探测例程之前调用。

sync

在驱动程序探测成功后调用。

dismiss

在驱动程序探测不成功和驱动程序删除后调用。

**说明**

电源域提供回调，在系统暂停，休眠，系统恢复以及在运行时PM转换期间执行，而不是子系统级别和驱动程序级别的回调。

# 通用时钟框架

本文试图解释通用时钟框架的细节以及如何将平台移植到此框架上。它还没有包括include / linux / clk.h中的时钟api的详细说明，但或许有朝一日它会包括该信息。

## 介绍和接口分离

通用时钟框架是一个接口，用于控制当今各种设备上可用的时钟节点。这可以采用时钟门控，速率调整，复用或其他操作的形式。此框架使用CONFIG\_COMMON\_CLK选项启用。

接口本身分为两半，每半都与其相对应的细节受到保护。首先是定义统一框架级会计和基础设施的struct clk的共同定义，这一点在各种平台上一直是重复的。第二个是在drivers / clk / clk.c中定义的clk.h api的一个共同实现。最后是struct clk\_ops，它的操作由clk api实现调用。

接口的第二半由在struct clk\_ops中注册的与硬件特定回调和模拟特定时钟所需的相应硬件特定结构组成。在本文余下部分中，任何对struct clk\_ops中的回调的引用，例如.enable或.set\_rate，都意味着该代码的硬件特定实现。同样，对struct clk\_foo的引用作为一种方便的简写，表示用于假设“ foo”硬件的硬件专用位的实现。

struct clk\_hw将这个接口的两个部分绑定在一起，在struct clk\_core中定义并指向。这允许在通用时钟接口的两个离散半部分之间进行轻松导航。

## 常见数据结构和api

以下是drivers / clk / clk.c中的通用struct clk\_core定义，为了简洁起见进行了修改:

struct clk\_core {

const char \*name;

const struct clk\_ops \*ops;

struct clk\_hw \*hw;

struct module \*owner;

struct clk\_core \*parent;

const char \*\*parent\_names;

struct clk\_core \*\*parents;

u8 num\_parents;

u8 new\_parent\_index;

...

};

以上成员构成时钟树拓扑结构的核心。时钟api本身定义了几个面向驱动程序的函数，它们对struct clk进行操作。该api在include / linux / clk.h中有文档。

使用通用结构clk\_core的平台和设备使用struct clk\_core中的struct clk\_ops指针执行clk-provider.h中定义的硬件特定操作:

struct clk\_ops {

int (\*prepare)(struct clk\_hw \*hw);

void (\*unprepare)(struct clk\_hw \*hw);

int (\*is\_prepared)(struct clk\_hw \*hw);

void (\*unprepare\_unused)(struct clk\_hw \*hw);

int (\*enable)(struct clk\_hw \*hw);

void (\*disable)(struct clk\_hw \*hw);

int (\*is\_enabled)(struct clk\_hw \*hw);

void (\*disable\_unused)(struct clk\_hw \*hw);

unsigned long (\*recalc\_rate)(struct clk\_hw \*hw,

unsigned long parent\_rate);

long (\*round\_rate)(struct clk\_hw \*hw,

unsigned long rate,

unsigned long \*parent\_rate);

int (\*determine\_rate)(struct clk\_hw \*hw,

struct clk\_rate\_request \*req);

int (\*set\_parent)(struct clk\_hw \*hw, u8 index);

u8 (\*get\_parent)(struct clk\_hw \*hw);

int (\*set\_rate)(struct clk\_hw \*hw,

unsigned long rate,

unsigned long parent\_rate);

int (\*set\_rate\_and\_parent)(struct clk\_hw \*hw,

unsigned long rate,

unsigned long parent\_rate,

u8 index);

unsigned long (\*recalc\_accuracy)(struct clk\_hw \*hw,

unsigned long parent\_accuracy);

int (\*get\_phase)(struct clk\_hw \*hw);

int (\*set\_phase)(struct clk\_hw \*hw, int degrees);

void (\*init)(struct clk\_hw \*hw);

void (\*debug\_init)(struct clk\_hw \*hw,

struct dentry \*dentry);

};

## 硬件时钟实现

通用struct clk\_core的优势在于它的.ops和.hw指针可以将struct clk的细节与硬件特定位抽象出来，反之亦然。为了说明这一点，考虑一下在drivers / clk / clk-gate.c中的简单可门控clk实现:

struct clk\_gate {

struct clk\_hw hw;

void \_\_iomem \*reg;

u8 bit\_idx;

...

};

struct clk\_gate包含struct clk\_hw hw以及有关哪个寄存器和位控制此clk的门控的硬件特定知识。这里不需要图像拓扑或会计信息，例如enable\_count或notifier\_count。所有这些都由常见框架代码和struct clk\_core处理。

让我们通过驱动程序代码启用此时钟:

struct clk \*clk;

clk = clk\_get(NULL, "my\_gateable\_clk");

clk\_prepare(clk);

clk\_enable(clk);

clk\_enable的调用图非常简单:

clk\_enable(clk);

clk->ops->enable(clk->hw);

[resolves to...]

clk\_gate\_enable(hw);

[resolves struct clk gate with to\_clk\_gate(hw)]

clk\_gate\_set\_bit(gate);

而clk\_gate\_set\_bit的定义为:

static void clk\_gate\_set\_bit(struct clk\_gate \*gate)

{

u32 reg;

reg = \_\_raw\_readl(gate->reg);

reg |= BIT(gate->bit\_idx);

writel(reg, gate->reg);

}

请注意，to\_clk\_gate定义为:

#define to\_clk\_gate(\_hw) container\_of(\_hw, struct clk\_gate, hw)

这种抽象的模式用于每个时钟硬件表示。

## 支持您自己的时钟硬件

在实现对新型时钟的支持时，唯一需要做的是包含以下标头:

#include <linux/clk-provider.h>

为了为您的平台构建时钟硬件结构，您必须定义以下内容:

struct clk\_foo {

struct clk\_hw hw;

... hardware specific data goes here ...

};

要利用您的数据，您需要支持时钟的有效操作:

struct clk\_ops clk\_foo\_ops = {

.enable = &clk\_foo\_enable,

.disable = &clk\_foo\_disable,

};

使用container\_of实现以上函数:

#define to\_clk\_foo(\_hw) container\_of(\_hw, struct clk\_foo, hw)

int clk\_foo\_enable(struct clk\_hw \*hw)

{

struct clk\_foo \*foo;

foo = to\_clk\_foo(hw);

... perform magic on foo ...

return 0;

};

下面是一个矩阵，根据时钟的硬件能力详细说明了哪些clk\_ops是必需的。标记为“y”的单元格表示必须，标记为“n”的单元格表示要么包括该回调无效，要么是没有必要的。空单元格是可选的或必须根据情况进行评估。

时钟硬件特征

|  | gate | change rate | single parent | multiplexer | root |
| --- | --- | --- | --- | --- | --- |
| .prepare |  |  |  |  |  |
| .unprepare |  |  |  |  |  |
|  |  |  |  |  |  |
| .enable | y |  |  |  |  |
| .disable | y |  |  |  |  |
| .is\_enabled | y |  |  |  |  |
|  |  |  |  |  |  |
| .recalc\_rate |  | y |  |  |  |
| .round\_rate |  | y [1](https://www.kernel.org/doc/html/latest/driver-api/clk.html" \l "id3) |  |  |  |
| .determine\_rate |  | y **[1](https://www.kernel.org/doc/html/latest/driver-api/clk.html" \l "id3)** |  |  |  |
| .set\_rate |  | y |  |  |  |
|  |  |  |  |  |  |
| .set\_parent |  |  | n | y | n |
| .get\_parent |  |  | n | y | n |
|  |  |  |  |  |  |
| .recalc\_accuracy |  |  |  |  |  |
|  |  |  |  |  |  |
| .init |  |  |  |  |  |

注：[1](1,2)：round\_rate或determine\_rate之一是必须的。

最后，在运行时使用特定于硬件的注册函数注册时钟。此函数仅填充struct clk\_foo的数据，然后使用以下调用将公共struct clk参数传递给框架

clk\_register(...)

请查看drivers/clk/clk-\*.c中的基本时钟类型示例。

## 禁用未使用时钟的时钟门控制

有时在开发过程中，能够绕过未使用时钟的默认禁用可以非常有用。例如，如果驱动程序未正确启用时钟但依赖于从引导加载程序启用时钟，则绕过禁用意味着驱动程序将保持可用，而问题正在解决。

要绕过此禁用，请在kernel的bootargs中包括“clk\_ignore\_unused”。

## 锁定

公共时钟框架使用两个全局锁，准备锁和启用锁。

启用锁是自旋锁，跨 .enable，.disable操作调用保持。因此，这些操作不被允许休眠，并且在原子上下文中允许调用clk\_enable()、clk\_disable() API函数。

对于clk\_is\_enabled() API，它还被设计为允许在原子上下文中使用。但是，除非您希望在该锁处处理启用状态信息之外的其他信息，否则在核心中持有启用锁并不真正有意义。否则，查看时钟是否启用只是一次读取启用状态的读取，因为锁被释放后很容易更改。因此，使用此API的用户需要处理在读取状态时与其使用的内容进行同步，以确保在此期间不会更改启用状态。

准备锁是一个互斥锁，在其他所有操作调用期间保持。所有这些操作都允许休眠，并且不允许在原子上下文中调用相应的API函数。

从锁定的角度来看，这有效地将操作分成两组。

无论这些资源是否被多个时钟共享，驱动程序都不需要手动保护这一组操作的资源。然而，共享两组操作之间的资源访问需要由驱动程序进行保护。此类资源的一个示例是控制时钟速率和时钟启用/禁用状态的寄存器。

时钟框架是可重入的，即驱动程序允许从其实现时钟操作中调用时钟框架函数。例如，这可能会导致调用一个钟的.set\_rate操作从另一个钟的.set\_rate操作中被调用。在这种情况下，必须考虑到这种情况的驱动程序实现，但代码流通常由驱动程序控制。

请注意，当在常见的时钟框架之外的代码需要访问时钟操作使用的资源时也必须考虑锁定。这被认为超出了本文档的范围。

# 独立于总线的设备访问

## 介绍

Linux 提供了一个 API，抽象出在所有总线和设备上执行 IO 的过程，使设备驱动程序能够独立于总线类型编写。

## 内存映射 IO

### 获取设备访问权限

最广泛支持的 IO 形式是内存映射 IO。也就是说，CPU 的地址空间的一部分被解释为对设备的访问，而不是对内存的访问。有些体系结构将设备定义为在固定地址处，但大多数有一些发现设备的方法。PCI 总线访问就是这样一个很好的例子。本文不包括如何接收这样的地址，但假设您已经有一个地址。物理地址属于 unsigned long 类型。

这个地址不能直接使用。相反，要获得一个适合传递给下面说明的访问器函数的地址，您应该调用 ioremap()。会返回一个适合访问设备的地址。

在完成使用设备后（例如在模块的退出例程中），调用 iounmap() 以将地址空间返回给内核。大多数体系结构每次调用 ioremap() 都会分配新的地址空间，如果不调用 iounmap()，可能会用完。

### 访问设备

驱动程序最常用的接口部分是读取和写入设备上的内存映射寄存器。Linux 提供接口来读取和写入 8 位、16 位、32 位和 64 位量。由于历史原因，它们被命名为字节、字、长和四字节访问。这些函数都支持读写访问；目前没有预取支持。

这些函数的名称是 readb()、readw()、readl()、readq()、readb\_relaxed()、readw\_relaxed()、readl\_relaxed()、readq\_relaxed()、writeb()、writew()、writel() 和 writeq()。

一些设备（如帧缓冲区）希望一次使用比 8 个字节更大的传输。对于这些设备，提供了 memcpy\_toio()、memcpy\_fromio() 和 memset\_io() 函数。不要在 IO 地址上使用 memset 或 memcpy；它们不能保证按顺序复制数据。

读和写函数被定义为有序的。也就是说，编译器不能重新排列 I/O 序列。当顺序可以编译器优化时，可以使用 \_\_readb() 和类似的函数指示放松的顺序。使用时要小心。

虽然基本函数被定义为在彼此之间同步和有序的，但设备所在的总线本身可能具有异步性。特别是许多作者遇到了 PCI 总线写入异步发布的事实。驱动程序作者必须从同一设备发出读取以确保写入已经发生在作者关心的特定情况下。这种属性不能在 API 中隐藏驱动程序作者。在某些情况下，用于刷新设备的读取可能会失败（例如，如果卡正在重置）。在这种情况下，应该从配置空间中进行读取，如果卡不响应，则保证软件失败。

以下是一个示例，当驱动程序希望确保写入的效果在继续执行之前可见时，刷新到设备的写入:

static inline void

qla1280\_disable\_intrs(struct scsi\_qla\_host \*ha)

{

struct device\_reg \*reg;

reg = ha->iobase;

/\* disable risc and host interrupts \*/

WRT\_REG\_WORD(&reg->ictrl, 0);

/\*

\* The following read will ensure that the above write

\* has been received by the device before we 返回 from this

\* function.

\*/

RD\_REG\_WORD(&reg->ictrl);

ha->flags.ints\_enabled = 0;

}

除了写入发布之外，在一些大型多处理系统（如 SGI Challenge、Origin 和 Altix 机器）中，来自不同 CPU 的发布写入顺序不会强制。因此，对进行内存映射写入的驱动程序的某些部分进行适当的保护，并使用 mmiowb() 确保它们按预期顺序到达是非常重要的。发出常规的 readX() 也将确保写入顺序，但应该仅在驱动程序必须确保写入实际到达设备（而不仅仅是与其他写入有序）时使用，因为完整的 readX() 是一个相对昂贵的操作。

通常，在释放使用 writeb() 或类似函数（它们没有被 readb() 调用包围）保护区域的自旋锁之前，应该使用 mmiowb()，它将确保顺序和刷新。以下伪代码说明了如果没有通过 mmiowb() 或其中一个 readX() 函数保证写入顺序会发生什么:

CPU A: spin\_lock\_irqsave(&dev\_lock, flags)

CPU A: ...

CPU A: writel(newval, ring\_ptr);

CPU A: spin\_unlock\_irqrestore(&dev\_lock, flags)

...

CPU B: spin\_lock\_irqsave(&dev\_lock, flags)

CPU B: writel(newval2, ring\_ptr);

CPU B: ...

CPU B: spin\_unlock\_irqrestore(&dev\_lock, flags)

在上面的情况下，newval2可能会在newval之前写入ring\_ptr。 修复这很容易:

CPU A: spin\_lock\_irqsave(&dev\_lock, flags)

CPU A: ...

CPU A: writel(newval, ring\_ptr);

CPU A: mmiowb(); /\* ensure no other writes beat us to the device \*/

CPU A: spin\_unlock\_irqrestore(&dev\_lock, flags)

...

CPU B: spin\_lock\_irqsave(&dev\_lock, flags)

CPU B: writel(newval2, ring\_ptr);

CPU B: ...

CPU B: mmiowb();

CPU B: spin\_unlock\_irqrestore(&dev\_lock, flags)

请参阅tg3.c，了解如何使用真实的示例mmiowb()

PCI排序规则还保证PIO读取响应比该总线上任何未完成的DMA写入到达，因为对于某些设备，readb（）调用的结果可能会向驱动程序发出信号说明DMA事务已完成。 但是，在许多情况下，驱动程序可能希望指示下一个readb（）调用与设备执行的任何先前DMA写入无关。 驱动程序可以对这些情况使用readb\_relaxed（），尽管仅有一些平台将遵守宽松语义。 在支持宽松读取函数的平台上使用宽松读取函数将提供显着的性能收益。 qla2xxx驱动程序提供了如何使用readX\_relaxed（）的示例。 在许多情况下，驱动程序的大多数readX（）调用可以安全地转换为readX\_relaxed（）调用，因为只有少数调用会指示或依赖于DMA完成。

## 端口空间访问

### 端口空间解释

另一种常见支持的IO形式是端口空间。 这是一个范围，与正常的内存地址空间分开。 对这些地址的访问通常不如访问内存映射地址快，它也有一个潜在的较小地址空间。

与内存映射IO不同，访问端口空间不需要做任何准备工作。

### 访问端口空间

对此空间的访问通过一组允许8位，16位和32位访问的函数提供；也称为字节，字和长。 这些函数为inb（），inw（），inl（），outb（），outw（）和outl（）。

这些函数提供了一些变体。 有些设备要求访问它们的端口速度变慢。 这个功能通过在函数末尾附加\_p提供。 还有相当于memcpy的东西。 ins（）和outs（）函数将字节，字或长复制到给定端口。

## 提供的公共函数

### phys\_addr\_t virt\_to\_phys（volatile void \* address）

将虚拟地址映射到物理地址

**参数**

volatile void \*地址

要重新映射的地址

返回的物理地址是给定内存地址的物理（CPU）映射。仅在直接映射或通过kmalloc分配的地址上使用此函数才有效。

此函数不提供DMA传输的总线映射。 在几乎所有可想象的情况下，设备驱动程序都不应使用此功能。

### void \* phys\_to\_virt（phys\_addr\_t）

将物理地址映射到虚拟地址

**参数**

phys\_addr\_taddress

要重新映射的地址

返回的虚拟地址是给定内存地址的当前CPU映射。仅在具有内核映射的地址上使用此功能才有效

此函数不处理DMA传输的总线映射。 在几乎所有可想象的情况下，设备驱动程序都不应使用此功能。

### void \_\_iomem \* ioremap（资源大小\_t offset，unsigned long size）

将总线内存映射到CPU空间

**参数**

资源大小\_t offset

内存的总线地址

unsigned long size

要映射的资源大小

**说明**

ioremap执行一个特定于平台的操作序列，使总线内存通过readb / readw / readl / writeb / writew / writel函数和其他mmio助手地址变得CPU可访问。

如果您尝试映射的区域是PCI BAR，则应查看pci\_iomap（）。

### void \_\_iomem \* pci\_iomap\_range（struct pci\_dev \* dev，int bar，unsigned long offset，unsigned long maxlen）

为PCI BAR创建虚拟映射cookie

**参数**

struct pci\_dev \* dev

拥有BAR的PCI设备

int bar

BAR号码

unsigned long offset

在BAR中给定偏移量的映射内存

unsigned long maxlen

要映射的存储器的最大长度

**说明**

使用此函数，您将获得一个\_\_iomem地址，可以使用ioread \*（）和iowrite \*（）访问您的设备BAR。 这些函数隐藏了这是否是MMIO或PIO地址空间的细节，并且会按正确的方式对它们执行您所期望的操作。

maxlen指定要映射的最大长度。 如果要从偏移量到结尾访问完整的BAR，请在此处传递0。

### void \_\_iomem \* pci\_iomap\_wc\_range（struct pci\_dev \* dev，int bar，unsigned long offset，unsigned long maxlen）

为PCI BAR创建虚拟WC映射cookie

**参数**

struct pci\_dev \*dev

拥有BAR的PCI设备

int bar

BAR编号

unsigned long offset

映射在BAR中给定偏移量处的内存

unsigned long maxlen

映射的最大内存长度

**说明**

使用此功能，您将获得指向设备BAR的\_\_iomem地址。您可以使用ioread\*()和iowrite\*()访问它们。这些函数隐藏了这是否为MMIO或PIO地址空间的细节，并将以正确的方式执行您所期望的操作。尽可能使用写组合。

maxlen指定要映射的最大长度。如果要从偏移量到结尾获取对整个BAR的访问权限，请在此处传递0。

### void \_\_iomem \*pci\_iomap(struct pci\_dev \*dev, int bar, unsigned long maxlen)

为PCI BAR创建虚拟映射cookie

**参数**

struct pci\_dev \*dev

拥有BAR的PCI设备

int bar

BAR编号

unsigned long maxlen

要映射的内存的长度

**说明**

使用此功能，您将获得指向设备BAR的\_\_iomem地址。您可以使用ioread\*()和iowrite\*()访问它们。这些函数隐藏了这是否为MMIO或PIO地址空间的细节，并将以正确的方式执行您所期望的操作。

maxlen指定要映射的最大长度。如果要先获取对整个BAR的访问权限，而不必先检查其长度，请在此处传递0。

### void \_\_iomem \*pci\_iomap\_wc(struct pci\_dev \*dev, int bar, unsigned long maxlen)

为PCI BAR创建虚拟WC映射cookie

**参数**

struct pci\_dev \*dev

拥有BAR的PCI设备

int bar

BAR编号

unsigned long maxlen

要映射的内存的长度

**说明**

使用此功能，您将获得指向设备BAR的\_\_iomem地址。您可以使用ioread\*()和iowrite\*()访问它们。这些函数隐藏了这是否为MMIO或PIO地址空间的细节，并将以正确的方式执行您所期望的操作。尽可能使用写组合。

maxlen指定要映射的最大长度。如果要先获取对整个BAR的访问权限，而不必先检查其长度，请在此处传递0。

# 设备连接

## 介绍

设备通常与直接子/父关系之外的其他设备有连接。串行或网络通信控制器（可能是PCI设备）可能需要能够获取其PHY组件的引用，例如附加到I2C总线上。某些设备驱动程序需要能够控制其设备的时钟或GPIO等等。

设备连接是两个独立设备之间任何类型连接的通用说明。

设备连接本身不会创建两个设备之间的依赖关系。它们只是说明，并不直接绑定到任何设备。只有当两个端点设备之一请求对另一个设备的引用时，两个设备之间才存在依赖关系。说明本身可以在固件中定义（尚未受支持），也可以内置。

## 用法

设备连接应该在任一端点设备的 ->probe 回调被调用之前存在。如果连接是在固件中定义的，这不是问题。如果连接说明是“内置”的，并且需要单独添加，则应考虑这一点。

连接说明由连接的两个设备的名称组成，即端点，并带有连接的唯一标识符，如果两个设备之间存在多个连接，则需要使用该标识符区分它们。

存在说明后，其中的设备可以请求引用另一个端点设备，也可以请求说明本身。

## API

### void \* device\_connection\_find\_match(struct device \* dev, const char \* con\_id, void \* data, void \*(\*match) (struct device\_connection \*con, int ep, void \*data)

查找到设备的物理连接

**参数**

struct device \* dev

具有连接的设备

const char \* con\_id

连接的标识符

void \* data

匹配函数使用的数据

void \*(\*)(struct device\_connection \*con, int ep, void \*data) match

用于检查和转换连接说明的函数

**说明**

查找 dev 和另一个设备之间的一个具有唯一标识符 con\_id 的连接。match 将用于将连接说明转换为调用者期望返回的数据。

### struct device \* device\_connection\_find(struct device \* dev, const char \* con\_id)

查找连接在一起的两个设备

**参数**

struct device \* dev

具有连接的设备

const char \* con\_id

连接的标识符

**说明**

查找 dev 和另一个设备之间的一个具有唯一标识符 con\_id 的连接。成功返回已连接到 dev 的设备的句柄，并增加找到的设备的引用计数。如果未找到匹配连接，则返回NULL，或者如果找到连接但另一个设备尚未enum，则返回ERR\_PTR（-EPROBE\_DEFER）。

### void device\_connection\_add（struct device\_connection \* con）

注册连接说明

**参数**

struct device\_connection \* con

需要注册的连接说明

### void device\_connection\_remove（struct device\_connection \* con）

取消注册连接说明

**参数**

struct device\_connection \* con

需要取消注册的连接说明

# 缓冲区共享和同步

dma-buf子系统提供了共享硬件（DMA）访问缓冲区的框架，跨多个设备驱动程序和子系统进行共享，并同步异步硬件访问。

例如，drm“prime”多GPU支持使用它，但当然不仅限于GPU用例。

其中的三个主要组件是（1）dma-buf，代表sg\_table并作为文件说明符向用户空间公开，以允许在设备之间传递，（2）fence，提供一种机制来指示一个设备已完成访问，（3）预留区，管理与缓冲区相关联的共享或独占fence（s）。

## 共享DMA缓冲区

本文档作为一份指南，指导设备驱动程序编写者了解什么是dma-buf缓冲区共享API，如何使用它进行导出和使用共享缓冲区。

任何希望成为DMA缓冲区共享的设备驱动程序都可以作为缓冲区的“导出者”或“用户”或“导入者”来实现。

例如，假设A驱动程序想要使用由B驱动程序创建的缓冲区，那么我们称B为导出者，称A为缓冲区用户/导入者。

导出者：

1. 实现并管理struct dma\_buf\_ops中缓冲区的操作
2. 通过使用dma\_buf共享API允许其他用户共享缓冲区
3. 管理缓冲区分配的详细信息，包装在struct dma\_buf中
4. 决定实际的后备存储空间，其中进行此分配
5. 并负责对所有（共享）缓冲区用户迁移散列表。

缓冲区用户：

1. 是缓冲区的（许多）共享用户之一
2. 不需要担心缓冲区如何分配或在哪里分配。
3. 需要一种机制来获取访问组成此缓冲区内存的散列表，将其映射到自己的地址空间，以便可以访问相同的内存区域。此接口由struct dma\_buf\_attachment提供。

dma-buf缓冲区共享框架的任何导出者或用户都必须在各自的Kconfigs中具有“选择DMA\_SHARED\_BUFFER”。

### 用户空间接口注意事项

对于用户空间而言，大多数情况下DMA缓冲区文件说明符只是一个不透明对象，因此通常公开的通用接口非常简单。但是，有一些事项需要考虑

1）自\_kernel 3.12以来，dma-buf FD支持llseek系统调用，但只支持offset = 0和whence = SEEK\_END | SEEK\_SET。支持SEEK\_SET以允许通常的大小发现模式大小=SEEK\_END（0）; SEEK\_SET（0）。每个其他llseek操作都将报告-EINVAL。

如果dma-buf FD上的llseek不支持内核，则会报告所有情况下的-ESPIPE。用户空间可以使用此来检测使用llseek发现dma-buf大小的支持。

2）为避免在exec上发生fd泄漏，必须在文件说明符上设置FD\_CLOEXEC标志。这不仅是资源泄漏，而且是潜在的安全漏洞。通过泄漏的fd，它可能会使新的exec'd应用程序访问应该否则不允许访问的缓冲区。

通过单独的fcntl（）调用而不是在创建fd时原子地进行此操作的问题在于，在多线程应用程序中，这在本质上是竞争。当是库代码打开/创建文件说明符时，问题变得更加严重，因为应用程序可能甚至不知道fd的情况。

为避免这个问题，用户空间必须有一种方法来请求在创建dma-buf fd时设置O\_CLOEXEC标志。因此，提供将dmabuf fd创建API的导出驱动程序必须提供一种让用户控制传递给dma\_buf\_fd（）的O\_CLOEXEC标志的方法。

3）还支持内存映射DMA缓冲区的内容。有关完整详细信息，请参见下面关于CPU访问DMA缓冲区对象的讨论。

4）DMA缓冲区FD也可以轮询，请参见下面的隐式栅栏轮询支持以获取详细信息。

### 基本操作和设备DMA访问

对于设备DMA访问共享DMA缓冲区，通常操作序列非常简单：

1. 导出者使用DEFINE\_DMA\_BUF\_EXPORT\_INFO（）定义其导出者实例，并调用dma\_buf\_export（）将私有缓冲区对象包装成为dma\_buf。然后，通过调用dma\_buf\_fd（）将其导出到用户空间作为文件说明符。
2. 用户空间将此文件说明符传递给希望共享此缓冲区的所有驱动程序首先，使用dma\_buf\_get（）将文件说明符转换为dma\_buf。然后，使用dma\_buf\_attach（）将缓冲区附加到设备。

到目前为止，出口商仍然可以自由迁移或重新分配后备存储。

1. 一旦缓冲区连接到所有设备，用户空间就可以启动对共享缓冲区的DMA访问。在内核中，这是通过调用dma\_buf\_map\_attachment（）和dma\_buf\_unmap\_attachment（）来完成的。
2. 一旦驱动程序完成了共享缓冲区，它需要调用dma\_buf\_detach（）（在清理任何映射之后），然后通过调用dma\_buf\_put（）释放通过dma\_buf\_get（）获得的引用。

对于出口商预计实现的详细语义，请参见dma\_buf\_ops。

### CPU访问DMA缓冲区对象

支持CPU访问dma缓冲区对象的原因有多种

1）在内核中进行回退操作，例如当设备通过USB连接时，内核需要先将数据移动到发送之前。通过调用dma\_buf\_begin\_cpu\_access（）和dma\_buf\_end\_cpu\_access（）访问来处理缓存一致性。

为了支持驻留在 highmem 中的 dma\_buf 对象，cpu 访问是基于页面的，使用类似于 kmap 的 api。访问 dma\_buf 是在 PAGE\_SIZE 大小的对齐块中完成的。在访问块之前需要对其进行映射，这会返回内核虚拟地址空间中的指针。之后该块需要再次取消映射。给定块可以映射和取消映射的频率没有限制，即导入器不需要在再次映射同一块之前再次调用 begin\_cpu\_access。

接口：

void \*dma\_buf\_kmap(struct dma\_buf \*, unsigned long);

 void dma\_buf\_kunmap(struct dma\_buf \*, unsigned long, void \*);

实现这些功能对于出口商来说是可选的，对于进口商来说，使用 kmap 的所有限制都适用。

超出 begin\_cpu\_access 指定范围的 dma\_buf kmap 调用是未定义的。如果范围不是 PAGE\_SIZE 对齐的，kmap 需要在开始和结束的部分块上成功，但可能会返回范围之外的陈旧或伪造数据（在这些部分块中）。

对于某些情况，kmap 的开销可能太高，因此引入了 vmap 接口。应该非常小心地使用此接口，因为 vmalloc 空间在许多体系结构上都是有限的资源。

接口：

void \*dma\_buf\_vmap（struct dma\_buf \* dmabuf，struct iosys\_map \*map）

void dma\_buf\_vunmap（struct dma\_buf \* dmabuf，struct iosys\_map \*map）

如果出口商没有vmap支持，或者vmap运行出现vmalloc空间不足，则vmap调用可能会失败。请注意，dma-buf层为所有vmap访问保留引用计数，并且仅当没有映射存在时才调用出口商的vmap函数，并且仅在解除映射后。采用dma\_buf.lock互斥锁保护并发的vmap / vunmap调用。

2）为了与现有用户空间接口的完全兼容性，在进口商方面可能需要支持mmap缓冲区。这在许多处理管道中是必需的（例如将软件渲染的图像传输到硬件管道中，缩略图创建，快照等）。此外，Android的ION框架已经支持了这一点，为了用DMA缓冲区文件说明符替换ION缓冲区，需要支持mmap。

没有特殊的接口，用户空间只需在dma-buf fd上调用mmap。但是，与CPU访问一样，需要使用ioctl（DMA\_BUF\_IOCTL\_SYNC）来处理实际访问的需要。请注意，DMA\_BUF\_IOCTL\_SYNC可能会出现-EAGAIN或-EINTR失败，这种情况下必须重新启动它。

某些系统可能需要某种缓存一致性管理，例如当CPU和GPU域同时通过dma-buf进行访问时。为了避免这个问题，有一个开始/结束一致性标记，直接转发到现有的dma-buf设备驱动程序vfunc挂钩。用户空间可以通过DMA\_BUF\_IOCTL\_SYNC ioctl使用这些标记。序列将如下所示：

映射dma-buf fd；

对于CPU中的每个绘图/上传周期1. SYNC\_START ioctl，2.读/写到mmap区域3. SYNC\_END ioctl。这可以重复多次（新数据被GPU或扫描设备消耗）；

一旦您不再需要缓冲区，即munmap。

为了正确性和最佳性能，始终需要在访问映射的地址之前和之后使用SYNC\_START和SYNC\_END。即使有系统可以依靠一致的访问，用户空间也不能依赖它们，即使在调用这些ioctl时存在这些系统。

3）并作为用户空间处理管道中的CPU回退。

与内核CPU访问的动机类似，重要的是给定导入子系统的用户空间代码可以使用与原生缓冲区对象相同的接口来使用导入的dma-buf缓冲区对象。这对于drm来说尤为重要，因为现代OpenGL，X和其他驱动程序的用户空间部分非常庞大，重做它们以使用不同的方式来映射缓冲区相当具有侵入性。

当前dma-buf接口的假设是重定向初始mmap就足够了。对一些现有子系统的调查显示，似乎没有驱动程序执行任何恶意事情，例如与设备上未完成的异步处理同步或在故障时间分配特殊资源。因此，希望这已经足够，因为添加拦截页错误的接口并允许pte刷新会大大增加复杂度。

接口：

int dma\_buf\_mmap（struct dma\_buf \*，struct vm\_area\_struct \*，

unsigned long）;

如果导入子系统只需提供特殊用途的mmap调用以在用户空间中设置映射，则使用dma\_buf.file调用do\_mmap同样可以实现dma-buf对象的目的。

### 栅栏轮询支持

为了支持缓冲区访问的交叉设备和交叉驱动程序同步，可以将隐式栅栏（在内核中用struct dma\_fence表示）附加到dma\_buf。这个胶水和一些相关的东西在dma\_resv结构中提供。

用户空间可以使用poll()和相关系统调用查询这些隐式跟踪栅栏的状态

1）检查EPOLLIN，即读访问，可用于查询最近一次写入或独占栅栏的状态。

2）检查EPOLLOUT，即写访问，可用于查询所有附加栅栏（共享和独占）的状态。

注意，这只表示相应栅栏的完成，即DMA传输完成。在开始CPU访问之前，仍需要进行缓存刷新和任何其他必要的准备工作。

### 内核函数和结构引用

#### struct dma\_buf \* dma\_buf\_export(const struct dma\_buf\_export\_info \*exp\_info)

创建一个新的dma\_buf，并将一个匿名文件与此缓冲区关联，以便可以导出此缓冲区。还将分配器特定的数据和操作连接到缓冲区。此外，为出口商提供有用的名称字符串；在调试中很有用。

**参数**

const struct dma\_buf\_export\_info \*exp\_info

[in]由出口商提供的所有与导出相关的信息。请参见struct dma\_buf\_export\_info以获取进一步的详细信息。

**说明**

成功时返回一个新创建的dma\_buf对象，该对象包装了dma\_buf\_ops的所提供的私有数据和操作。如果缺少操作或在分配struct dma\_buf时出错，则返回负错误。

对于大多数情况，创建exp\_info的最简单方法是通过DEFINE\_DMA\_BUF\_EXPORT\_INFO宏。

#### int dma\_buf\_fd(struct dma\_buf \*dmabuf, int flags)

返回给定dma\_buf的文件说明符

**参数**

struct dma\_buf \*dmabuf

[in]需要fd的dma\_buf指针。

int flags

[in]要给fd的标志

**说明**

成功时返回关联的“fd”。否则，返回错误。

#### struct dma\_buf \* dma\_buf\_get(int fd)

返回与fd相关联的dma\_buf结构

**参数**

int fd

与要返回的dma\_buf相关联的fd

**说明**

成功时返回与fd关联的dma\_buf结构；使用fget执行文件的引用计数来增加引用计数。否则返回ERR\_PTR。

#### void dma\_buf\_put(struct dma\_buf \*dmabuf)

减少缓冲区的引用计数

**参数**

struct dma\_buf \*dmabuf

[in]要减少引用计数的缓冲区

**说明**

隐式地使用fput()对该文件的引用计数进行调用。

如果调用的结果是引用计数变为0，则调用与该fd相关联的“释放”文件操作。依次调用dma\_buf\_ops.release vfunc，并在导出时释放为dmabuf分配的内存。

#### struct dma\_buf\_attachment \* dma\_buf\_attach(struct dma\_buf \*dmabuf, struct device \*dev)

将设备添加到DMA缓冲区的附件列表中；可以选择调用dma\_buf\_ops的attach()来允许特定于设备的附加功能

**参数**

struct dma\_buf \*dmabuf

[in]要附加设备的缓冲区。

struct device \*dev

[in]要附加的设备。

**说明**

返回此附件的struct dma\_buf\_attachment指针。必须通过调用dma\_buf\_detach()来清理附件。

**返回**

成功时，返回新创建的dma\_buf\_attachment指针，否则将负错误代码包装到指针中。

请注意，如果dmabuf的后备存储不可访问于dev，则此操作可能失败，并且不能移动到更合适的位置。这用错误代码-EBUSY表示。

#### void dma\_buf\_detach(struct dma\_buf \*dmabuf, struct dma\_buf\_attachment \*attach)

从dmabuf的附件列表中删除给定的附件；可以选择为特定于设备的分离调用dma\_buf\_ops的detach()

**参数**

struct dma\_buf \*dmabuf

[in]要与其分离的缓冲区。

struct dma\_buf\_attachment \*attach

[in]要分离的附件；此调用后将被释放。

**说明**

清理通过调用dma\_buf\_attach()获得的设备附件。

#### struct sg\_table \* dma\_buf\_map\_attachment(struct dma\_buf\_attachment \*attach, enum dma\_data\_direction direction)

返回附件的分散列表表；映射为\_device\_地址空间。是dma\_buf\_ops的map\_dma\_buf()的封装器。

**参数**

struct dma\_buf\_attachment \*attach

[in]要返回其scatterlist的附件

enum dma\_data\_direction direction

[in]DMA传输的方向

**说明**

返回包含要返回的scatterlist的sg\_table；返回错误时返回ERR\_PTR。如果被信号打断，可能会返回-EINTR。

必须使用dma\_buf\_unmap\_attachment()取消映射映射。请注意，底层后备存储器在映射存在的情况下被固定，因此用户/导入者不应持有映射时间过长。

#### void dma\_buf\_unmap\_attachment(struct dma\_buf\_attachment \*attach, struct sg\_table \*sg\_table, enum dma\_data\_direction direction)

取消映射并减少缓冲区的使用计数；可能会释放相关的分散列表。是dma\_buf\_ops的unmap\_dma\_buf()的封装器。

**参数**

struct dma\_buf\_attachment \*attach

[in]要从其中取消映射缓冲区的附件

struct sg\_table \*sg\_table

[in]要取消映射其缓冲区的scatterlist信息

enum dma\_data\_direction direction

[in]DMA传输的方向

**说明**

这将取消映射通过dma\_buf\_map\_attachment()获得的附加的DMA映射。

#### int dma\_buf\_begin\_cpu\_access(struct dma\_buf \*dmabuf, enum dma\_data\_direction direction)

必须在内核上下文中从CPU访问dma\_buf之前调用。调用begin\_cpu\_access以允许出口商特定的准备工作。对于指定的访问方向，仅在指定范围内保证一致性。

**参数**

struct dma\_buf \* dmabuf

[in] 为CPU准备访问的缓冲区。

enum dma\_data\_direction direction

[in] CPU访问范围的长度。

**说明**

CPU访问完成后，调用方应调用dma\_buf\_end\_cpu\_access()。只有在CPU访问同时由两个调用包围时，才保证与其他DMA访问一致。

可能返回负错误值，成功返回0。

#### int dma\_buf\_end\_cpu\_access(struct dma\_buf \* dmabuf，enum dma\_data\_direction direction)

在内核上下文中从CPU访问dma\_buf后必须调用。调用end\_cpu\_access以允许出口特定操作。一致性仅在指定范围和指定访问方向内保证。

**参数**

struct dma\_buf \* dmabuf

[in] 完成CPU访问的缓冲区。

enum dma\_data\_direction direction

[in] CPU访问范围的长度。

**说明**

这终止了使用dma\_buf\_begin\_cpu\_access()启动的CPU访问。

可能返回负错误值，成功返回0。

#### void \* dma\_buf\_kmap(struct dma\_buf \* dmabuf，unsigned long page\_num)

将缓冲对象的页面映射到内核地址空间。与kmap和朋友们的限制相同。

**参数**

struct dma\_buf \* dmabuf

[in] 映射页面的缓冲区。

unsigned long page\_num

[in] 要映射的页面，以PAGE\_SIZE为单位。

**说明**

此调用必须始终成功，任何可能失败的必要准备工作都需要在begin\_cpu\_access中完成。

#### void dma\_buf\_kunmap(struct dma\_buf \* dmabuf，unsigned long page\_num，void \* vaddr)

取消映射由dma\_buf\_kmap获得的页面。

**参数**

struct dma\_buf \* dmabuf

[in] 取消映射页面的缓冲区。

unsigned long page\_num

[in] 要取消映射的页面，以PAGE\_SIZE为单位。

void \* vaddr

[in] 从dma\_buf\_kmap获得的内核空间指针。

**说明**

此调用必须始终成功。

#### int dma\_buf\_mmap(struct dma\_buf \* dmabuf，struct vm\_area\_struct \* vma，unsigned long pgoff)

设置具有给定vma的用户空间mmap

**参数**

struct dma\_buf \* dmabuf

[in] 应支持vma的缓冲区

struct vm\_area\_struct \* vma

[in] mmap的vma

unsigned long pgoff

[in] 从dma-buf缓冲区中的哪个页面偏移量开始此mmap应开始。

**说明**

此函数调整传入的vma，以便指向dma\_buf操作的文件。它还调整起始pgoff并对vma的大小进行边界检查。然后它调用出口方的mmap函数来设置映射。

可能返回负错误值，成功返回0。

#### void \* dma\_buf\_vmap(struct dma\_buf \* dmabuf)

在内核地址空间中为缓冲区对象创建虚拟映射。与vmap和朋友们的限制相同。

**参数**

struct dma\_buf \* dmabuf

[in] 要vmap的缓冲区

**说明**

由于缺少虚拟映射地址空间，此调用可能会失败。这些调用在驱动程序中是可选的。它们的预期用途是用于高使用对象的线性内核空间映射。请尝试在考虑这些接口之前使用kmap / kunmap。

出错时返回NULL。

#### void dma\_buf\_vunmap(struct dma\_buf \* dmabuf，void \* vaddr)

取消映射由dma\_buf\_vmap获得的vmap。

**参数**

struct dma\_buf \* dmabuf

[in] 要取消映射的缓冲区

void \* vaddr

[in] 要取消映射的vmap

#### struct dma\_buf\_ops

可在struct dma\_buf上执行的操作

**定义**

struct dma\_buf\_ops {

int (\*attach)(struct dma\_buf \*, struct dma\_buf\_attachment \*);

void (\*detach)(struct dma\_buf \*, struct dma\_buf\_attachment \*);

struct sg\_table \* (\*map\_dma\_buf)(struct dma\_buf\_attachment \*, enum dma\_data\_direction);

void (\*unmap\_dma\_buf)(struct dma\_buf\_attachment \*,struct sg\_table \*, enum dma\_data\_direction);

void (\*release)(struct dma\_buf \*);

int (\*begin\_cpu\_access)(struct dma\_buf \*, enum dma\_data\_direction);

int (\*end\_cpu\_access)(struct dma\_buf \*, enum dma\_data\_direction);

void \*(\*map)(struct dma\_buf \*, unsigned long);

void (\*unmap)(struct dma\_buf \*, unsigned long, void \*);

int (\*mmap)(struct dma\_buf \*, struct vm\_area\_struct \*vma);

void \*(\*vmap)(struct dma\_buf \*);

void (\*vunmap)(struct dma\_buf \*, void \*vaddr);

};

**成员**

attach

调用 from[dma\_buf\_attach()](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_buf_attach" \o "dma_buf_attach)以确保给定的 [dma\_buf\_attachment.dev](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_buf_attachment" \o "dma_buf_attachment)可以访问提供的[dma\_buf](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_buf" \o "dma_buf)。支持特殊位置（如 VRAM 或特定于设备的分割区域）中的缓冲区对象的导出器应检查缓冲区是否可以移动到系统内存（或由提供的设备直接访问），否则需要使附加操作失败。

导出器通常还应检查当前分配是否满足新设备的 DMA 约束。如果不是这种情况，并且无法移动分配，则附加操作也应该失败。

任何出口商私人管家数据都可以存储在 [dma\_buf\_attachment.priv](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_buf_attachment" \o "dma_buf_attachment)指针中。

此回调是可选的。

退货：

成功时为 0，失败时为负错误代码。它可能会返回 -EBUSY 以表示后备存储已经分配并且与请求设备的要求不兼容。

detach

这被调用[dma\_buf\_detach()](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_buf_detach" \o "dma_buf_detach)以释放一个[dma\_buf\_attachment](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_buf_attachment" \o "dma_buf_attachment). 提供这样的出口商可以清理 [dma\_buf\_attachment](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_buf_attachment" \o "dma_buf_attachment).

此回调是可选的。

map\_dma\_buf

这由调用[dma\_buf\_map\_attachment()](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_buf_map_attachment" \o "dma_buf_map_attachment)并用于将共享映射[dma\_buf](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_buf" \o "dma_buf)到设备地址空间，这是强制性的。只有成功调用attach后才能调用它。这实质上将 DMA 缓冲区固定到位，并且不能再移动它

该调用可能会休眠，例如，当首先需要分配后备存储或将其移动到适合所有当前连接的设备的位置时。

请注意，此功能所需的任何特定缓冲区属性都应添加到 device\_dma\_parameters 中，可 [device.dma\_params](https://www.kernel.org/doc/html/v4.19/driver-api/infrastructure.html" \l "c.device" \o "设备)通过[dma\_buf\_attachment](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_buf_attachment" \o "dma_buf_attachment). 附加回调也应该检查这些约束。

如果这是第一次调用，导出器现在可以选择扫描此缓冲区的附件列表，整理附加设备的要求，并为缓冲区选择合适的后备存储。

基于 enum dma\_data\_direction，可能有多个用户同时访问（可能是为了阅读），或者导出器可能希望为缓冲区用户提供的任何其他类型的共享。

退货：

DMA 缓冲区的分散列表或后备存储，已映射到所提供的附件sg\_table的设备地址空间中。[device](https://www.kernel.org/doc/html/v4.19/driver-api/infrastructure.html" \l "c.device" \o "设备)[dma\_buf\_attachment](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_buf_attachment" \o "dma_buf_attachment)

失败时，返回一个包含在指针中的负错误值。在被阻塞时收到信号时也可能返回 -EINTR。

unmap\_dma\_buf

这由调用[dma\_buf\_unmap\_attachment()](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_buf_unmap_attachment" \o "dma_buf_unmap_attachment)并且应该取消映射并释放sg\_table在map\_dma\_buf中分配的，这是强制性的。如果这是 DMA 缓冲区的最后一个映射，它还应该取消固定后备存储，如果导出器支持后备存储迁移。

release

在最后一个 dma\_buf\_put 释放之后调用[dma\_buf](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_buf" \o "dma_buf)，并且是强制的。

begin\_cpu\_access

这被调用[dma\_buf\_begin\_cpu\_access()](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_buf_begin_cpu_access" \o "dma_buf_begin_cpu_access)并允许导出器确保内存实际上可用于 cpu 访问 - 导出器可能需要分配或换入并固定后备存储。导出器还需要确保 cpu 访问对于访问方向是一致的。导出器可以使用该方向来优化缓存刷新，即使用不同方向（读取而不是写入）的访问可能会返回陈旧甚至伪造的数据（例如，当导出器需要将数据复制到临时存储时）。

此回调是可选的。

FIXME：这既是通过用户空间的 DMA\_BUF\_IOCTL\_SYNC 命令调用的（不应固定存储以避免将 de-factor mlock 权限交给用户空间），也适用于各种 kmap 接口的内核内部用户，其中后备存储必须是固定以保证原子 kmap 调用可以成功。由于没有 km​​ap 接口的内核用户，所以这不是真正的问题。

退货：

成功时为 0，失败时为负错误代码。例如，当无法分配后备存储时，这可能会失败。当调用被中断并需要重新启动时，也可以返回 -ERESTARTSYS 或 -EINTR。

end\_cpu\_access

[dma\_buf\_end\_cpu\_access()](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_buf_end_cpu_access" \o "dma_buf_end_cpu_access)当导入器完成访问 CPU 时调用它。导出器可以使用它来刷新缓存并取消固定在begin\_cpu\_access中固定的任何资源。end\_cpu\_access 之后的任何 dma\_buf kmap 调用的结果是未定义的。

此回调是可选的。

退货：

成功时为 0，失败时为负错误代码。当调用被中断并需要重新启动时，可以返回 -ERESTARTSYS 或 -EINTR。

map

[可选] 将页面从缓冲区映射到内核地址空间。

unmap

[可选] 从缓冲区取消映射页面。

mmap

[dma\_buf\_mmap()](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_buf_mmap" \o "dma_buf_mmap)此回调由函数使用

请注意，映射需要不连贯，用户空间需要使用 DMA\_BUF\_IOCTL\_SYNC 接口来限制 CPU 访问。

因为 dma-buf 缓冲区在其生命周期内大小不变，dma-buf 内核会检查 vma 是否太大并拒绝此类映射。因此，出口商无需重复此支票。驾驶员无需自行检查。

如果导出器需要手动刷新缓存，因此需要为 mmap 支持伪造一致性，则它需要能够清除所有指向后备存储的 ptes。现在 linux mm 需要一个与存储在 vma->vm\_file 中的结构文件关联的结构地址空间，以使用函数 unmap\_mapping\_range 来完成。但是dma\_buf 框架只用anon\_file 结构文件支持每个dma\_buf fd，即所有dma\_buf 共享同一个文件。

因此，导出器需要通过设置 vma->vm\_file 并在 dma\_buf mmap 回调中调整 vma->vm\_pgoff 来设置自己的文件（和地址空间）关联。在 gem 驱动程序的特定情况下，导出器可以使用 gem 已经提供的 shmem 文件（并设置 vm\_pgoff = 0）。然后导出器可以通过取消映射与他们自己的文件关联的结构地址空间的相应范围来切换文件。

此回调是可选的。

退货：

成功时为 0，失败时为负错误代码。

vmap

[可选] 为缓冲区创建到内核地址空间的虚拟映射。与 vmap 和 friends 相同的限制适用。

vunmap

[可选] 从缓冲区取消映射 vmap

#### struct dma\_buf

共享缓冲对象

**定义**

struct dma\_buf {

size\_t size;

struct file \*file;

struct list\_head attachments;

const struct dma\_buf\_ops \*ops;

struct mutex lock;

unsigned vmapping\_counter;

void \*vmap\_ptr;

const char \*exp\_name;

struct module \*owner;

struct list\_head list\_node;

void \*priv;

struct reservation\_object \*resv;

wait\_queue\_head\_t poll;

struct dma\_buf\_poll\_cb\_t {

struct dma\_fence\_cb cb;

wait\_queue\_head\_t \*poll;

\_\_poll\_t active;

} cb\_excl, cb\_shared;

};

**成员**

size

缓冲区的大小

file

用于跨共享缓冲区和重新计数的文件指针。

attachments

表示所有附加设备的 dma\_buf\_attachment 列表。

ops

与此缓冲区对象关联的 dma\_buf\_ops。

lock

在内部用于序列化列表操作、附加/分离和 vmap/unmap

vmapping\_counter

内部用于引用 vmap

vmap\_ptr

当前 vmap ptr 如果 vmapping\_counter > 0

exp\_name

出口商名称；对调试很有用。

owner

指向导出器模块的指针；当导出器是内核模块时用于引用计数。

list\_node

dma\_buf 计费和调试节点。

priv

此缓冲区对象的导出器特定私有数据。

resv

链接到此 dma-buf 的保留对象

poll

用于用户空间投票支持

cb\_excl

用于用户空间投票支持

cb\_shared

用于用户空间投票支持

**说明**

这表示一个共享缓冲区，通过调用创建[dma\_buf\_export()](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_buf_export" \o "dma_buf_export)。用户空间表示是一个普通的文件描述符，可以通过调用[dma\_buf\_fd()](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_buf_fd" \o "dma_buf_fd).

[dma\_buf\_put()](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_buf_put" \o "dma_buf_put)共享 dma 缓冲区使用和 进行引用计数[get\_dma\_buf()](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.get_dma_buf" \o "get_dma_buf)。

设备 DMA 访问由单独的.[struct dma\_buf\_attachment](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_buf_attachment" \o "dma_buf_attachment)

#### struct dma\_buf\_attachment

保存设备-缓冲区附件数据

**定义**

struct dma\_buf\_attachment {

struct dma\_buf \*dmabuf;

struct device \*dev;

struct list\_head node;

void \*priv;

};

**成员**

dmabuf

该附件的缓冲区。

dev

附加到缓冲区的设备。

node

dma\_buf\_attachment 列表，由 dmabuf 的 dma\_resv 锁保护。

priv

出口商特定的附件数据。

**说明**

此结构保存 dma\_buf 缓冲区与其用户设备之间的附件信息。列表包含每个附加到缓冲区的设备的一个附件结构。

通过调用 dma\_buf\_attach() 创建附件，通过调用 dma\_buf\_detach() 释放附件。需要启动传输的 DMA 映射是通过 dma\_buf\_map\_attachment() 创建的，并通过调用 dma\_buf\_unmap\_attachment() 再次释放。

#### struct dma\_buf\_export\_info

保存导出 dma\_buf 所需的信息

**定义**

struct dma\_buf\_export\_info {

const char \*exp\_name;

struct module \*owner;

const struct dma\_buf\_ops \*ops;

size\_t size;

int flags;

struct dma\_resv \*resv;

void \*priv;

};

**成员**

exp\_name

导出方的名称-对于调试很有用。

owner

导出器模块的指针-用于引用计数内核模块

ops

分配器定义的附加分配器 dma 缓冲区操作

size

缓冲区的大小-在缓冲区的生命周期中不变

flags

文件的模式标志

resv

预留对象，NULL 以分配默认对象

priv

将分配器的私有数据附加到此缓冲区

**说明**

此结构保存导出缓冲区所需的信息。仅与 dma\_buf\_export() 一起使用。

#### DEFINE\_DMA\_BUF\_EXPORT\_INFO

DEFINE\_DMA\_BUF\_EXPORT\_INFO (name)

出口商辅助宏

**参数**

名称

导出信息名称

**说明**

DEFINE\_DMA\_BUF\_EXPORT\_INFO 宏定义 struct dma\_buf\_export\_info，将其清零并预先填充 exp\_name。

#### void get\_dma\_buf(struct dma\_buf \*dmabuf)

方便的包装器 get\_file。

**参数**

struct dma\_buf \*dmabuf

[in] dma\_buf 指针

**说明**

增加 dma-buf 的引用计数，这在需要在内核端创建对 dmabuf 的其他引用的驱动程序中是必需的。例如，需要保留 dmabuf 指针的导出器，以便后续导出不会创建新的 dmabuf。

## 保留对象

保留对象提供一种管理与资源相关联的 dma\_fence 对象容器的机制。一个保留对象可以有任意数量的围栏附加到它上面。每个围栏都携带一个使用参数，用于确定由围栏所表示的操作如何使用资源。RCU 机制用于保护从锁定的写侧更新中的围栏的读取访问。

### int reservation\_object\_reserve\_shared(struct reservation\_object \* obj)

保留空间以向保留对象添加共享围栏。

**参数**

struct reservation\_object \* obj

保留对象

**说明**

应在 reservation\_object\_add\_shared\_fence() 调用之前调用。必须使用 obj->lock 调用。

返回成功为零，或返回 -errno

### void reservation\_object\_add\_shared\_fence(struct reservation\_object \* obj, struct dma\_fence \* fence)

向共享位置添加围栏。

**参数**

struct reservation\_object \* obj

保留对象

struct dma\_fence \* fence

要添加的共享围栏

**说明**

向共享位置添加围栏，必须使用 obj->lock 调用，并且必须已经调用了 reservation\_object\_reserve\_shared()。

### void reservation\_object\_add\_excl\_fence(struct reservation\_object \* obj, struct dma\_fence \* fence)

添加独占围栏。

**参数**

struct reservation\_object \* obj

保留对象

struct dma\_fence \* fence

要添加的共享围栏

**说明**

向独占位置添加围栏。obj->lock 必须被保持。

### int reservation\_object\_copy\_fences(struct reservation\_object \* dst, struct reservation\_object \* src)

将所有围栏从 src 复制到 dst。

**参数**

struct reservation\_object \* dst

目标保留对象

struct reservation\_object \* src

源保留对象

**说明**

将所有围栏从 src 复制到 dst。必须保持 dst-lock。

### int reservation\_object\_get\_fences\_rcu(struct reservation\_object \* obj, struct dma\_fence \*\* pfence\_excl, unsigned \* pshared\_count, struct dma\_fence \*\*\* pshared)

获取对象的共享和独占围栏，而无需保持更新侧锁定

**参数**

struct reservation\_object \* obj

保留对象

struct dma\_fence \*\* pfence\_excl

返回的独占围栏（或 NULL）

unsigned \* pshared\_count

返回的共享围栏数量

struct dma\_fence \*\*\* pshared

返回的共享围栏指针数组（数组被 krealloc 到所需的大小，并且必须由调用者释放）

**说明**

检索保留对象中的所有围栏。如果未指定独占围栏的指针，则围栏也放入共享围栏数组中。返回零或 -ENOMEM。

### long reservation\_object\_wait\_timeout\_rcu(struct reservation\_object \* obj, bool wait\_all, bool intr, unsigned long timeout)

等待保留对象的共享和/或独占围栏。

**参数**

struct reservation\_object \* obj

保留对象

bool wait\_all

如果为真，则等待全部栅栏，否则仅等待独占栅栏。

bool intr

如果为真，则等待可中断

unsigned long timeout

jiffies计数器的超时值，如果为零则立即返回

**说明**

返回-ERESTARTSYS表示被中断，返回0表示等待超时，返回大于零的值表示成功。

### bool reservation\_object\_test\_signaled\_rcu（struct reservation\_object \* obj，bool test\_all）

测试保留对象的栅栏是否已发出信号。

**参数**

struct reservation\_object \* obj

保留对象

bool test\_all

如果为真，则测试所有栅栏，否则仅测试独占栅栏

**说明**

如果所有栅栏均已发出信号，则返回true，否则返回false。

### struct reservation\_object\_list

共享栅栏列表

**定义**

struct reservation\_object\_list {

struct rcu\_head rcu;

u32 shared\_count, shared\_max;

struct dma\_fence \_\_rcu \*shared[];

};

**成员**

rcu

用于内部使用

shared\_count

共享栅栏表

shared\_max

用于增加共享栅栏表

shared

共享栅栏表

### struct reservation\_object

保留对象管理缓冲区的栅栏

**定义**

struct reservation\_object {

struct ww\_mutex lock;

seqcount\_t seq;

struct dma\_fence \_\_rcu \*fence\_excl;

struct reservation\_object\_list \_\_rcu \*fence;

struct reservation\_object\_list \*staged;

};

**成员**

lock

更新时加锁

seq

用于管理RCU读端同步的序列计数器

fence\_excl

如果目前存在，则为独占栅栏

fence

当前共享栅栏列表

staged

共享栅栏的被更新版本备份，用于RCU更新

### void reservation\_object\_init(struct reservation\_object \* obj)

初始化保留对象

**参数**

struct reservation\_object \* obj

保留对象

### void reservation\_object\_fini(struct reservation\_object \* obj)

销毁保留对象

**参数**

struct reservation\_object \* obj

保留对象

### struct reservation\_object\_list \* reservation\_object\_get\_list(struct reservation\_object \* obj)

用更新端的锁获取保留对象的共享栅栏列表

**参数**

struct reservation\_object \* obj

保留对象

**说明**

返回共享栅栏列表。不获取栅栏引用。必须持有obj->lock。

### int reservation\_object\_lock(struct reservation\_object \* obj，struct ww\_acquire\_ctx \* ctx)

锁定保留对象

**参数**

struct reservation\_object \* obj

保留对象

struct ww\_acquire\_ctx \* ctx

锁定上下文

**说明**

对保留对象进行独占访问和修改。请注意，锁只针对其他写入者，读取者将在RCU下并发运行写入者。 seqlock用于通知读取者是否与写入者重叠。

由于保留对象可能被多个方参与以未定义的顺序锁定，因此将#ww\_acquire\_ctx传递用于检测到循环时进行解开。请参见ww\_mutex\_lock()和ww\_acquire\_init()。通过将NULL传递为ctx，可以将保留对象锁定本身。

### int reservation\_object\_lock\_interruptible(struct reservation\_object \* obj，struct ww\_acquire\_ctx \* ctx)

锁定保留对象

**参数**

struct reservation\_object \* obj

保留对象

struct ww\_acquire\_ctx \* ctx

锁定上下文

**说明**

以可中断方式锁定保留对象，对其进行独占访问和修改。请注意，锁只针对其他写入者，读取者将在RCU下并发运行写入者。 seqlock用于通知读取者是否与写入者重叠。

由于保留对象可能被多个方参与以未定义的顺序锁定，因此将#ww\_acquire\_ctx传递用于检测到循环时进行解开。请参见ww\_mutex\_lock()和ww\_acquire\_init()。通过将NULL传递为ctx，可以将保留对象锁定本身。

### bool reservation\_object\_trylock(struct reservation\_object \* obj)

尝试锁定保留对象

**参数**

struct reservation\_object \* obj

保留对象

**说明**

尝试对保留对象进行独占访问和修改。请注意，锁只针对其他写入者，读取者将在RCU下并发运行写入者。 seqlock用于通知读取者是否与写入者重叠。

也请注意，由于未提供上下文，因此无法进行死锁保护。

如果成功获取锁定则返回true，否则返回false。

### void reservation\_object\_unlock(struct reservation\_object \* obj)

解锁保留对象

**参数**

struct reservation\_object \* obj

保留对象

**说明**

解锁进行独占访问后的保留对象。

### struct dma\_fence \* reservation\_object\_get\_excl（struct reservation\_object \* obj）

获取保留对象的独占栅栏，并保持更新端锁定

**参数**

struct reservation\_object \* obj

保留对象

**说明**

返回独占栅栏（如果有）。不引用。必须持有obj->lock。

返回独占栅栏或NULL

### struct dma\_fence \* reservation\_object\_get\_excl\_rcu(struct reservation\_object \* obj)

获取保留对象的独占栅栏，而无需锁定

**参数**

struct reservation\_object \* obj

保留对象

**说明**

如果存在独占栅栏，则原子地增加它的引用计数并返回它。

返回独占栅栏或NULL（如果没有）

## DMA栅栏

DMA栅栏（由struct dma\_fence表示）是DMA操作的内核内同步原语，例如GPU渲染、视频编码/解码或在屏幕上显示缓冲区等操作。

使用dma\_fence\_init()初始化栅栏，并使用dma\_fence\_signal()完成栅栏。栅栏与上下文相关联，通过dma\_fence\_context\_alloc()分配，并且同一上下文中的所有栅栏完全有序。

由于栅栏的目的是促进跨设备和跨应用程序的同步，因此有多种使用它的方式

1）单个围栏可以作为同步文件暴露出来，从用户空间作为文件说明符访问，通过调用 sync\_file\_create() 创建。这称为显式围栏，因为用户空间传递显式同步点。

2）一些子系统还具有自己的显式围栏原语，如 drm\_syncobj。与 sync\_file 相比，drm\_syncobj 允许更新底层围栏。

3）然后还有隐式围栏，其中同步点隐含地作为共享 dma\_buf 实例的一部分传递。这样的隐式围栏存储在struct dma\_resv 中，通过 dma\_buf.resv 指针。

### DMA 围栏函数参考

#### u64 dma\_fence\_context\_alloc(unsigned num)

分配一个围栏上下文数组

**参数**

unsigned num

要分配的上下文数量

**说明**

此函数将返回分配的围栏上下文数量的第一个索引。通过将上下文传递给 dma\_fence\_init()，可以使用围栏上下文将 dma\_fence.context 设置为唯一编号。

#### int dma\_fence\_signal\_locked(struct dma\_fence \*fence)

信号围栏完成

**参数**

struct dma\_fence \*fence

要信号的围栏

**说明**

对围栏上的软件回调进行信号完成，这将解除 dma\_fence\_wait() 调用的阻塞，并运行使用 dma\_fence\_add\_callback() 添加的所有回调。可以多次调用，但由于围栏只能从未信号到信号状态，而不能反过来，因此它只会在第一次有效。

与 dma\_fence\_signal() 不同，此函数必须在具有 dma\_fence.lock 的情况下调用。

返回 0 表示成功，当围栏已经被信号时返回负错误值。

#### int dma\_fence\_signal(struct dma\_fence \*fence)

信号围栏完成

**参数**

struct dma\_fence \*fence

要信号的围栏

**说明**

对围栏上的软件回调进行信号完成，这将解除 dma\_fence\_wait() 调用的阻塞，并运行使用 dma\_fence\_add\_callback() 添加的所有回调。可以多次调用，但由于围栏只能从未信号到信号状态，而不能反过来，因此它只会在第一次有效。

返回 0 表示成功，当围栏已经被信号时返回负错误值。

#### signed long dma\_fence\_wait\_timeout(struct dma\_fence \*fence, bool intr, signed long timeout)

睡眠直到围栏得到信号或超时结束

**参数**

struct dma\_fence \*fence

要等待的围栏

bool intr

如果为 true，则进行可中断的等待

signed long timeout

jiffies 中的超时值，或 MAX\_SCHEDULE\_TIMEOUT

**说明**

如果被中断，则返回 -ERESTARTSYS，如果等待超时，则返回 0，如果成功则返回剩余的超时时间（以 jiffies 为单位）。可能会返回其他错误值，取决于自定义实现。

对此围栏执行同步等待。假定调用者直接或间接地（在保留和提交之间的缓冲管理器）持有指向围栏的引用，否则围栏可能在返回之前被释放，导致未定义的行为。

参见 dma\_fence\_wait() 和 dma\_fence\_wait\_any\_timeout()。

#### void dma\_fence\_release(struct kref \*kref)

围栏的默认释放函数

**参数**

struct kref \*kref

dma\_fence.recfount

**说明**

这是 dma\_fence 的默认释放函数。驱动程序不应直接调用此函数，而应调用 dma\_fence\_put()。

#### void dma\_fence\_free(struct dma\_fence \*fence)

dma\_fence 的默认释放函数。

**参数**

struct dma\_fence \*fence

要释放的围栏

**说明**

这是 dma\_fence\_ops.release 的默认实现。它在围栏上调用 kfree\_rcu()。

#### void dma\_fence\_enable\_sw\_signaling(struct dma\_fence \*fence)

启用围栏的信号

**参数**

struct dma\_fence \*fence

要启用的围栏

**说明**

这将请求启用 sw 信号，以便尽快使围栏完成。这将在内部调用 dma\_fence\_ops.enable\_signaling。

#### int dma\_fence\_add\_callback(struct dma\_fence \*fence, struct dma\_fence\_cb \*cb, dma\_fence\_func\_t func)

添加回调以在围栏被信号时调用

**参数**

struct dma\_fence \*fence

要等待的围栏

struct dma\_fence\_cb \*cb

要注册的回调

dma\_fence\_func\_t func

要调用的函数

**说明**

向围栏添加软件回调。调用者应保留对围栏的引用。

dma\_fence\_add\_callback() 将初始化 cb，不需要调用者进行初始化。可以向围栏注册任意数量的回调，但回调只能一次注册到一个围栏。

如果围栏已经被信号，则此函数将返回 -ENOENT（并且不调用回调）。

请注意，回调可能从原子上下文或中断上下文调用。

在成功的情况下返回 0，在围栏已被信号的情况下返回 -ENOENT，在出错的情况下返回 -EINVAL。

#### int dma\_fence\_get\_status(struct dma\_fence \*fence)

返回完成时的状态

**参数**

struct dma\_fence \*fence

要查询的 dma\_fence

**说明**

这会包装 dma\_fence\_get\_status\_locked()，返回已信号围栏上的错误状态条件。有关更多详细信息，请参见 dma\_fence\_get\_status\_locked()。

如果围栏尚未被信号，则返回 0；如果围栏已被信号但没有错误条件，则返回 1；如果围栏以 err 完成，则返回负错误代码。

#### bool dma\_fence\_remove\_callback(struct dma\_fence \*fence, struct dma\_fence\_cb \*cb)

从信令列表中删除回调函数

**参数**

struct dma\_fence \*fence

要等待的栅栏

struct dma\_fence\_cb \*cb

要删除的回调函数

**说明**

从栅栏中删除已经排队的回调函数。如果成功删除回调函数，则此函数返回真，否则如果栅栏已经被触发，则返回假。

警告只有在确切知道自己在做什么时才应该取消回调函数，因为很容易发生死锁和竞争条件。因此，它只应在硬件锁定恢复时进行，同时保持对栅栏的引用。

如果cb在之前没有使用dma\_fence\_add\_callback()添加到fence中，则其行为未定义。

#### signed long dma\_fence\_default\_wait(struct dma\_fence \*fence, bool intr, signed long timeout)

默认睡眠直到栅栏触发或超时

**参数**

struct dma\_fence \*fence

要等待的栅栏

bool intr

如果为真，则进行可中断的等待

signed long timeout

jiffies中的超时值或MAX\_SCHEDULE\_TIMEOUT

**说明**

如果被中断，则返回-ERESTARTSYS，如果等待超时，则返回0，否则在成功时返回剩余的jiffies超时时间。如果超时为零，则如果栅栏已经发出信号，则返回为1，以与其他带有jiffies超时的函数保持一致。

#### signed long dma\_fence\_wait\_any\_timeout(struct dma\_fence \*\*fences, uint32\_t count, bool intr, signed long timeout, uint32\_t \*idx)

睡眠直到任何栅栏被触发或超时

**参数**

struct dma\_fence \*\*fences

要等待的栅栏数组

uint32\_t count

要等待的栅栏数

bool intr

如果为真，则进行可中断的等待

signed long timeout

jiffies中的超时值或MAX\_SCHEDULE\_TIMEOUT

uint32\_t \*idx

用于存储第一个发出信号的栅栏的索引，仅在返回为正时有意义

**说明**

对于数组中的第一个栅栏进行同步等待，调用者需要持有数组中所有栅栏的引用，否则在返回之前可能会释放某个栅栏，导致未定义的行为。

另请参见dma\_fence\_wait()和dma\_fence\_wait\_timeout()。

#### void dma\_fence\_init(struct dma\_fence \*fence, const struct dma\_fence\_ops \*ops, spinlock\_t \*lock, u64 context, u64 seqno)

初始化自定义栅栏。

**参数**

struct dma\_fence \*fence

要初始化的栅栏

const struct dma\_fence\_ops \*ops

关于此栅栏操作的dma\_fence\_ops

spinlock\_t \*lock

用于锁定此栅栏的irqsafe自旋锁

u64 context

此栅栏运行的执行上下文

u64 seqno

此上下文中线性递增的序列号

**说明**

初始化已分配的栅栏，调用者在提交此栅栏后不必保留其引用计数，但如果调用dma\_fence\_ops.enable\_signaling，则需要再次持有引用计数。

context和seqno用于容易比较不同栅栏，可以使用dma\_fence\_later()简单地检查哪个栅栏稍后被发出信号。

#### struct dma\_fence

软件同步原语

**定义**

struct dma\_fence {

struct kref refcount;

const struct dma\_fence\_ops \*ops;

struct rcu\_head rcu;

struct list\_head cb\_list;

spinlock\_t \*lock;

u64 context;

unsigned seqno;

unsigned long flags;

ktime\_t timestamp;

int error;

};

**成员**

refcount

这个围栏的重新计算

ops

与此围栏关联的 dma\_fence\_ops

rcu

用于使用 kfree\_rcu 释放栅栏

cb\_list

要调用的所有回调的列表

lock

spin\_lock\_irqsave 用于锁定

context

此栅栏所属的执行上下文，由返回 [dma\_fence\_context\_alloc()](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_fence_context_alloc" \o "dma_fence_context_alloc)

seqno

可以比较执行上下文中此栅栏的序列号，以决定稍后向哪个栅栏发出信号。

flags

下面定义了 DMA\_FENCE\_FLAG\_\* 的掩码

timestamp

围栏发出信号时的时间戳。

error

可选，仅当 < 0 时有效，必须在调用 dma\_fence\_signal 之前设置，表示 fence 已完成但有错误。

**说明**

flags成员必须使用适当的原子操作（bit\_\*）进行操作和读取，因此大部分时候不需要使用自旋锁。

DMA\_FENCE\_FLAG\_SIGNALED\_BIT-栅栏已经发出信号DMA\_FENCE\_FLAG\_TIMESTAMP\_BIT-记录用于栅栏发出信号的时间戳DMA\_FENCE\_FLAG\_ENABLE\_SIGNAL\_BIT-可能已调用enable\_signalingDMA\_FENCE\_FLAG\_USER\_BITS-未使用比特的开头，可以由栅栏实现者用于其自己的目的。不同的栅栏实现者可以以不同的方式使用，因此不要依赖于此。

由于使用了原子的位操作，因此不能保证这是正确的。特别是，如果设置了该位，但在设置该位之前调用了dma\_fence\_signal，则它将能够在调用enable\_signaling之前设置DMA\_FENCE\_FLAG\_SIGNALED\_BIT。在设置DMA\_FENCE\_FLAG\_ENABLE\_SIGNAL\_BIT后，在设置DMA\_FENCE\_FLAG\_SIGNALED\_BIT之后添加对DMA\_FENCE\_FLAG\_SIGNALED\_BIT的检查可以消除这种竞争，并确保在调用dma\_fence\_signal之后，任何enable\_signaling调用均已完成或根本未调用。。

#### struct dma\_fence\_cb

dma\_fence\_add\_callback()的回调函数

**定义**

struct dma\_fence\_cb {

struct list\_head node;

dma\_fence\_func\_t func;

};

**成员**

node

用于将此结构附加到fence::cb\_list的dma\_fence\_add\_callback()函数。

func

调用的dma\_fence\_func\_t函数。

**说明**

此结构将由dma\_fence\_add\_callback()函数进行初始化，可以通过将dma\_fence\_cb嵌入到另一个结构中来传递附加数据。

#### struct dma\_fence\_ops

fence实现的操作

**定义**

struct dma\_fence\_ops {

const char \* (\*get\_driver\_name)(struct dma\_fence \*fence);

const char \* (\*get\_timeline\_name)(struct dma\_fence \*fence);

bool (\*enable\_signaling)(struct dma\_fence \*fence);

bool (\*signaled)(struct dma\_fence \*fence);

signed long (\*wait)(struct dma\_fence \*fence, bool intr, signed long timeout);

void (\*release)(struct dma\_fence \*fence);

void (\*fence\_value\_str)(struct dma\_fence \*fence, char \*str, int size);

void (\*timeline\_value\_str)(struct dma\_fence \*fence, char \*str, int size);

};

**成员**

get\_driver\_name

返回驱动程序名称。这是一个回调函数，允许驱动程序在运行时计算名称，而无需对每个fence永久存储，或者构建一些缓存。

此回调是强制性的。

get\_timeline\_name

返回此fence属于的上下文的名称。这是一个回调函数，允许驱动程序在运行时计算名称，而无需对每个fence永久存储，或者构建一些缓存。

此回调是强制性的。

enable\_signaling

启用fence的软件信号。

对于具有hw->hw信号功能的fence实现，它们可以实现此操作以启用必要的中断，或向cmdstream插入命令等，以避免仅需要hw->hw同步的常见情况下进行这些昂贵的操作。这在第一个dma\_fence\_wait()或dma\_fence\_add\_callback()路径中调用，以让fence实现知道还有另一个驱动程序在等待信号(即hw->sw情况)。

这个函数可以从原子上下文调用，但不能从irq上下文调用，因此可以使用普通的自旋锁。

返回false表示fence已经通过，或发生了使其无法启用信号的故障。True表示成功启用。

dma\_fence.error可能在enable\_signaling中设置，但仅当返回false时。

由于许多实现可以在enable\_signaling之前调用dma\_fence\_signal()，所以存在竞争窗口，在这个窗口中dma\_fence\_signal()可能会导致最终fence引用被释放并且其内存被释放。为避免这种情况，此回调的实现应使用dma\_fence\_get()来获取自己的引用，以便在fence被信号(例如通过中断处理程序)时释放引用。

此回调是可选的。如果回调不在，则驱动程序必须始终启用信号。

signaled

查看fence是否已经发出信号，作为dma\_fence\_wait()或dma\_fence\_add\_callback()的快速路径优化。请注意，此回调不需要做出除了一个fence曾经表明为已发信号必须始终返回true之外的任何保证。即使fence已经完成，此回调也可能返回false，在此情况下，信息尚未传播到整个系统。另请参阅dma\_fence\_is\_signaled()。

如果返回true，则可能设置dma\_fence.error。

此回调是可选的。

wait

自定义等待实现，默认为dma\_fence\_default\_wait()如果未设置。

已过期，不应由新实现使用。只有现有实现需要其硬件重置过程的特殊处理才会使用它。

必须返回-ERESTARTSYS，如果等待是intr=true并且等待被中断，以及如果fence已经发出信号，则返回剩余的jiffies，或如果等待超时则返回0。还可以在自定义实现中返回其他错误值，这些应被视为fence已发出信号。例如，硬件锁定可能会报告这样的情况。

release

在fence销毁时调用以释放其他资源。可以从irq上下文调用。此回调是可选的。如果为空，则默认实现为dma\_fence\_free()。

fence\_value\_str

回调填充特定于此fence的自由形式调试信息，如序列号。

此回调是可选的。

timeline\_value\_str

以字符串的形式填充时间线的当前值，例如序列号。请注意，传递给此函数的特定fence应该不重要，驱动程序应该只使用它来查找相应的时间线结构。

#### void dma\_fence\_put(struct dma\_fence \*fence)

减少fence的引用计数。

**参数**

struct dma\_fence \*fence

要减少引用计数的fence

#### struct dma\_fence \*dma\_fence\_get(struct dma\_fence \*fence)

增加fence的引用计数。

**参数**

struct dma\_fence \*fence

要增加引用计数的fence

**说明**

返回引用计数增加1的同一fence。

#### struct dma\_fence \*dma\_fence\_get\_rcu(struct dma\_fence \*fence)

使用rcu读锁从dma\_resv\_list获取栅栏。

**参数**

struct dma\_fence \*fence

要增加引用计数的栅栏

**说明**

如果无法获得引用计数，则返回NULL或栅栏。

#### struct [dma\_fence](https://www.kernel.org/doc/html/latest/driver-api/dma-buf.html" \l "c.dma_fence" \o "dma_fence) \*dma\_fence\_get\_rcu\_safe(struct [dma\_fence](https://www.kernel.org/doc/html/latest/driver-api/dma-buf.html" \l "c.dma_fence" \o "dma_fence) \_\_rcu \*\*fencep)

获取RCU跟踪围栏的引用

**参数**

struct dma\_fence \_\_rcu \*\*fencep

指向增加引用计数的围栏的指针

**说明**

如果无法获取引用计数，则返回NULL或围栏。此函数处理获取在RCU优化期间（例如使用SLAB\_TYPESAFE\_BY\_RCU）内可能重新分配的围栏的引用，只要调用者在围栏指针上使用RCU即可。

另一种机制是使用序列锁保护一堆围栏，例如由struct dma\_resv使用。当使用序列锁时，在获取对围栏的引用之前和之后必须获取并检查序列锁（如此处所示）。

调用者需要持有RCU读锁。

#### bool dma\_fence\_is\_signaled\_locked(struct dma\_fence \*fence)

返回围栏是否已被信号的指示。

**参数**

struct dma\_fence \*fence

要检查的围栏

**说明**

如果围栏已经被信号，则返回true；如果未被信号，则返回false。由于此函数不会启用信号，因此无法保证始终会返回true，除非之前没有调用dma\_fence\_add\_callback()、dma\_fence\_wait()或dma\_fence\_enable\_sw\_signaling()。

此函数要求持有dma\_fence.lock。

另请参见dma\_fence\_is\_signaled()。

#### bool dma\_fence\_is\_signaled(struct dma\_fence \*fence)

返回围栏是否已被信号的指示。

**参数**

struct dma\_fence \*fence

要检查的围栏

**说明**

如果围栏已经被信号，则返回true；如果未被信号，则返回false。由于此函数不会启用信号，因此无法保证始终会返回true，除非之前没有调用dma\_fence\_add\_callback()、dma\_fence\_wait()或dma\_fence\_enable\_sw\_signaling()。

建议对于seqno fences，当操作完成时调用dma\_fence\_signal，这样可以通过在调用硬件特定的等待指令之前检查此函数的返回来防止在发行时间和使用时间之间出现环绕问题。

另请参见dma\_fence\_is\_signaled\_locked()。

#### bool \_\_dma\_fence\_is\_later(u32 f1, u32 f2)

返回f1是否在时间上晚于f2

**参数**

u32f1

第一个栅栏的seqno

u32 f2

来自相同上下文的第二个栅栏的seqno

const struct dma\_fence\_ops \*ops

与seqno关联的dma\_fence\_ops

**说明**

如果f1时间上晚于f2，则返回true。两个围栏必须来自相同的上下文，因为seqno在跨上下文中不常见。

#### bool dma\_fence\_is\_later(struct dma\_fence \*f1, struct dma\_fence \*f2)

返回f1是否在时间上晚于f2

**参数**

struct dma\_fence \*f1

来自相同上下文的第一个围栏

struct dma\_fence \*f2

来自相同上下文的第二个围栏

**说明**

如果f1时间上晚于f2，则返回true。两个围栏必须来自相同的上下文，因为seqno在跨上下文中不常见。

#### struct dma\_fence \*dma\_fence\_later(struct dma\_fence \*f1, struct dma\_fence \*f2)

返回时间上晚的围栏

**参数**

struct dma\_fence \*f1

来自相同上下文的第一个围栏

struct dma\_fence \*f2

来自相同上下文的第二个围栏

**说明**

如果两个围栏均信号，返回NULL；否则返回最后一个将被信号的围栏。两个围栏必须来自相同的上下文，因为seqno在跨上下文中不会被重用。

#### int dma\_fence\_get\_status\_locked(struct dma\_fence \*fence)

在完成时返回状态

**参数**

struct dma\_fence \*fence

要查询的dma\_fence

**说明**

驱动程序可以在信号围栏之前提供可选的错误状态条件（以指示围栏是否因错误而非成功而完成）。仅当围栏已被信号时，状态条件的值才有效，dma\_fence\_get\_status\_locked()在报告错误状态之前首先检查信号状态。

如果围栏尚未被信号，则返回0；如果围栏已被信号而没有错误条件，则返回1；如果围栏已在err中完成，则返回负错误代码。

#### void dma\_fence\_set\_error(struct dma\_fence \*fence, int error)

在围栏上标记错误条件

**参数**

struct dma\_fence \*fence

dma\_fence

int error

要存储的错误

**说明**

驱动程序可以在信号围栏之前提供可选的错误状态条件，以指示围栏因错误而非成功而完成。这必须在发信号之前设置（以便在唤醒信号回调上的任何等待者之前该值可见）。此助手程序存在以帮助捕获错误设置#dma\_fence.error。

#### signed long dma\_fence\_wait(struct dma\_fence \*fence, bool intr)

睡眠，直到围栏被信号

**参数**

struct dma\_fence \*fence

要等待的围栏

bool intr

如果为true，则执行可中断等待

**说明**

如果被信号中断，则返回-ERESTARTSYS；如果围栏被信号，则返回0。其他错误值可能会在自定义实现中返回。

对此围栏执行同步等待。假定调用者直接或间接地持有对围栏的引用，否则在返回之前可能释放围栏，导致未定义的行为。

请参见

dma\_fence\_wait\_timeout()和dma\_fence\_wait\_any\_timeout()。

### 序号硬件栅栏

#### struct seqno\_fence \* to\_seqno\_fence(struct dma\_fence \* fence)

将栅栏转换为seqno\_fence

**参数**

struct dma\_fence \* fence

要转换为seqno\_fence的栅栏

**说明**

如果栅栏不是seqno\_fence，则返回NULL，否则返回seqno\_fence。

#### void seqno\_fence\_init(struct seqno\_fence \* fence，spinlock\_t \* lock，struct dma\_buf \* sync\_buf，uint32\_t context，uint32\_t seqno\_ofs，uint32\_t seqno，enum seqno\_fence\_condition cond，const struct dma\_fence\_ops \* ops)

初始化seqno栅栏

**参数**

struct seqno\_fence \* fence

要初始化的seqno\_fence

spinlock\_t \* lock

用于栅栏的自旋锁指针

struct dma\_buf \* sync\_buf

包含要在其上进行信号传递的内存位置的缓冲区

uint32\_t context

此栅栏所属的执行上下文

uint32\_t seqno\_ofs

在sync\_buf内的偏移量

uint32\_t seqno

要发出信号的序号

enum seqno\_fence\_condition cond

栅栏等待条件

const struct dma\_fence\_ops \* ops

用于在此seqno栅栏上执行操作的fence\_ops

**说明**

此函数使用传递的参数初始化struct seqno\_fence，并对在fence销毁时释放的sync\_buf进行引用计数。

seqno\_fence是可以在启用信号传递时由软件完成的dma\_fence，但也会在（s32）（（sync\_buf）[seqno\_ofs] - seqno）> = 0为真时完成。

该seqno\_fence将在同步\_buf上占用引用计数，直到销毁为止，但如果其中一个调用程序引用它，则sync\_buf的实际生命周期可能更长。

某些硬件具有在命令流中插入此类型的等待条件的指令，因此不需要来自软件的干预。可以在完成之前破坏此类栅栏，但可以对sync\_buf dma-buf进行引用。建议重复使用相同的dma-buf用于sync\_buf，因为映射或取消映射sync\_buf到设备的vm可能很昂贵。

建议seqno\_fence的创建者在销毁之前调用dma\_fence\_signal()。这将防止在发行时与检查时间的wraparound可能存在的问题，因为用户可以在提交指令供硬件等待栅栏之前检查dma\_fence\_is\_signaled()。但是，当未调用ops.enable\_signaling时，不必尽快执行此操作，只需在存在任何序号环绕的真正危险之前即可。

### DMA栅栏数组

#### struct dma\_fence\_array \* dma\_fence\_array\_create(int num\_fences，struct dma\_fence \*\* fences，u64 context，unsigned seqno，bool signal\_on\_any)

创建自**定义**栅栏数组

**参数**

int num\_fences

[在]要在数组中添加的栅栏数

struct dma\_fence \*\* fences

[in]包含栅栏的数组

u64 context

[在]要使用的栅栏上下文

unsigned seqno

[在]要使用的顺序号

bool signal\_on\_any

[在]信号任何栅栏数组上

**说明**

分配dma\_fence\_array对象并使用dma\_fence\_init（）初始化基本栅栏。如果出现错误，则返回NULL。

调用者应使用num\_fences大小分配fences数组，并将其填充上要添加到对象中的栅栏。该数组的所有权已被接管，并在释放时对每个栅栏使用dma\_fence\_put()。

如果signal\_on\_any为true，则栅栏数组在任何栅栏中发出信号，否则在数组中所有栅栏都发出信号时发出信号。

#### bool dma\_fence\_match\_context(struct dma\_fence \* fence，u64 context)

检查所有栅栏是否来自给定上下文

**参数**

struct dma\_fence \* fence

[在]栅栏或栅栏数组

u64 context

[在]检查所有栅栏相对于的栅栏上下文

**说明**

将提供的栅栏或栅栏数组中的所有栅栏与给定的上下文进行比较。如果任何栅栏来自不同的上下文，则返回false。

#### struct dma\_fence\_array\_cb

栅栏数组的回调辅助程序

**定义**

struct dma\_fence\_array\_cb {

struct dma\_fence\_cb cb;

struct dma\_fence\_array \*array;

};

**成员**

cb

用于信号传递的栅栏回调结构

array

对父栅栏数组对象的引用

#### struct dma\_fence\_array

代表栅栏阵列的围栏

**定义**

struct dma\_fence\_array {

struct dma\_fence base;

spinlock\_t lock;

unsigned num\_fences;

atomic\_t num\_pending;

struct dma\_fence \*\*fences;

struct irq\_work work;

};

**成员**

base

栅栏基类

lock

用于处理栅栏的自旋锁

num\_fences

数组中的栅栏数

num\_pending

仍悬而未决的阵列中的栅栏

fences

栅栏数组

work

内部irq\_work函数

#### bool dma\_fence\_is\_array（struct [dma\_fence](https://www.kernel.org/doc/html/v4.19/driver-api/dma-buf.html" \l "c.dma_fence" \o "dma_fence) \* fence）

检查栅栏是否来自数组子类

**参数**

struct dma\_fence \* fence

栅栏测试

**说明**

如果它是 dma\_fence\_array 则返回 true，否则返回 false。

#### struct dma\_fence\_array \* to\_dma\_fence\_array(struct dma\_fence \* fence)

将栅栏转换为dma\_fence\_array

**参数**

struct dma\_fence \* fence

要转换为dma\_fence\_array的栅栏

**说明**

如果栅栏不是dma\_fence\_array，则返回NULL，否则返回dma\_fence\_array。

### DMA栅栏uABI /同步文件

#### struct sync\_file \* sync\_file\_create(struct dma\_fence \* fence)

创建同步文件

**参数**

struct dma\_fence \* fence

要添加到sync\_fence的栅栏

**说明**

创建containg fence的sync\_file。如果成功，则此函数会获取并对新创建的sync\_file的fence进行附加引用。 sync\_file可使用fput（sync\_file->file）释放。在错误的情况下返回sync\_file或NULL。

#### struct dma\_fence \* sync\_file\_get\_fence(int fd)

获取与sync\_file fd相关的围栏

**参数**

int fd

sync\_file fd以获取围栏

**说明**

确保fd引用有效的sync\_file并返回表示sync\_file中所有围栏的围栏。发生错误时返回NULL。

#### struct sync\_file

同步文件导出到用户空间

**定义**

struct sync\_file {

struct file \*file;

char user\_name[32];

#ifdef CONFIG\_DEBUG\_FS;

struct list\_head sync\_file\_list;

#endif;

wait\_queue\_head\_t wq;

unsigned long flags;

struct dma\_fence \*fence;

struct dma\_fence\_cb cb;

};

**成员**

file

代表此围栏的文件

user\_name

由用户空间提供的同步文件名称，用于合并围栏。否则通过驱动程序回调生成（在这种情况下，整个数组为0）。

sync\_file\_list

全局文件列表的成员资格

wq

围栏信号等待队列

flags

sync\_file的标志

fence

具有sync\_file中围栏的围栏

CB

围栏回调信息

**说明**

标志POLL\_ENABLED用户空间当前是否处于poll（）状态

# 设备链接

默认情况下，驱动程序核心仅强制执行设备层次结构中父/子关系之间产生的设备之间的依赖关系在暂停，恢复或关闭系统时，根据此关系对设备进行排序，即始终在其父级之前暂停并且在其子级之前恢复父级。

有时需要表示除了父/子关系之外的设备依赖关系，例如在兄弟姐妹之间，并且由驱动程序核心自动处理它们。

其次，默认情况下，驱动程序核心不强制执行任何驱动程序存在依赖关系，即必须在另一个设备可以正确探测或运行之前，一个设备必须绑定到驱动程序。

通常这两种依赖类型同时出现，因此一个设备在驱动程序存在和暂停/恢复和关闭排序方面都依赖于另一个设备。

设备链接允许在驱动程序核心中表示这些依赖关系。

在其标准或托管形式中，设备链接结合了两种依赖关系类型它保证正确的暂停/恢复和关闭排序在“供应商”设备和其“使用者”设备之间，并且保证供应商上的驱动程序存在。在供应商绑定到驱动程序之前不会探测使用者设备，并且在供应商取消绑定之前它们将被取消绑定。

当供应商上的驱动程序存在是无关紧要的，仅需要正确的暂停/恢复和关闭排序时，可以使用DL\_FLAG\_STATELESS标志简单地设置设备链接。换句话说，在供应商上施加驱动程序存在是可选的。

另一个可选功能是运行时PM集成通过在添加设备链接时设置DL\_FLAG\_PM\_RUNTIME标志，告诉PM核心在何时以及任何时候运行时恢复供应商并保持其处于活动状态，只要使用者处于运行时恢复状态。

## 用法

添加设备链接的最早时间是在为供应商调用device\_add（）并为使用柿子调用device\_initialize（）之后。

添加它们以后是合法的，但必须注意系统保持一致的状态例如，不能在暂停/恢复过渡的中间添加设备链接，因此需要使用lock\_system\_sleep（）防止这种过渡的开始，或者需要从保证不会并行运行到暂停/恢复过渡的函数中添加设备链接，例如从设备->探测回调或引导时PCI奇怪的时间。

另一个不一致状态的例子是表示驱动程序存在依赖关系的设备链接，但是在供应商开始探测之前，从使用者的->探测回调中添加的设备链接如果驱动程序核心更早了解设备链接，则不会首先探测使用者。因此，使用者有责任在添加链接后检查供应商的存在，并推迟对非存在的探测。

->probe如果在供应商或消费者驱动程序的回调中添加设备链接，通常会在其->remove回调中删除它以实现对称。这样，如果驱动程序被编译为模块，则设备链接会在模块加载时添加，并在卸载时按顺序删除。适用于设备链接添加的相同限制（例如，排除并行挂起/恢复转换）同样适用于删除。

设备链接添加时可以指定几个标志，其中两个已经在上面提到了DL\_FLAG\_STATELESS用于表示不需要驱动程序存在的依赖关系（但只需要正确的挂起/恢复和关闭顺序），DL\_FLAG\_PM\_RUNTIME用于表示需要运行时电源管理集成。

另外两个标志专门针对使用情况，其中设备链接是从消费者的->probe回调中添加的情况DL\_FLAG\_RPM\_ACTIVE可以指定为在消费者运行时挂起之前运行时恢复供应商并防止其挂起。 DL\_FLAG\_AUTOREMOVE\_CONSUMER会在消费者未能探测或稍后解除绑定时自动清除设备链接。

同样，当设备链接是从供应商的->probe回调添加时，DL\_FLAG\_AUTOREMOVE\_SUPPLIER会在供应商无法探测或稍后取消绑定时自动清除设备链接。

## 限制

驱动程序作者应注意，对于受控设备链接的驱动程序存在依赖关系（即在链接添加中未指定DL\_FLAG\_STATELESS），可能会导致消费者的探测无限期地被推迟。如果消费者需要在某个特定的initcall级别之前进行探测，则可能会出现问题。更糟糕的是，如果供应商驱动程序被列入黑名单或丢失，则无法探测消费者。

有时，驱动程序依赖于可选资源。当这些资源不存在时，它们能够以降级模式（功能集或性能降低）运行。例如，可以在SPI控制器中使用DMA引擎或以PIO模式工作。控制器可以在探测时确定可选资源的存在，但是，在非存在情况下，无法知道它们是否会在不久的将来变得可用（由于供应商驱动程序的探测），或永远无法知道。因此，无法确定是否推迟探测。可以在探测后通知驱动程序可选资源何时变为可用，但这对于驱动程序来说成本很高，因为基于可用性在运行时在操作模式之间切换将比基于探测推迟的机制复杂得多。无论如何，可选资源超出了设备链接的范围。

## 示例

在总线主设备旁存在一个MMU设备，两者都在同一个电源域中。 MMU实现了总线主设备的DMA地址转换，并且当总线主设备处于活动状态时，应运行时挂起并保持活动状态。在绑定MMU之前，不应绑定总线主设备的驱动程序。为实现这一点，可以从总线主设备（消费者）到MMU设备（供应商）添加具有运行时PM集成的设备链接。与运行时PM相关的效果与MMU是母设备的主设备相同。

两个设备共享同一电源域，通常建议使用struct dev\_pm\_domain或struct generic\_pm\_domain，但这些不是独立的设备，它们恰好共享电源开关，而是MMU设备服务于总线主设备且在没有总线主设备时是无用的。设备链接创建了设备之间的合成层次关系，因此更加适合。

一个Thunderbolt主机控制器包含一些PCIe热插拔端口和一个NHI设备来管理PCIe交换机。从系统睡眠中恢复时，在热插拔端口重新启动之前，NHI设备需要重新建立与附加设备的PCI通道。如果热插拔端口是NHI的子节点，则PM核心会自动强制执行这个恢复顺序，但不幸的是它们是阿姨。解决方案是从热插拔端口（使用者）到NHI设备（供应者）添加设备链接。对于这个用例，不需要驱动程序的存在依赖关系。

混合图形笔记本电脑中的离散GPU通常具有用于HDMI/DP音频的HDA控制器。在设备层次结构中，HDA控制器是VGA设备的兄弟，但两者共享相同的电源域，只有在HDMI/DP显示器连接到VGA设备时才需要HDA控制器。从HDA控制器（使用者）到VGA设备（供应者）的设备链接恰当地代表了这种关系。

ACPI允许通过\_DEP对象定义设备启动顺序。一个典型的例子是当一个设备的ACPI电源管理方法以I2C访问的方式实现，并且要求一个特定的I2C控制器出现并且对于所涉及的设备的电源管理工作正常运行。

在一些SoC中，显示器、视频编解码器和视频处理IP核对处理暴发访问和压缩/解压缩的透明存储器访问IP核存在功能依赖。

## 替代方案

可以使用struct dev\_pm\_domain来覆盖总线、类或设备类型回调。它用于共享单个开/关开关的设备，但它不能保证特定的挂起/恢复顺序，这需要单独实现。它本身也不跟踪所涉及设备的运行时PM状态，并仅在全部处于运行时暂停状态时才关闭电源开关。此外，它不能用于强制执行特定的关闭顺序或驱动程序存在依赖性。

struct generic\_pm\_domain比设备连接更加笨重，不允许关闭顺序或驱动程序存在依赖性。它也不能在ACPI系统上使用。

## 实现

设备层次结构（顾名思义）是一棵树，添加设备链接后变成了一个有向无环图。

在挂起/恢复期间，这些设备的顺序由dpm\_list决定。在关闭期间，由devices\_kset决定。如果没有设备链接，这两个列表是设备树的一个展平的一维表示，设备被放在所有祖先后面。这是通过自上而下遍历ACPI命名空间或OpenFirmware设备树并在发现设备时将它们附加到列表中来实现的。

添加设备链接后，列表需要满足一个附加约束，即设备放置在所有它的供应者后面，递归地。为确保这一点，在添加设备链接时，将使用者和其下面的整个子图（使用者的所有子节点和消费者）移动到列表的末尾。（从device\_link\_add()调用device\_reorder\_to\_tail()。）

为了防止将依赖环引入图中，在设备链接添加时进行验证，以确保供应者不依赖使用者或使用者的任何子节点或消费者。（从device\_link\_add()调用device\_is\_dependent()。）如果违反了这个约束，device\_link\_add()将返回NULL，并记录一个警告。

值得注意的是，这也阻止了从父设备到子设备的设备链接添加。但是，反过来是被允许的，即从子设备到父设备的设备链接。由于驱动程序核心已经保证父设备和子设备之间的正确挂起/恢复和关闭顺序，如果需要驱动程序存在依赖性，这样的设备链接只有在这种情况下才有意义。在这种情况下，驱动程序作者应该仔细权衡是否设备链接是适合这个目的的正确工具。一个更合适的方法可能是简单地使用延迟探测或添加一个设备标志，导致父驱动程序在子驱动程序之前进行探测。

## 状态机

### enum device\_link\_state

**常量**

DL\_STATE\_NONE

未跟踪驱动程序的存在。

DL\_STATE\_DORMANT

供应商/使用者驱动程序均不在场。

DL\_STATE\_AVAILABLE

供应者驱动程序存在，但使用者不存在。

DL\_STATE\_CONSUMER\_PROBE

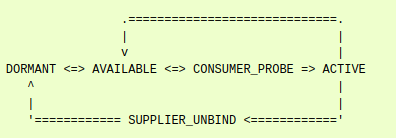
使用者正在探测（供应者驱动程序存在）。

DL\_STATE\_ACTIVE

供应者和使用者驱动程序都存在。

DL\_STATE\_SUPPLIER\_UNBIND

供应者驱动程序正在解绑。



设备链接的初始状态由device\_link\_add()自动确定，基于供应商和使用者的驱动程序存在。如果在探测任何设备之前创建链接，它将被设置为DL\_STATE\_DORMANT。

当供应商设备绑定到驱动程序时，其消费者的链接进展到DL\_STATE\_AVAILABLE状态。(从driver\_bound()调用device\_links\_driver\_bound()。)

在探测消费者设备之前，验证供应商驱动程序的存在，方法是通过检查消费者设备是否不在wait\_for\_suppliers列表中，并通过检查到供应商的链接是否处于DL\_STATE\_AVAILABLE状态来实现。链接的状态更新为DL\_STATE\_CONSUMER\_PROBE。(从really\_probe()调用device\_links\_check\_suppliers()。)这可以防止供应商取消绑定。(从device\_links\_unbind\_consumers()调用wait\_for\_device\_probe()。)

如果探测失败，则链接回退到DL\_STATE\_AVAILABLE状态。(从really\_probe()调用device\_links\_no\_driver()。)如果探测成功，则链接到供应商将进展到DL\_STATE\_ACTIVE状态。(从driver\_bound()调用device\_links\_driver\_bound()。)

当消费者的驱动程序稍后被移除时，链接到供应商将回退到DL\_STATE\_AVAILABLE状态。(从\_\_device\_release\_driver()调用device\_links\_driver\_cleanup()中的\_\_device\_links\_no\_driver()。)

在移除供应商的驱动程序之前，将未绑定到驱动程序的消费者的链接更新为DL\_STATE\_SUPPLIER\_UNBIND。(从\_\_device\_release\_driver()调用device\_links\_busy()。)这可以防止消费者的绑定。(从really\_probe()调用device\_links\_check\_suppliers()。)已绑定的消费者将被释放其驱动程序；正在探测的消费者将等待直到完成。(从\_\_device\_release\_driver()调用device\_links\_unbind\_consumers()。)一旦所有链接到消费者的链接处于DL\_STATE\_SUPPLIER\_UNBIND状态，供应商驱动程序将被释放，并且链接将恢复到DL\_STATE\_DORMANT状态。(从\_\_device\_release\_driver()调用device\_links\_driver\_cleanup()。)

## API

### struct device\_link \* device\_link\_add(struct device \* consumer, struct device \* supplier, u32 flags)

创建两个设备之间的链接。

**参数**

struct device \* consumer

链接的消费者端。

struct device \* supplier

链接的供应商端。

u32 flags

链接标志。

**说明**

调用者负责将链接创建与运行时 PM 的适当同步。首先，设置 DL\_FLAG\_PM\_RUNTIME 标志将导致运行时 PM 框架考虑链接。其次，如果除此之外还设置了 DL\_FLAG\_RPM\_ACTIVE 标志，则供应商设备将被强制进入活动的“metastate”，并在创建链接时引用计数。如果未设置 DL\_FLAG\_PM\_RUNTIME，则 DL\_FLAG\_RPM\_ACTIVE 将被忽略。

如果设置了 DL\_FLAG\_AUTOREMOVE\_CONSUMER，则在消费者设备驱动程序取消绑定时，链接将自动删除。同时设置 DL\_FLAG\_AUTOREMOVE\_CONSUMER 和 DL\_FLAG\_STATELESS 是无效的，将导致返回 NULL。

链接创建的副作用是通过将消费者设备及其所有依赖项移动到这些列表的末尾（只对在调用此函数时尚未注册的设备不发生这种情况）来重新排序 dpm\_list 和 devices\_kset 列表。

在调用此函数时，要求已注册供应商设备，如果未注册，将返回 NULL。消费者设备不需要注册。

### void device\_link\_del(struct device\_link \* link)

删除两个设备之间的链接。

**参数**

struct device\_link \* link

要删除的设备链接。

**说明**

调用者必须确保此函数与运行时 PM 正确同步。如果链接添加多次，则需要删除多次。对于热插拔设备需要特别注意它们的链接会在去除时清除，此时不能再调用 device\_link\_del()。

# 基于消息的设备

## 融合消息设备

### u8 mpt\_register(MPT\_CALLBACK cbfunc, MPT\_DRIVER\_CLASS dclass, char \*func\_name)

注册特定于协议的主回调处理程序。

**参数**

MPT\_CALLBACK cbfunc

回调函数指针

MPT\_DRIVER\_CLASS dclass

协议驱动程序的类别（MPT\_DRIVER\_CLASS enum值）

char \*func\_name

调用函数的名称

**说明**

协议特定的驱动程序（SCSI 主机，LAN，SCSI 目标）调用此例程以注册其答复回调例程。每个协议特定的驱动程序必须在使用任何 IOC 资源之前这样做，例如获取请求帧。

**注释**

SCSI 协议驱动程序当前调用此例程三次，为“普通”SCSI IO、MptScsiTaskMgmt 请求和扫描/DV 请求注册单独的回调。

如果成功，返回为在范围（和 S.O.D. 顺序）内的 u8 值“句柄” {N,…,7,6,5,…,1}。调用者应将返回为 MPT\_MAX\_PROTOCOL\_DRIVERS（包括零）视为错误。

### void mpt\_deregister(u8 cb\_idx)

取消注册协议驱动程序资源。

**参数**

u8 cb\_idx

以前注册的回调句柄

每个协议特定的驱动程序在卸载其模块时都应调用此例程。

### int mpt\_event\_register(u8 cb\_idx, MPT\_EVHANDLER ev\_cbfunc)

注册特定于协议的事件回调处理程序。

**参数**

u8 cb\_idx

先前已注册的（通过 mpt\_register 注册）回调句柄

MPT\_EVHANDLER ev\_cbfunc

回调函数

**说明**

如果/当一个或多个协议特定的驱动程序选择被通知 MPT 事件时，可以调用此例程。

成功返回 0。

### void mpt\_event\_deregister(u8 cb\_idx)

取消注册协议特定的事件回调处理程序。

**参数**

u8 cb\_idx

以前注册的回调句柄

**说明**

每个协议特定的驱动程序在它不（或不能再）处理事件时或在其模块被卸载时都应调用此例程。

### int mpt\_reset\_register(u8 cb\_idx, MPT\_RESETHANDLER reset\_func)

注册协议特定的 IOC 重置处理程序

**参数**

u8 cb\_idx

之前通过 mpt\_register 注册的回调函数句柄

MPT\_RESETHANDLER reset\_func

重置函数

**说明**

如果协议特定的驱动选择在 IOC 重置时获得通知，则可以由一个或多个协议特定的驱动程序调用此例程。

成功返回 0。

### void mpt\_reset\_deregister (u8 cb\_idx)

取消注册协议特定的 IOC 重置处理程序。

**参数**

u8 cb\_idx

之前注册的回调句柄

**说明**

每个协议特定的驱动程序在不处理 IOC 重置处理或卸载其模块时应调用此例程。

### int mpt\_device\_driver\_register（struct mpt\_pci\_driver \*dd\_cbfunc，u8 cb\_idx）

注册设备驱动程序挂钩

**参数**

struct mpt\_pci\_driver \* dd\_cbfunc

驱动程序回调结构

u8 cb\_idx

MPT 协议驱动程序索引

### void mpt\_device\_driver\_deregister（u8 cb\_idx）

取消注册设备驱动程序挂钩

**参数**

u8 cb\_idx

MPT 协议驱动程序索引

### MPT\_FRAME\_HDR \*mpt\_get\_msg\_frame（u8 cb\_idx，MPT\_ADAPTER \* ioc）

从池中获取 MPT 请求帧

**参数**

u8 cb\_idx

已注册的 MPT 协议驱动的句柄

MPT\_ADAPTER \* ioc

指向 MPT 适配器结构的指针

**说明**

从每个 MPT 适配器分配的池中获取 MPT 请求帧（1024 个）。

如果没有可用或 IOC 未激活，则返回指向 MPT 请求帧的指针为空。

### void mpt\_put\_msg\_frame（u8 cb\_idx，MPT\_ADAPTER \* ioc，MPT\_FRAME\_HDR \* mf）

将协议特定的 MPT 请求帧发送到 IOC。

**参数**

u8 cb\_idx

已注册的 MPT 协议驱动的句柄

MPT\_ADAPTER \* ioc

指向 MPT 适配器结构的指针

MPT\_FRAME\_HDR \* mf

指向 MPT 请求帧的指针

**说明**

此例程将 MPT 请求帧发布到特定 MPT 适配器的请求发布 FIFO 中。

### void mpt\_put\_msg\_frame\_hi\_pri（u8 cb\_idx，MPT\_ADAPTER \* ioc，MPT\_FRAME\_HDR \* mf）

发送高优先级的协议特定的 MPT 请求帧

**参数**

u8 cb\_idx

已注册的 MPT 协议驱动的句柄

MPT\_ADAPTER \* ioc

指向 MPT 适配器结构的指针

MPT\_FRAME\_HDR \* mf

指向 MPT 请求帧的指针

**说明**

使用高优先级请求队列将协议特定的 MPT 请求帧发送到 IOC。

此例程将 MPT 请求帧发布到特定 MPT 适配器的请求发布 FIFO 中。

### void mpt\_free\_msg\_frame（MPT\_ADAPTER \* ioc，MPT\_FRAME\_HDR \* mf）

将 MPT 请求帧放回 FreeQ。

**参数**

MPT\_ADAPTER \* ioc

指向 MPT 适配器结构的指针

MPT\_FRAME\_HDR \* mf

指向 MPT 请求帧的指针

**说明**

此例程将 MPT 请求帧放回 MPT 适配器的 FreeQ。

### int mpt\_send\_handshake\_request（u8 cb\_idx，MPT\_ADAPTER \* ioc，int reqBytes，u32 \* req，int sleepFlag）

通过 doorbell 握手方法发送 MPT 请求。

**参数**

u8 cb\_idx

已注册的 MPT 协议驱动的句柄

MPT\_ADAPTER \* ioc

指向 MPT 适配器结构的指针

int reqBytes

请求的大小（字节为单位）

u32 \* req

指向 MPT 请求帧的指针

int sleepFlag

如果 CAN\_SLEEP 则使用 schedule，否则使用 udelay。

**说明**

此例程仅用于发送 MptScsiTaskMgmt 请求，因为它们必须通过 doorbell 握手发送。

**注意**

呼叫者有责任对执行大小超过 1 字节的请求中的字段进行字节交换。

成功返回 0，失败返回非零。

### int mpt\_verify\_adapter（int iocid，MPT\_ADAPTER \*\* iocpp）

给定 IOC 标识符，将指针设置为其适配器结构。

**参数**

int iocid

IOC 唯一标识符（整数）

MPT\_ADAPTER \*\* iocpp

指向 IOC 适配器的指针

**说明**

给定唯一的 IOC 标识符，设置指向相关 MPT 适配器结构的指针。

如果找到 iocid，则返回 iocid 并设置 iocpp。如果未找到 iocid，则返回 -1。

### int mpt\_attach（struct pci\_dev \* pdev，const struct pci\_device\_id \* id）

安装 PCI 智能 MPT 适配器。

**参数**

struct pci\_dev \* pdev

pci\_dev 结构的指针

const struct pci\_device\_id \* id

PCI 设备 ID 信息

**说明**

此例程执行将 MPT 适配器的 IOC 带入 OPERATIONAL 状态所需的所有步骤。这包括注册内存区域、注册中断并分配请求和回复内存池。

此例程还预获取光纤通道 MPT 适配器的 LAN MAC 地址。

成功返回 0，失败返回非零。

TODO: 添加对轮询控制器的支持

### void mpt\_detach（struct pci\_dev \* pdev）

移除 PCI 智能 MPT 适配器。

**参数**

struct pci\_dev \* pdev

pci\_dev 结构的指针

### int mpt\_suspend（struct pci\_dev \* pdev，pm\_message\_t state）

Fusion MPT 基本驱动程序挂起例程。

**参数**

struct pci\_dev \* pdev

pci\_dev 结构的指针

pm\_message\_t state

要进入的新状态

### int mpt\_resume（struct pci\_dev \* pdev）

Fusion MPT 基础驱动程序恢复例程。

**参数**

struct pci\_dev \* pdev

pci\_dev 结构的指针

### u32 mpt\_GetIocState（MPT\_ADAPTER \* ioc，int cooked）

获取 MPT 适配器的当前状态。

**参数**

MPT\_ADAPTER \* ioc

指向 MPT\_ADAPTER 结构的指针

int cooked

请求原始或熟悉的 IOC 状态

**说明**

如果 cooked==0，则返回所有 IOC Doorbell 寄存器位，否则仅返回 MPI\_IOC\_STATE\_MASK 中的 Doorbell 位。

### int mpt\_alloc\_fw\_memory（MPT\_ADAPTER \* ioc，int size）

分配固件内存

**参数**

MPT\_ADAPTER \* ioc

指向 MPT\_ADAPTER 结构的指针

int size

总固件字节数

**说明**

如果已经分配了内存，则返回相同（缓存的）值。

如果成功则返回0，否则返回非零值

### void mpt\_free\_fw\_memory（MPT\_ADAPTER \* ioc）

释放固件内存

**参数**

MPT\_ADAPTER \* ioc

指向MPT\_ADAPTER结构的指针

**说明**

如果alt\_img为NULL，则从ioc结构中删除。 否则，在相同格式中删除一个辅助映像。

### int mptbase\_sas\_persist\_operation（MPT\_ADAPTER \* ioc，u8 persist\_opcode）

对SAS持久表执行操作

**参数**

MPT\_ADAPTER \* ioc

指向MPT\_ADAPTER结构的指针

u8 persist\_opcode

见下文

**说明**

MPI\_SAS\_OP\_CLEAR\_NOT\_PRESENT - 释放所有持久的 TargetID 映射

当前不存在的设备。

MPI\_SAS\_OP\_CLEAR\_ALL\_PERSISTENT - 清除所有持久 TargetID 映射

**注意**

不要在中断时间使用此函数。

成功返回0，非零错误返回

### int mpt\_raid\_phys\_disk\_pg0（MPT\_ADAPTER \* ioc，u8 phys\_disk\_num，RaidPhysDiskPage0\_t \* phys\_disk）

返回物理磁盘页零

**参数**

MPT\_ADAPTER \* ioc

指向适配器结构的指针

u8 phys\_disk\_num

由ioc生成的IO单元唯一物理磁盘编号

RaidPhysDiskPage0\_t \* phys\_disk

请求的有效负载数据返回

**返回**

成功返回0

-如果读取配置页标题失败或数据指针不为NULL，则返回-EFAULT；

-如果pci\_alloc失败，则返回-ENOMEM。

### int mpt\_raid\_phys\_disk\_get\_num\_paths（MPT\_ADAPTER \* ioc，u8 phys\_disk\_num）

返回与此物理编号关联的路径数

**参数**

MPT\_ADAPTER \* ioc

指向适配器结构的指针

u8 phys\_disk\_num

由ioc生成的IO单元唯一物理磁盘编号

**返回**

返回路径数

### int mpt\_raid\_phys\_disk\_pg1（MPT\_ADAPTER \* ioc，u8 phys\_disk\_num，RaidPhysDiskPage1\_t \* phys\_disk）

返回物理磁盘页1

**参数**

MPT\_ADAPTER \* ioc

指向适配器结构的指针

u8 phys\_disk\_num

由ioc生成的IO单元唯一物理磁盘编号

RaidPhysDiskPage1\_t \* phys\_disk

请求的有效负载数据返回

**返回**

成功返回0

-如果读取配置页标题失败或数据指针不为NULL，则返回-EFAULT；

-如果pci\_alloc失败，则返回-ENOMEM。

### int mpt\_findImVolumes（MPT\_ADAPTER \* ioc）

识别隐藏磁盘和RAID卷的ID

**参数**

MPT\_ADAPTER \* ioc

指向适配器结构的指针

**返回**

成功返回0

-如果读取配置页标题失败或数据指针不为NULL，则返回-EFAULT；

-如果pci\_alloc失败，则返回-ENOMEM。

### int mpt\_config（MPT\_ADAPTER \* ioc，CONFIGPARMS \* pCfg）

通用功能以发出配置消息

**参数**

MPT\_ADAPTER \* ioc

指向适配器结构的指针

CONFIGPARMS \* pCfg

指向配置结构的指针。 结构包含操作，页面地址，方向，物理地址和指向配置页标题的指针，页面头已更新。

**说明**

成功返回0

-如果当前没有可用的消息帧，则返回-EAGAIN；

-如果未成功回复或未回复（超时） -EFAULT。

### void mpt\_print\_ioc\_summary（MPT\_ADAPTER \* ioc，char \* buffer，int \* size，int len，int showlan）

将IOC的ASCII摘要写入缓冲区。

**参数**

MPT\_ADAPTER \* ioc

指向MPT\_ADAPTER结构的指针

char \* buffer

指向应将IOC摘要信息写入的缓冲区的指针

int \* size

指向我们写入的字节数（由此例程设置）的指针

int len

在缓冲区中开始写入的偏移量

int showlan

是否显示LAN？

**说明**

此例程将ASCII文本（可读的英语）写入缓冲区，该文本表示IOC信息的摘要。

### int mpt\_set\_taskmgmt\_in\_progress\_flag（MPT\_ADAPTER \* ioc）

设置与任务管理相关的标志

**参数**

MPT\_ADAPTER \* ioc

指向MPT\_ADAPTER结构的指针

**说明**

如果成功，则返回0；如果失败，则返回-1。

如果返回-1，则无法设置标志

### void mpt\_clear\_taskmgmt\_in\_progress\_flag（MPT\_ADAPTER \* ioc）

清除与任务管理相关的标志

**参数**

MPT\_ADAPTER \* ioc

指向MPT\_ADAPTER结构的指针

### void mpt\_halt\_firmware（MPT\_ADAPTER \* ioc）

如果固件正在运行，则停止固件并使内核处于恐慌状态

**参数**

MPT\_ADAPTER \* ioc

指向MPT\_ADAPTER结构的指针

### int mpt\_Soft\_Hard\_ResetHandler（MPT\_ADAPTER \* ioc，int sleepFlag）

尝试更便宜的重置

**参数**

MPT\_ADAPTER \* ioc

指向MPT\_ADAPTER结构的指针

int sleepFlag

指示必须调用sleep或schedule。

**说明**

如果它失败了，请尝试softreset，仅在失败时才进行昂贵的HardReset。

### int mpt\_HardResetHandler（MPT\_ADAPTER \* ioc，int sleepFlag）

常规重置处理程序

**参数**

MPT\_ADAPTER \* ioc

指向MPT\_ADAPTER结构的指针

int sleepFlag

指示必须调用sleep或schedule。

**说明**

基于输入参数值发出SCSI任务管理调用。 如果TaskMgmt失败，则返回相关的SCSI请求。

备注\_HardResetHandler可以从中断线程（定时器）或非中断线程调用。 在前者中，不能调用调度程序（）。

**注意**

返回-1是致命错误的情况，因为它意味着发生FW重新加载/初始化失败。

成功返回0

### const char \* mptscsih\_info（struct Scsi\_Host \* SChost）

返回有关MPT适配器的信息

**参数**

struct Scsi\_Host \* SChost

指向Scsi\_Host结构的指针

（linux scsi\_host\_template.info routine）

返回写入信息的缓冲区的指针。

### int mptscsih\_qcmd（struct scsi\_cmnd \* SCpnt）

主要的Fusion MPT SCSI发起者IO启动例程。

**参数**

struct scsi\_cmnd \* SCpnt

指向scsi\_cmnd结构的指针

**说明**

（linux scsi\_host\_template.queuecommand routine）这是主要的SCSI IO启动例程。从linux scsi\_cmnd请求创建MPI SCSIIORequest并将其发送到IOC。

返回0。（rtn值被Linux SCSI中层舍弃）

### int mptscsih\_IssueTaskMgmt（MPT\_SCSI\_HOST \* hd，u8 type，u8 channel，u8 id，u64 lun，int ctx2abort，ulong timeout）

通用发送任务管理函数。

**参数**

MPT\_SCSI\_HOST \* hd

指向MPT\_SCSI\_HOST结构的指针

u8 type

任务管理类型

u8 channel

任务管理通道号

u8 id

重置的逻辑目标ID（如果适用）

u64 lun

重置的逻辑单元（如果适用）

int ctx2abort

要中止任务的上下文（如果适用）

ulong timeout

任务管理控制的超时时间

**说明**

备注\_HardResetHandler可以从中断线程（定时器）或非中断线程调用。 在前者中，不得调用schedule（）。

并非所有字段对所有任务类型都有意义。

返回0表示成功，或FAILED。

### int mptscsih\_abort（struct scsi\_cmnd \* SCpnt）

中止linux scsi\_cmnd例程，new\_eh变体

**参数**

struct scsi\_cmnd \* SCpnt

指向要中止的scsi\_cmnd结构的指针IO

（linux scsi\_host\_template.eh\_abort\_handler例程）

返回成功或失败。

### int mptscsih\_dev\_reset（struct scsi\_cmnd \* SCpnt）

执行SCSI TARGET\_RESET！ new\_eh变体

**参数**

struct scsi\_cmnd \* SCpnt

指向IO的scsi\_cmnd结构的指针，该结构需要进行重置

**说明**

（linux scsi\_host\_template.eh\_dev\_reset\_handler例程）

返回成功或失败。

### int mptscsih\_bus\_reset（struct scsi\_cmnd \* SCpnt）

执行SCSI BUS\_RESET！ new\_eh变体

**参数**

struct scsi\_cmnd \* SCpnt

指向IO的scsi\_cmnd结构的指针，该结构需要进行重置

**说明**

（linux scsi\_host\_template.eh\_bus\_reset\_handler例程）

返回成功或失败。

### int mptscsih\_host\_reset（struct scsi\_cmnd \* SCpnt）

执行SCSI主机适配器重置（new\_eh变体）

**参数**

struct scsi\_cmnd \* SCpnt

指向IO的scsi\_cmnd结构的指针，该结构需要进行重置

**说明**

（linux scsi\_host\_template.eh\_host\_reset\_handler例程）

返回成功或失败。

### int mptscsih\_taskmgmt\_complete（MPT\_ADAPTER \* ioc，MPT\_FRAME\_HDR \* mf，MPT\_FRAME\_HDR \* mr）

注册到Fusion MPT基本驱动程序

**参数**

MPT\_ADAPTER \* ioc

指向MPT\_ADAPTER结构的指针

MPT\_FRAME\_HDR \* mf

指向SCSI任务管理请求帧的指针

MPT\_FRAME\_HDR \* mr

指向SCSI任务管理回复帧的指针

**说明**

在任何SCSI任务管理请求完成时，从mptbase.c :: mpt\_interrupt（）调用此例程。 该例程通过mpt\_register（）API调用在驱动程序加载/初始化时向MPT（base）驱动程序注册。

返回1，表示分配的请求帧指针应被释放。

### struct scsi\_cmnd \* mptscsih\_get\_scsi\_lookup（MPT\_ADAPTER \* ioc，int i）

检索scmd条目

**参数**

MPT\_ADAPTER \* ioc

指向MPT\_ADAPTER结构的指针

int i

数组中的索引

**说明**

返回scsi\_cmd指针see below

# 声音设备

## snd\_printk(fmt, ...)

打印包装器

**参数**

fmt

格式字符串

...

变量参数

**说明**

与printk（）类似，但在配置为CONFIG\_SND\_VERBOSE\_PRINTK时打印调用者的文件和行。

## snd\_printd（fmt，...）

调试printk

**参数**

fmt

格式字符串

...

变量参数

**说明**

与snd\_printk（）类似，用于调试目的。 在未设置CONFIG\_SND\_DEBUG时被忽略。

## snd\_BUG（）

给出一个BUG警告消息和堆栈跟踪

**参数**

**说明**

如果设置了CONFIG\_SND\_DEBUG，则调用WARN（）。 在未设置CONFIG\_SND\_DEBUG时被忽略。

## snd\_printd\_ratelimit（）

**参数**

snd\_BUG\_ON（cond）

调试检查宏

**参数**

cond

要评估的条件

**说明**

当设置CONFIG\_SND\_DEBUG时具有与WARN\_ON相同的行为，否则仅评估条件并返回。

## snd\_printdd（format，...）

调试printk

**参数**

format

格式字符串

...

变量参数

**说明**

与snd\_printk（）类似，用于调试目的。 在未设置CONFIG\_SND\_DEBUG\_VERBOSE时被忽略。

## int register\_sound\_special\_device（const struct file\_operations \* fops，int unit，struct device \* dev）

注册特殊声音节点

**参数**

const struct file\_operations \* fops

驱动程序的文件操作

int unit

要分配的单元号

struct device \* dev

设备指针

**说明**

从声音子系统按较小号分配特殊声音设备。

**返回**

成功时返回分配的数字。 失败时返回负错误代码。

## int register\_sound\_mixer（const struct file\_operations \* fops，int dev）

注册混合器设备

**参数**

const struct file\_operations \* fops

驱动程序的文件操作

int dev

要分配的单元号

**说明**

分配混合器设备。 单元是请求的混合器数量。 传递-1以请求下一个空闲混合器单元。

**返回**

成功时返回分配的数字。 失败时返回负错误代码。

## int register\_sound\_dsp（const struct file\_operations \* fops，int dev）

注册DSP设备

**参数**

const struct file\_operations \* fops

驱动程序的文件操作

int dev

要分配的单元号

**说明**

分配DSP设备。 单元是请求的DSP数量。 传递-1以请求下一个空闲DSP单元。

此函数一起分配音频和dsp设备条目，并且将始终将它们分配为匹配对 - 例如dsp3 / audio3

**返回**

成功时返回分配的数字。 失败时返回负错误代码。

## void unregister\_sound\_special（int unit）

注销特殊声音设备

**参数**

int unit

要分配的单元号

**说明**

释放使用register\_sound\_special（）分配的声音设备。 传递的单元是注册函数的返回。

## void unregister\_sound\_mixer（int unit）

注销混合器

**参数**

int unit

要分配的单元号

**说明**

释放使用register\_sound\_mixer（）分配的声音设备。 传递的单元是注册函数的返回。

## void unregister\_sound\_dsp（int unit）

注销DSP设备

**参数**

int unit

要分配的单元号

**说明**

释放使用register\_sound\_dsp（）分配的声音设备。 传递的单元是注册函数的返回。

两个分配的单元一起自动释放。

## int snd\_pcm\_stream\_linked（struct snd\_pcm\_substream \* substream）

检查substream是否链接到其他流

**参数**

snd\_pcm\_substream \* substream

要检查的流

**说明**

如果给定的子流正在与其他流链接，则返回true。

## snd\_pcm\_stream\_lock\_irqsave（substream，flags）

锁定PCM流程

**参数**

substream

PCM子流

flags

irq标志

**说明**

这锁定PCM流程就像snd\_pcm\_stream\_lock（）一样，但是在本地IRQ（仅在非原子操作时）中。在非原子情况下，这与snd\_pcm\_stream\_lock（）相同。

## snd\_pcm\_group\_for\_each\_entry（s，substream）

迭代链接的子流

**参数**

s

迭代器

substream

子流

**说明**

迭代到链接到给定子流的所有链接子流。如果子流没有与其他人链接，则将返回子流本身一次。

## int snd\_pcm\_running（struct snd\_pcm\_substream \* substream）

检查子流是否处于运行状态

**参数**

struct snd\_pcm\_substream \* substream

要检查的流

**说明**

如果给定的子流处于状态RUNNING或状态DRAINING（用于播放）中，则返回true。

## ssize\_t bytes\_to\_samples（struct snd\_pcm\_runtime \* runtime，ssize\_t size）

从字节转换为样本的单位转换

**参数**

struct snd\_pcm\_runtime \* runtime

PCM运行时实例

ssize\_t size

大小（以字节为单位）

## snd\_pcm\_sframes\_t bytes\_to\_frames（struct snd\_pcm\_runtime \* runtime，ssize\_t size）

从字节转换为帧的单位转换

**参数**

struct snd\_pcm\_runtime \* runtime

PCM运行时实例

ssize\_t size

大小（以字节为单位）

## ssize\_t samples\_to\_bytes（struct snd\_pcm\_runtime \* runtime，ssize\_t size）

从样本转换为字节的单位转换

**参数**

struct snd\_pcm\_runtime \* runtime

PCM运行时实例

ssize\_t size

大小（以样本为单位）

## ssize\_t frames\_to\_bytes（struct snd\_pcm\_runtime \* runtime，snd\_pcm\_sframes\_t size）

从帧转换为字节的单位转换

**参数**

struct snd\_pcm\_runtime \* runtime

PCM运行时实例

snd\_pcm\_sframes\_t size

大小（以帧为单位）

## int frame\_aligned（struct snd\_pcm\_runtime \* runtime，ssize\_t bytes）

检查字节大小是否对齐到帧

**参数**

struct snd\_pcm\_runtime \* runtime

PCM运行时实例

ssize\_t bytes

大小（以字节为单位）

## size\_t snd\_pcm\_lib\_buffer\_bytes（struct snd\_pcm\_substream \* substream）

获取当前PCM的缓冲区大小（以字节为单位）

**参数**

struct snd\_pcm\_substream \* substream

PCM子流

## size\_t snd\_pcm\_lib\_period\_bytes（struct snd\_pcm\_substream \* substream）

获取当前PCM的周期大小（以字节为单位）

**参数**

struct snd\_pcm\_substream \* substream

PCM子流

## snd\_pcm\_uframes\_t snd\_pcm\_playback\_avail（struct snd\_pcm\_runtime \* runtime）

获取播放可用（可写）空间

**参数**

struct snd\_pcm\_runtime \* runtime

PCM运行时实例

**说明**

结果在0 ...（边界-1）之间

## snd\_pcm\_uframes\_t snd\_pcm\_capture\_avail（struct snd\_pcm\_runtime \* runtime）

获取可捕获的可用（可读）空间

**参数**

struct snd\_pcm\_runtime \* runtime

PCM运行时实例

**说明**

结果在0 ...（边界-1）之间

## snd\_pcm\_sframes\_t snd\_pcm\_playback\_hw\_avail（struct snd\_pcm\_runtime \* runtime）

获取播放队列空间

**参数**

struct snd\_pcm\_runtime \* runtime

PCM运行时实例

## snd\_pcm\_sframes\_t snd\_pcm\_capture\_hw\_avail（struct snd\_pcm\_runtime \* runtime）

获取捕获的空闲空间

**参数**

struct nd\_pcm\_runtime \* runtime

PCM运行时实例

## int snd\_pcm\_playback\_ready（struct snd\_pcm\_substream \* substream）

检查播放缓冲区是否可用

**参数**

struct snd\_pcm\_substream \* substream

pcm子流实例

**说明**

检查播放缓冲区上是否有足够的空闲空间。

**返回**

如果可用，则为非零；如果不是，则为零。

## int snd\_pcm\_capture\_ready（struct snd\_pcm\_substream \* substream）

检查捕获缓冲区是否可用

**参数**

struct snd\_pcm\_substream \* substream

pcm子流实例

**说明**

检查捕获缓冲区上是否有足够的捕获数据。

**返回**

如果可用，则为非零；如果不是，则为零。

## int snd\_pcm\_playback\_data（struct snd\_pcm\_substream \* substream）

检查播放缓冲区上是否存在任何数据

**参数**

struct snd\_pcm\_substream \* substream

pcm子流实例

**说明**

检查播放缓冲区上是否存在任何数据。

**返回**

如果存在任何数据，则为非零；如果不存在，则为零。如果stop\_threshold大于或等于边界，则该函数始终返回非零。

## int snd\_pcm\_playback\_empty（struct snd\_pcm\_substream \* substream）

检查播放缓冲区是否为空

**参数**

struct snd\_pcm\_substream \* substream

pcm子流实例

**说明**

检查播放缓冲区是否为空。

**返回**

如果为空，则为非零；如果不是，则为零。

## int snd\_pcm\_capture\_empty（struct snd\_pcm\_substream \* substream）

检查捕获缓冲区是否为空

**参数**

struct snd\_pcm\_substream \* substream

pcm子流实例

**说明**

检查捕获缓冲区是否为空。

**返回**

如果为空，则为非零；如果不是，则为零。

## void snd\_pcm\_trigger\_done（struct snd\_pcm\_substream \* substream，struct snd\_pcm\_substream \* master）

标记主子流

**参数**

struct snd\_pcm\_substream \* substream

pcm子流实例

snd\_pcm\_substream \* master

链接的主子流

**说明**

当同一卡的多个子流链接在一起并且硬件支持单次操作时，驱动程序在snd\_pcm\_group\_for\_each\_entry（）循环中调用此函数以标记子流为“done”。然后，大多数触发操作仅对给定的主子流执行。

在触发操作结束时，时间戳更新时清除trigger\_master标记。

## unsigned int params\_channels（const struct snd\_pcm\_hw\_params \* p）

从硬件参数中获取通道数

**参数**

struct const struct snd\_pcm\_hw\_params \* p

硬件参数

## unsigned int params\_rate（const struct snd\_pcm\_hw\_params \* p）

从硬件参数获取采样率

**参数**

const struct snd\_pcm\_hw\_params \*p

硬件参数

## unsigned int params\_period\_size(const struct snd\_pcm\_hw\_params \*p)

从硬件参数获取周期大小（以帧为单位）

**参数**

const struct snd\_pcm\_hw\_params \*p

硬件参数

## unsigned int params\_periods(const struct snd\_pcm\_hw\_params \*p)

从硬件参数取周期数

**参数**

const struct snd\_pcm\_hw\_params \*p

硬件参数

## unsigned int params\_buffer\_size(const struct snd\_pcm\_hw\_params \*p)

从硬件参数获取缓冲区大小（以帧为单位）

**参数**

const struct snd\_pcm\_hw\_params \*p

硬件参数

## unsigned int params\_buffer\_bytes(const struct snd\_pcm\_hw\_params \*p)

从硬件参数获取缓冲区大小（以字节为单位）

**参数**

const struct snd\_pcm\_hw\_params \*p

硬件参数

## int snd\_pcm\_hw\_constraint\_single(struct snd\_pcm\_runtime \*runtime，snd\_pcm\_hw\_param\_t var，unsigned int val)

将参数限制为单个值

**参数**

struct snd\_pcm\_runtime \*runtime

PCM运行时实例

snd\_pcm\_hw\_param\_t var

要约束的hw\_params变量

unsigned int val

要限制的值

**返回**

如果值被更改，则为正数；如果未更改，则为零，如果是负错误代码。

## int snd\_pcm\_format\_cpu\_endian(snd\_pcm\_format\_t format)

检查PCM格式是否为CPU字节序

**参数**

snd\_pcm\_format\_t format

要检查的格式

**返回**

如果给定的PCM格式是CPU字节序，则返回1，否则返回0，如果未指定字节序，则返回负错误代码。

## void snd\_pcm\_set\_runtime\_buffer(struct snd\_pcm\_substream \*substream，struct snd\_dma\_buffer \*bufp)

设置PCM运行时缓冲区

**参数**

struct snd\_pcm\_substream \*substream

要设置的PCM子流

struct snd\_dma\_buffer \*bufp

缓冲区信息，为NULL则清除

**说明**

当bufp非NULL时，将缓冲区信息复制到runtime->dma\_buffer。否则清除当前的缓冲区信息。

## void snd\_pcm\_gettime(struct snd\_pcm\_runtime \*runtime，struct timespec \*tv)

根据时间戳模式填写timespec

**参数**

struct snd\_pcm\_runtime \*runtime

PCM运行时实例

struct timespec \*tv

要填充的timespec

## int snd\_pcm\_lib\_alloc\_vmalloc\_buffer(struct snd\_pcm\_substream \*substream，size\_t size)

分配虚拟DMA缓冲区

**参数**

struct snd\_pcm\_substream \*substream

要将缓冲区分配给的子流

size\_t size

请求的缓冲区大小，以字节为单位

**说明**

使用vmalloc()分配PCM子流缓冲区，即内存在内核虚拟空间中具有连续性，但在物理内存中没有。如果缓冲区由内核代码访问但不由设备DMA访问，则使用此选项。

**返回**

如果缓冲区已更改，则返回1；如果未更改，则返回0，否则返回负错误代码。

## int snd\_pcm\_lib\_alloc\_vmalloc\_32\_buffer(struct snd\_pcm\_substream \*substream，size\_t size)

分配32位可寻址缓冲区

**参数**

struct snd\_pcm\_substream \*substream

要将缓冲区分配给的子流

size\_t size

请求的缓冲区大小，以字节为单位

**说明**

此函数的工作方式类似于snd\_pcm\_lib\_alloc\_vmalloc\_buffer()，但使用vmalloc\_32()，即从32位可寻址内存中分配页面。

**返回**

如果缓冲区已更改，则返回1；如果未更改，则返回0，否则返回负错误代码。

## dma\_addr\_t snd\_pcm\_sgbuf\_get\_addr(struct snd\_pcm\_substream \*substream，unsigned int ofs)

获取相应偏移量处的DMA地址

**参数**

struct snd\_pcm\_substream \*substream

PCM子流

unsigned int ofs

字节偏移量

## void \*snd\_pcm\_sgbuf\_get\_ptr(struct snd\_pcm\_substream \*substream，unsigned int ofs)

获取相应偏移量处的虚拟地址

**参数**

struct snd\_pcm\_substream \*substream

PCM子流

unsigned int ofs

字节偏移量

## unsigned int snd\_pcm\_sgbuf\_get\_chunk\_size(struct snd\_pcm\_substream \*substream，unsigned int ofs，unsigned int size)

计算适合连续页面中的最大大小

**参数**

struct snd\_pcm\_substream \*substream

PCM子流

unsigned int ofs

字节偏移量

unsigned int size

要检查的字节大小

## void snd\_pcm\_mmap\_data\_open(struct vm\_area\_struct \*area)

增加mmap计数器

**参数**

struct vm\_area\_struct \*area

VMA

**说明**

PCM mmap回调应适当处理此计数器

## void snd\_pcm\_mmap\_data\_close(struct vm\_area\_struct \*area)

减少mmap计数器

**参数**

struct vm\_area\_struct \*area

VMA

**说明**

PCM mmap回调应适当处理此计数器

## void snd\_pcm\_limit\_isa\_dma\_size(int dma，size\_t \*max)

获取适合ISA DMA传输的最大大小

**参数**

int dma

DMA编号

size\_t \*max

指向存储最大大小的指针

## const char \*snd\_pcm\_stream\_str(struct snd\_pcm\_substream \*substream)

获取命名流的字符串方向

**参数**

struct snd\_pcm\_substream \*substream

pcm子流实例

**返回**

命名流方向的字符串。

## struct snd\_pcm\_substream \*snd\_pcm\_chmap\_substream(struct snd\_pcm\_chmap \*info，unsigned int idx)

获取分配给给定chmap信息的PCM子流

**参数**

struct snd\_pcm\_chmap \*info

chmap信息

unsigned int idx

子流编号索引

## u64 pcm\_format\_to\_bits(snd\_pcm\_format\_t pcm\_format)

pcm\_format到二进制的强类型转换

**参数**

snd\_pcm\_format\_t pcm\_format

PCM格式

## const char \*snd\_pcm\_format\_name(snd\_pcm\_format\_t format)

为给定的PCM格式返回一个名称字符串

**参数**

snd\_pcm\_format\_t format

PCM格式

## int snd\_pcm\_new\_stream(struct snd\_pcm \*pcm，int stream，int substream\_count)

创建新的PCM流

**参数**

struct snd\_pcm \*pcm

pcm实例

int stream

流方向，SNDRV\_PCM\_STREAM\_XXX

int substream\_count

子流的数量

**说明**

为PCM创建新流。 在调用此函数之前，pcm上相应的流必须为空，即必须为snd\_pcm\_new()的参数传递零。

**返回**

成功则为零，失败则为负错误代码。

## int snd\_pcm\_new(struct snd\_card \* card, const char \* id, int device, int playback\_count, int capture\_count, struct snd\_pcm \*\* rpcm)

创建一个新的PCM实例

**参数**

struct snd\_card \* card

卡的实例

const char \* id

ID字符串

int device

设备索引（从零开始）

int playback\_count

播放的子流数

int capture\_count

捕获的子流数

struct snd\_pcm \*\* rpcm

用于存储新PCM实例的指针

**说明**

创建一个新的PCM实例。

后续需要通过snd\_pcm\_set\_ops()为新实例设置PCM操作符。

**返回**

成功返回零，失败返回负错误代码。

## int snd\_pcm\_new\_internal(struct snd\_card \* card，const char \* id, int device, int playback\_count, int capture\_count, struct snd\_pcm \*\* rpcm)

创建一个新的内部PCM实例

**参数**

struct snd\_card \* card

卡的实例

const char \* id

ID字符串

int device

设备索引（基于共享的正常PCM）

int playback\_count

播放的子流数

int capture\_count

捕获的子流数

struct snd\_pcm \*\* rpcm

用于存储新PCM实例的指针

**说明**

创建一个新的没有用户空间设备或procfs条目的内部PCM实例。这用于 ASoC 后端PCM，以创建仅由内核驱动程序内部使用的PCM。即不能被用户空间打开。它为现有的 ASoC 组件驱动程序提供了一个子流，并提供了访问任何私有数据的方式。

后续需要通过snd\_pcm\_set\_ops()为新实例设置PCM操作符。

**返回**

成功返回零，失败返回负错误代码。

## int snd\_pcm\_notify(struct snd\_pcm\_notify \* notify, int nfree)

添加/删除通知列表

**参数**

struct snd\_pcm\_notify \* notify

PCM通知列表

int nfree

0 = 注册，1 = 反注册

**说明**

将给定的通知添加到全局列表中，以使回调为每个已注册的PCM设备调用。到目前为止，这仅存在于 PCM OSS 模拟中。

## int snd\_device\_new(struct snd\_card \* card, enum snd\_device\_type type, void \* device\_data, struct snd\_device\_ops \* ops)

创建一个 ALSA 设备组件

**参数**

struct snd\_card \* card

卡的实例

enum snd\_device\_type type

设备类型，SNDRV\_DEV\_XXX

void \* device\_data

该设备的数据指针

struct snd\_device\_ops \* ops

操作表

**说明**

为给定的数据指针创建一个新的设备组件。该设备将分配给卡，并由卡一起管理。

数据指针也起到标识符的作用，因此指针地址必须是唯一且不变的。

**返回**

成功返回零，失败返回负错误代码。

## void snd\_device\_disconnect(struct snd\_card \* card, void \* device\_data)

断开该设备

**参数**

struct snd\_card \* card

卡的实例

void \* device\_data

要断开的数据指针

**说明**

将该设备转换为断开状态，如果设备已经注册，则调用dev\_disconnect回调。通常从snd\_card\_disconnect()中调用。

**返回**

成功返回零，失败返回负错误代码或找不到设备。

## void snd\_device\_free(struct snd\_card \* card, void \* device\_data)

释放该设备

**参数**

struct snd\_card \* card

卡的实例

void \* device\_data

要释放的数据指针

**说明**

从卡的列表中删除该设备，并调用与状态相对应的回调dev\_disconnect和dev\_free。然后释放该设备。

## int snd\_device\_register(struct snd\_card \* card, void \* device\_data)

注册该设备

**参数**

struct snd\_card \* card

卡的实例

void \* device\_data

要注册的数据指针

**说明**

已通过snd\_device\_new()创建设备。通常在调用snd\_card\_register()后调用，但如果创建了任何新设备，则稍后可以调用它们。

**返回**

成功返回零，失败返回负错误代码或找不到设备。

## int snd\_info\_get\_line(struct snd\_info\_buffer \* buffer, char \* line, int len)

从procfs缓冲区中读取一行

**参数**

struct snd\_info\_buffer \* buffer

procfs缓冲区

char \* line

要存储的缓冲区

int len

最大缓冲区大小

**说明**

从缓冲区中读取一行并存储字符串。

**返回**

成功返回零，错误或EOF则返回1。

## const char \* snd\_info\_get\_str(char \* dest, const char \* src, int len)

解析一个字符串标记

**参数**

char \* dest

要存储字符串标记的缓冲区

const char \* src

原始字符串

int len

标记的最大长度 - 1

**说明**

解析原始字符串并将标记复制到给定的字符串缓冲区中。

**返回**

更新后的原始字符串指针，以便于下一次调用使用。

## struct snd\_info\_entry \* snd\_info\_create\_module\_entry(struct module \* module, const char \* name, struct snd\_info\_entry \* parent)

为给定的模块创建一个信息条目

**参数**

struct module \* module

模块指针

const char \* name

文件名

struct snd\_info\_entry \* parent

父目录

**说明**

新建一个信息入口并分配到给定的模块。

**返回**

新实例的指针，或失败时为NULL。

## struct snd\_info\_entry \* snd\_info\_create\_card\_entry(struct snd\_card \* card, const char \* name, struct snd\_info\_entry \* parent)

为给定的卡创建信息入口

**参数**

struct snd\_card \* card

卡实例

const char \* name

文件名

struct snd\_info\_entry \* parent

父目录

**说明**

创建一个新的信息入口并将其分配到给定的卡。

**返回**

新实例的指针，或失败时为NULL。

## void snd\_info\_free\_entry(struct snd\_info\_entry \* entry)

释放信息入口

**参数**

struct snd\_info\_entry \* entry

信息入口

**说明**

释放信息入口。

## int snd\_info\_register(struct snd\_info\_entry \* entry)

注册信息入口

**参数**

struct snd\_info\_entry \* entry

信息入口

**说明**

注册处理器信息入口。

**返回**

成功时为零，失败时为负错误代码。

## int snd\_rawmidi\_receive(struct snd\_rawmidi\_substream \* substream, const unsigned char \* buffer, int count)

从设备接收输入数据

**参数**

struct snd\_rawmidi\_substream \* substream

rawmidi子流

const unsigned char \* buffer

缓冲区指针

int count

要读取的数据大小

**说明**

从内部缓冲区读取数据。

**返回**

读取数据的大小，或失败时为负错误代码。

## int snd\_rawmidi\_transmit\_empty(struct snd\_rawmidi\_substream \* substream)

检查输出缓冲区是否为空

**参数**

struct snd\_rawmidi\_substream \* substream

rawmidi子流

**返回**

如果内部输出缓冲区为空则为1，否则为0。

## int \_\_snd\_rawmidi\_transmit\_peek(struct snd\_rawmidi\_substream \* substream, unsigned char \* buffer, int count)

从内部缓冲区复制数据

**参数**

struct snd\_rawmidi\_substream \* substream

rawmidi子流

unsigned char \* buffer

缓冲区指针

int count

要传输的数据大小

**说明**

这是一个没有自旋锁的snd\_rawmidi\_transmit\_peek()变体。

## int snd\_rawmidi\_transmit\_peek(struct snd\_rawmidi\_substream \* substream, unsigned char \* buffer, int count)

从内部缓冲区复制数据

**参数**

struct snd\_rawmidi\_substream \* substream

rawmidi子流

unsigned char \* buffer

缓冲区指针

int count

要传输的数据大小

**说明**

将数据从内部输出缓冲区复制到给定的缓冲区。

当midi输出准备好时，在中断处理程序中调用此函数，并在传输完成后调用snd\_rawmidi\_transmit\_ack()。

**返回**

成功时为复制数据的大小，否则为负错误代码。

## int \_\_snd\_rawmidi\_transmit\_ack(struct snd\_rawmidi\_substream \* substream, int count)

确认传输

**参数**

struct snd\_rawmidi\_substream \* substream

rawmidi子流

int count

传输的计数

**说明**

这是一个没有自旋锁的\_\_snd\_rawmidi\_transmit\_ack()变体。

## int snd\_rawmidi\_transmit\_ack(struct snd\_rawmidi\_substream \* substream, int count)

确认传输

**参数**

struct snd\_rawmidi\_substream \* substream

rawmidi子流

int count

传输的计数

**说明**

将内部输出缓冲区的硬件指针向前移动给定大小并更新状态。传输完成后调用此函数。

**返回**

成功时为先进的大小，否则为负错误代码。

## int snd\_rawmidi\_transmit(struct snd\_rawmidi\_substream \* substream, unsigned char \* buffer, int count)

从缓冲区复制到设备

**参数**

struct snd\_rawmidi\_substream \* substream

rawmidi子流

unsigned char \* buffer

缓冲区指针

int count

要传输的数据大小

**说明**

将数据从缓冲区复制到设备并移动指针。

**返回**

成功时为已复制的大小，否则为负错误代码。

## int snd\_rawmidi\_new(struct snd\_card \* card, char \* id, int device, int output\_count, int input\_count, struct snd\_rawmidi \*\* rrawmidi)

创建一个rawmidi实例

**参数**

struct snd\_card \* card

卡实例

char \* id

id字符串

int device

设备索引

int output\_count

输出流的数量

int input\_count

输入流的数量

struct snd\_rawmidi \*\* rrawmidi

存储新的rawmidi实例的指针

**说明**

创建一个新的rawmidi实例。使用snd\_rawmidi\_set\_ops()设置新实例的操作。

**返回**

成功时为零，否则为负错误代码。

## void snd\_rawmidi\_set\_ops(struct snd\_rawmidi \* rmidi, int stream, const struct snd\_rawmidi\_ops \* ops)

设置rawmidi运算符

**参数**

struct snd\_rawmidi \* rmidi

rawmidi实例

int stream

流方向，SNDRV\_RAWMIDI\_STREAM\_XXX

const struct snd\_rawmidi\_ops \* ops

算子表

**说明**

设置给定流方向的rawmidi运算符。

## void snd\_request\_card(int card)

尝试加载卡模块

**参数**

int card

卡号

**说明**

尝试通过request\_module为给定的卡号加载模块“snd-card-X”。如果已加载，则立即返回。

## void \* snd\_lookup\_minor\_data(unsigned int minor, int type)

获取已注册设备的用户数据

**参数**

unsigned int minor

小号

int type

设备类型（SNDRV\_DEVICE\_TYPE\_XXX）

**说明**

检查是否已注册指定类型的小型设备，并返回其用户数据指针。

如果找到具有给定小号和类型的关联实例，则该函数会增加卡实例的参考计数器。调用者必须适当调用snd\_card\_unref()。

**返回**

如果找到指定设备，则用户数据指针。否则为NULL。

## int snd\_register\_device(int type, struct snd\_card \* card, int dev, const struct file\_operations \* f\_ops, void \* private\_data, struct device \* device)

为卡注册ALSA设备文件

**参数**

int type

设备类型，SNDRV\_DEVICE\_TYPE\_XXX

struct snd\_card \* card

卡实例

int dev

设备索引

const struct file\_operations \* f\_ops

文件操作

void \* private\_data

f\_ops->c:func:open()的用户指针

struct device \* device

注册设备

**说明**

为给定卡注册ALSA设备文件。操作员必须在参数中设置。

**返回**

成功则返回零，否则返回负错误代码。

## int snd\_unregister\_device（struct device \* dev）

注销给定卡上的设备

**参数**

struct device \* dev

设备实例

**说明**

注销已通过snd\_register\_device()注册的设备文件。

**返回**

成功则返回零，否则返回负错误代码。

## int copy\_to\_user\_fromio(void \_\_user \* dst，const volatile void \_\_iomem \* src，size\_t count)

将数据从mmio空间复制到用户空间

**参数**

void \_\_user \* dst

用户空间上的目标指针

const volatile void \_\_iomem \* src

mmio上的源指针

size\_t count

要以字节为单位复制的数据大小

**说明**

将数据从mmio空间复制到用户空间。

**返回**

成功则返回零，否则返回非零值。

## int copy\_from\_user\_toio(volatile void \_\_iomem \* dst，const void \_\_user \* src，size\_t count)

将数据从用户空间复制到mmio空间

**参数**

volatile void \_\_iomem \* dst

mmio空间上的目标指针

const void \_\_user \* src

用户空间上的源指针

size\_t count

要以字节为单位复制的数据大小

**说明**

将数据从用户空间复制到mmio空间。

**返回**

成功则返回零，否则返回非零值。

## int snd\_pcm\_lib\_preallocate\_free\_for\_all(struct snd\_pcm \* pcm)

释放pcm上的所有预分配缓冲区

**参数**

struct snd\_pcm \* pcm

pcm实例

**说明**

释放给定pcm上的所有预分配缓冲区。

**返回**

成功则返回零，否则返回负错误代码。

## int snd\_pcm\_lib\_preallocate\_pages(struct snd\_pcm\_substream \* substream，int type，struct device \* data，size\_t size，size\_t max)

为给定的DMA类型进行预分配

**参数**

struct snd\_pcm\_substream \* substream

pcm子流实例

int type

DMA类型（SNDRV\_DMA\_TYPE\_\*）

struct device \* data

依赖于DMA类型的数据

size\_t size

以字节为单位的请求预分配大小

size\_t max

允许的最大预分配大小

**说明**

为给定的DMA缓冲区类型做预分配。

**返回**

成功则返回零，否则返回负错误代码。

## int snd\_pcm\_lib\_preallocate\_pages\_for\_all(struct snd\_pcm \* pcm，int type，void \* data，size\_t size，size\_t max)

为连续内存类型进行预分配（所有子流）

**参数**

struct snd\_pcm \* pcm

pcm实例

int type

DMA类型（SNDRV\_DMA\_TYPE\_\*）

void \* data

依赖于DMA类型的数据

size\_t size

以字节为单位的请求预分配大小

size\_t max

允许的最大预分配大小

**说明**

为给定pcm的所有子流进行预分配。

**返回**

成功则返回零，否则返回负错误代码。

## struct page \* snd\_pcm\_sgbuf\_ops\_page(struct snd\_pcm\_substream \* substream，unsigned long offset)

获取给定偏移量处的页面结构

**参数**

struct snd\_pcm\_substream \* substream

pcm子流实例

unsigned long offset

缓冲区偏移量

**说明**

作为PCM操作的页面回调使用。

**返回**

在给定缓冲区偏移量处的页面结构。失败则返回NULL。

## int snd\_pcm\_lib\_malloc\_pages(struct snd\_pcm\_substream \* substream，size\_t size)

分配DMA缓冲区

**参数**

struct snd\_pcm\_substream \* substream

要将DMA缓冲区分配给的子流

size\_t size

要请求的缓冲区大小（以字节为单位）

**说明**

在snd\_pcm\_lib\_preallocate\_xxx\_pages()中给出的BUS类型上为DMA缓冲区分配空间。

**返回**

如果缓冲区已更改，则返回1，如果未更改，则返回0，否则返回负代码以表示失败。

## int snd\_pcm\_lib\_free\_pages(struct snd\_pcm\_substream \* substream)

释放已分配的DMA缓冲区。

**参数**

struct snd\_pcm\_substream \* substream

要释放DMA缓冲区的子流

**说明**

通过snd\_pcm\_lib\_malloc\_pages()分配的DMA缓冲区。

**返回**

成功则返回零，否则返回负错误代码。

## int snd\_pcm\_lib\_free\_vmalloc\_buffer(struct snd\_pcm\_substream \* substream)

释放vmalloc缓冲区

**参数**

struct snd\_pcm\_substream \* substream

通过snd\_pcm\_lib\_alloc\_vmalloc\_buffer()分配缓冲区的子流

**说明**

返回成功或者返回错误的代码。

## struct page \* snd\_pcm\_lib\_get\_vmalloc\_page(struct snd\_pcm\_substream \* substream，unsigned long offset)

将vmalloc缓冲区偏移量映射到页面结构

**参数**

struct snd\_pcm\_substream \* substream

通过snd\_pcm\_lib\_alloc\_vmalloc\_buffer()分配缓冲区的子流

unsigned long offset

缓冲区该位置的偏移量

**说明**

该函数用作PCM操作中的页面回调。

**返回**

页面结构或NULL（失败时）。

## void snd\_device\_initialize(struct device \* dev，struct snd\_card \* card)

为声音设备初始化结构设备

**参数**

struct device \* dev

要初始化的设备

struct snd\_card \* card

要分配的卡，可选

## int snd\_card\_new(struct device \* parent，int idx，const char \* xid，struct module \* module，int extra\_size，struct snd\_card \*\* card\_ret)

创建并初始化音频卡结构

**参数**

struct device \* parent

父设备对象

int idx

卡索引（地址）[0...（SNDRV\_CARDS-1)]

const char \* xid

卡标识（ASCII字符串）

struct module \* module

顶层模块用于锁定

int extra\_size

分配此额外大小后，分配声卡结构

struct snd\_card \*\* card\_ret

指向存储已创建的卡实例的指针

**说明**

创建和初始化声卡结构。

该函数通过kzalloc分配snd\_card实例，其中包含驱动程序可自由使用的给定空间。所分配的结构存储在给定的card\_ret指针中。

**返回**

成功为零，否则为负错误代码。

## int snd\_card\_disconnect(struct snd\_card \* card)

从文件操作（用户空间）中断开所有API

**参数**

struct snd\_card \* card

声卡结构

**说明**

从文件操作（用户空间）中断开所有API。

**返回**

零，否则为负错误代码。

**注意**

当前实现将所有活动文件-> f\_op替换为特殊的虚拟文件操作（它们除了释放外什么也不做）。

## void snd\_card\_disconnect\_sync(struct snd\_card \* card)

断开卡并等待文件关闭

**参数**

struct snd\_card \* card

要断开的卡对象

**说明**

这会调用snd\_card\_disconnect（）以断开所有相关组件，并等待所有挂起的文件关闭。它确保所有来自用户空间的访问都已完成，以便驱动程序可以优雅地释放其资源。

## int snd\_card\_free\_when\_closed(struct snd\_card \* card)

断开卡，最终进行释放

**参数**

struct snd\_card \* card

声卡结构

**说明**

与snd\_card\_free（）不同，此函数不会立即尝试释放卡资源，而是首先尝试断开卡。当卡仍在使用时，该函数在释放资源之前返回。当refcount达到零时，卡资源将被释放。

## int snd\_card\_free(struct snd\_card \* card)

释放给定的声卡结构

**参数**

struct snd\_card \* card

声卡结构

**说明**

此函数会自动释放声卡结构和所有分配的设备。也就是说，您不需要自己释放设备。

此功能将等待所有资源正确释放。

**返回**

零。释放与给定声卡相关联的所有设备，并释放关联的控制接口。

## void snd\_card\_set\_id(struct snd\_card \* card，const char \* nid)

设置卡标识名称

**参数**

struct snd\_card \* card

声卡结构

const char \* nid

新标识字符串

**说明**

此函数设置卡标识并检查名称冲突。

## int snd\_card\_add\_dev\_attr(struct snd\_card \* card，const struct attribute\_group \* group)

向卡添加新的sysfs属性组

**参数**

struct snd\_card \* card

卡实例

const struct attribute\_group \* group

要添加的属性组

## int snd\_card\_register(struct snd\_card \* card)

注册声卡

**参数**

struct snd\_card \* card

声卡结构

**说明**

此函数注册分配给声卡的所有设备。在调用此函数之前，ALSA控制接口会被外部访问所阻塞。因此，您应该在卡的初始化结束时调用此函数。

**返回**

如果注册失败，则为零，否则为负错误代码。

## int snd\_component\_add(struct snd\_card \* card，const char \* component)

添加组件字符串

**参数**

struct snd\_card \* card

声卡结构

const char \* component

组件ID字符串

**说明**

此函数将组件ID字符串添加到支持列表中。组件可以从alsa-lib中引用。

**返回**

如果成功为零，否则为负错误代码。

## int snd\_card\_file\_add(struct snd\_card \* card，struct file \* file)

将文件添加到卡的文件列表中

**参数**

struct snd\_card \* card

声卡结构

struct file \* file

文件指针

**说明**

此函数将文件添加到卡的文件链表中。此链表用于跟踪连接状态并避免热插拔释放繁忙资源。

**返回**

零或负错误代码。

## int snd\_card\_file\_remove(struct snd\_card \* card，struct file \* file)

从文件列表中删除文件

**参数**

struct snd\_card \* card

声卡结构

struct file \* file

文件指针

**说明**

此函数从先前通过snd\_card\_file\_add（）函数添加到卡中的文件中删除文件。如果已删除所有文件并且先前调用了snd\_card\_free\_when\_closed（），则它会处理挂起的资源释放。

**返回**

为零或负错误代码。

## int snd\_power\_wait(struct snd\_card \* card，unsigned int power\_state)

等待电源状态改变。

**参数**

struct snd\_card \* card

声卡结构

unsigned int power\_state

期望的电源状态

**说明**

等待电源状态改变。

**返回**

如果成功为零，否则为负错误代码。

## void snd\_dma\_program(unsigned long dma，unsigned long addr，unsigned int size，unsigned short mode)

为ISA DMA传输编程

**参数**

unsigned long dma

DMA编号

unsigned long addr

缓冲区的物理地址

unsigned int size

DMA传输大小

unsigned short mode

DMA传输模式，DMA\_MODE\_XXX

**说明**

为给定缓冲区编程ISA DMA传输。

## void snd\_dma\_disable(unsigned long dma)

停止ISA DMA传输

**参数**

unsigned long dma

DMA编号

**说明**

停止ISA DMA传输。

## unsigned int snd\_dma\_pointer(unsigned long dma，unsigned int size)

返回DMA传输缓冲区中当前指针的字节数

**参数**

unsigned long dma

DMA编号

unsigned int size

DMA传输大小

**返回**

DMA传输缓冲区中当前指针的字节数。

## void snd\_ctl\_notify（struct snd\_card \*card，unsigned int mask，struct snd\_ctl\_elem\_id \*id）

发送控制变化通知到用户空间

**参数**

struct snd\_card \*card

要发送通知的声卡

unsigned int mask

事件掩码，SNDRV\_CTL\_EVENT\_\*

struct snd\_ctl\_elem\_id \*id

要发送通知的控件元素ID

**说明**

此函数添加具有给定ID和掩码的事件记录，附加到列表并唤醒用户空间进行通知。这可以在原子上下文中调用。

## struct snd\_kcontrol \*snd\_ctl\_new1（const struct snd\_kcontrol\_new \*ncontrol，void \*private\_data）

从模板创建控制实例

**参数**

const struct snd\_kcontrol\_new \*ncontrol

初始化记录

void \*private\_data

要设置的私有数据

**说明**

从给定的模板分配新的struct snd\_kcontrol实例并进行初始化。当ncontrol的访问字段为0时，它被假定为READWRITE访问。当计数字段为0时，它被视为一个。

**返回**

新生成实例的指针，或者失败时为NULL。

## void snd\_ctl\_free\_one（struct snd\_kcontrol \*kcontrol）

释放控制实例

**参数**

struct snd\_kcontrol \*kcontrol

要释放的控制实例

**说明**

释放通过snd\_ctl\_new（）或snd\_ctl\_new1（）创建的控制实例。在将控制添加到卡之后不要调用此函数。

## int snd\_ctl\_add（struct snd\_card \*card，struct snd\_kcontrol \*kcontrol）

将控制实例添加到卡

**参数**

struct snd\_card \*card

卡实例

struct snd\_kcontrol \*kcontrol

要添加的控件实例

**说明**

将通过snd\_ctl\_new（）或snd\_ctl\_new1（）创建的控制实例添加到给定的卡上。还分配一个用于快速搜索的唯一numid。

它会自动释放无法添加的控件。

**返回**

成功为零，失败为负错误代码。

## int snd\_ctl\_replace（struct snd\_card \*card，struct snd\_kcontrol \*kcontrol，bool add\_on\_replace）

替换卡的控制实例

**参数**

struct snd\_card \*card

卡实例

struct snd\_kcontrol \*kcontrol

要替换的控制实例

bool add\_on\_replace

如果尚未添加，则添加控件

**说明**

替换给定的控件。如果给定的控件不存在并且设置了add\_on\_replace标志，则添加控件。如果控件存在，则首先销毁控件。

它会自动释放无法添加或替换的控件。

**返回**

成功为零，失败为负错误代码。

## int snd\_ctl\_remove（struct snd\_card \*card，struct snd\_kcontrol \*kcontrol）

从卡中删除控制并释放它

**参数**

struct snd\_card \*card

卡实例

struct snd\_kcontrol \*kcontrol

要删除的控制实例

**说明**

从卡中删除控件，然后释放实例。您不需要调用snd\_ctl\_free\_one（）。您必须在写锁定 - down\_write（card->controls\_rwsem）中。

**返回**

成功为0，失败为负错误代码。

## int snd\_ctl\_remove\_id（struct snd\_card \*card，struct snd\_ctl\_elem\_id \*id）

删除给定ID的控制并释放它

**参数**

struct snd\_card \*card

卡实例

struct snd\_ctl\_elem\_id \*id

要删除的控制ID

**说明**

查找具有给定ID的控制实例，从卡列表中删除它并释放它。

**返回**

成功为0，失败为负错误代码。

## int snd\_ctl\_activate\_id（struct snd\_card \*card，struct snd\_ctl\_elem\_id \*id，int active）

激活/停用给定ID的控制

**参数**

struct snd\_card \*card

卡实例

struct snd\_ctl\_elem\_id \*id

要激活/停用的控制ID

int active

非零以激活

**说明**

查找具有给定ID的控制实例，并激活或停用控制以及通知（如果更改）。所给ID数据填满了完整信息。

**返回**

如果未更改，则为0，如果更改，则为1，否则为负错误代码。

## int snd\_ctl\_rename\_id（struct snd\_card \*card，struct snd\_ctl\_elem\_id \*src\_id，struct snd\_ctl\_elem\_id \*dst\_id）

替换卡中控件的ID

**参数**

struct snd\_card \*card

卡实例

struct snd\_ctl\_elem\_id \*src\_id

旧ID

struct snd\_ctl\_elem\_id \*dst\_id

新ID

**说明**

从卡中查找具有旧ID的控件，并将ID替换为新ID。

**返回**

成功为零，失败为负错误代码。

## struct snd\_kcontrol \*snd\_ctl\_find\_numid（struct snd\_card \*card，unsigned int numid）

查找具有给定数字ID的控制实例

**参数**

struct snd\_card \*card

卡实例

unsigned int numid

要搜索的数字ID

**说明**

从卡中查找具有给定数字ID的控制实例。

调用方必须在调用此函数之前降低card->controls\_rwsem（如果可能发生竞争条件）。

**返回**

如果找到，则为实例指针，否则为NULL。

## struct snd\_kcontrol \*snd\_ctl\_find\_id（struct snd\_card \*card，struct snd\_ctl\_elem\_id \*id）

查找具有给定ID的控制实例

**参数**

结构snd\_card \*card

卡实例

结构snd\_ctl\_elem\_id \*id

要查找的id

**说明**

从卡中找到具有给定id的控制实例。

调用者必须在调用此函数之前下降card->controls\_rwsem（如果可能发生竞争）。

**返回**

如果找到，则为实例的指针，否则为NULL。

## int snd\_ctl\_register\_ioctl(snd\_kctl\_ioctl\_func\_t fcn)

注册设备特定的控制ioctls

**参数**

snd\_kctl\_ioctl\_func\_t fcn

ioctl回调函数

**说明**

从pcm.c、hwdep.c等每个设备管理器调用。

## int snd\_ctl\_register\_ioctl\_compat(snd\_kctl\_ioctl\_func\_t fcn)

注册设备特定的32位兼容控制ioctls

**参数**

snd\_kctl\_ioctl\_func\_t fcn

ioctl回调函数

## int snd\_ctl\_unregister\_ioctl(snd\_kctl\_ioctl\_func\_t fcn)

注销设备特定的控制ioctls

**参数**

snd\_kctl\_ioctl\_func\_t fcn

要注销的ioctl回调函数

## int snd\_ctl\_unregister\_ioctl\_compat(snd\_kctl\_ioctl\_func\_t fcn)

注销设备特定的兼容32位控制ioctls

**参数**

snd\_kctl\_ioctl\_func\_t fcn

要注销的ioctl回调函数

## int snd\_ctl\_boolean\_mono\_info(struct snd\_kcontrol \*kcontrol, struct snd\_ctl\_elem\_info \*uinfo)

单通道标准布尔信息回调的助手函数

**参数**

struct snd\_kcontrol \*kcontrol

kcontrol实例

结构snd\_ctl\_elem\_info \*uinfo

要存储的信息

**说明**

这是可以用作具有单个单声道通道的标准布尔控件的信息回调的函数。

## int snd\_ctl\_boolean\_stereo\_info(struct snd\_kcontrol \*kcontrol, struct snd\_ctl\_elem\_info \*uinfo)

具有立体声两个通道的标准布尔信息回调的助手函数

**参数**

struct snd\_kcontrol \*kcontrol

kcontrol实例

结构snd\_ctl\_elem\_info \*uinfo

要存储的信息

**说明**

这是可以用作具有立体声两个通道的标准布尔控件的信息回调的函数。

## int snd\_ctl\_enum\_info(struct snd\_ctl\_elem\_info \*info, unsigned int channels, unsigned int items, const char \*const names)

填充enum控件的信息结构

**参数**

struct snd\_ctl\_elem\_info \*info

要填充的结构

unsigned int channels

控件通道的数量；通常为一

unsigned int items

控件值的数量；也是名称的大小

const char \*const names

包含所有控件值名称的数组

**说明**

将所有必需的字段设置为相应的值。 如果控件的访问性不是默认值（可读可写），则调用者必须填写info->access。

**返回**

零。

## void snd\_pcm\_set\_ops（struct snd\_pcm \* pcm，int direction，const struct snd\_pcm\_ops \* ops）

设置PCM运算符

**参数**

struct snd\_pcm \* pcm

pcm实例

int direction

流方向，SNDRV\_PCM\_STREAM\_XXX

const struct snd\_pcm\_ops \* ops

操作员表

**说明**

将给定的PCM运算符设置到PCM实例中。

## void snd\_pcm\_set\_sync(struct snd\_pcm\_substream \* substream)

设置PCM同步ID

**参数**

struct snd\_pcm\_substream \* substream

pcm子流

**说明**

为卡设置PCM同步标识符。

## int snd\_interval\_refine(struct snd\_interval \* i, const struct snd\_interval \* v)

配置器的区间值精炼

**参数**

struct snd\_interval \* i

要精炼的区间值

const struct snd\_interval \* v

要引用的区间值

**说明**

使用参考值细化区间值。 区间更改为满足两个区间的范围。 评估区间状态（最小值，最大值，整数等）。

**返回**

如果该价值被改变，则为正值，如果该价值未被改变，则为零，或者是负错误代码。

## int snd\_interval\_ratnum(struct snd\_interval \* i,unsigned int rats\_count,const struct snd\_ratnum \* rats,unsigned int \* nump,unsigned int \* denp)

区间值精炼

**参数**

struct snd\_interval \* i

要细化的区间

unsigned int rats\_count

率数

const struct snd\_ratnum \* rats

ratnum\_t数组

unsigned int \* nump

指针以存储结果分子

unsigned int \* denp

指针以存储结果分母

**返回**

如果该价值被改变，则为正值，如果该价值未被改变，则为零，或者是负错误代码。

## int snd\_interval\_list(struct snd\_interval \* i,unsigned int count,const unsigned int \* list,unsigned int mask)

从列表中筛选区间值

**参数**

struct snd\_interval \* i

要精炼的区间值

unsigned int count

列表中的元素数量

const unsigned int \* list

价值清单

unsigned int mask

要评估的位掩码

**说明**

从列表中细化区间值。 当mask不为零时，仅计算与位1对应的元素。

**返回**

如果该价值被改变，则为正值，如果该价值未被改变，则为零，或者是负错误代码。

## int snd\_interval\_ranges(struct snd\_interval \* i,unsigned int count,const struct snd\_interval \* ranges,unsigned int mask)

从范围列表中筛选区间值

**参数**

struct snd\_interval \* i

要细化的区间值

unsigned int count

范围列表中的元素数量

const struct snd\_interval \* ranges

范围列表

unsigned int mask

要评估的位掩码

**说明**

从范围列表中细化区间值。 当mask不为零时，仅计算与位1对应的元素。

**返回**

如果该价值被改变，则为正值，如果该价值未被改变，则为零，或者是负错误代码。

## int snd\_pcm\_hw\_rule\_add（struct snd\_pcm\_runtime \* runtime，unsigned int cond，int var，snd\_pcm\_hw\_rule\_func\_t func，void \* private，int dep，...）

添加硬件约束规则

**参数**

struct snd\_pcm\_runtime \* runtime

PCM运行时实例

unsigned int cond

条件位

int var

要评估的变量

snd\_pcm\_hw\_rule\_func\_t func

评估函数

空的\* private

传递给函数的私有数据指针

int dep

依赖变量

...

可变参数

**返回**

如果成功，则为零，如果失败，则为负面错误代码。

## int snd\_pcm\_hw\_constraint\_mask64(struct snd\_pcm\_runtime \* runtime，snd\_pcm\_hw\_param\_t var，u\_int64\_t mask)

应用给定的位图掩码约束

**参数**

struct snd\_pcm\_runtime \* runtime

PCM运行时实例

snd\_pcm\_hw\_param\_t var

应用掩码的hw\_params变量

u\_int64\_t mask

64位位图掩码

**说明**

将所给位图掩码的约束应用到64位掩码参数中。

**返回**

如果成功，则为零，如果失败，则为负面错误代码。

## int snd\_pcm\_hw\_constraint\_integer(struct snd\_pcm\_runtime \* runtime，snd\_pcm\_hw\_param\_t var)

将整数约束应用于间隔

**参数**

struct snd\_pcm\_runtime \* runtime

PCM运行时实例

snd\_pcm\_hw\_param\_t var

要应用整数约束的hw\_params变量

**说明**

将整数约束应用于间隔参数。

**返回**

如果值更改，则为正，如果值未更改，则为零，如果值未更改，则为负面错误代码。

## int snd\_pcm\_hw\_constraint\_minmax(struct snd\_pcm\_runtime \* runtime，snd\_pcm\_hw\_param\_t var，unsigned int min，unsigned int max)

将min / max范围约束应用于间隔

**参数**

struct snd\_pcm\_runtime \* runtime

PCM运行时实例

snd\_pcm\_hw\_param\_t var

应用范围的hw\_params变量

unsigned int min

最小值

unsigned int max

最大值

**说明**

将min / max范围约束应用到间隔参数。

**返回**

如果值更改，则为正，如果值未更改，则为零，如果值未更改，则为负面错误代码。

## int snd\_pcm\_hw\_constraint\_list(struct snd\_pcm\_runtime \* runtime，unsigned int cond，snd\_pcm\_hw\_param\_t var，const struct snd\_pcm\_hw\_constraint\_list \* l)

将约束列表应用于参数

**参数**

struct snd\_pcm\_runtime \* runtime

PCM运行时实例

unsigned int cond

条件位

snd\_pcm\_hw\_param\_t var

要应用列表约束的hw\_params变量

conststruct snd\_pcm\_hw\_constraint\_list \* l

清单

**说明**

将约束列表应用到间隔参数。

**返回**

如果成功，则为零，如果失败，则为负面错误代码。

## int snd\_pcm\_hw\_constraint\_ranges(struct snd\_pcm\_runtime \* runtime，unsigned int cond，snd\_pcm\_hw\_param\_t var，const struct snd\_pcm\_hw\_constraint\_ranges \* r)

将一系列范围约束应用于参数

**参数**

struct snd\_pcm\_runtime \* runtime

PCM运行时实例

unsigned int cond

条件位

snd\_pcm\_hw\_param\_t var

应用范围列表的hw\_params变量

conststruct snd\_pcm\_hw\_constraint\_ranges \* r

范围

**说明**

将一系列范围约束应用于间隔参数。

**返回**

如果成功，则为零，如果失败，则为负面错误代码。

## int snd\_pcm\_hw\_constraint\_ratnums(struct snd\_pcm\_runtime \* runtime，unsigned int cond，snd\_pcm\_hw\_param\_t var，const struct snd\_pcm\_hw\_constraint\_ratnums \* r)

将ratnums约束应用于参数

**参数**

struct snd\_pcm\_runtime \* runtime

PCM运行时实例

unsigned int cond

条件位

snd\_pcm\_hw\_param\_t var

hw\_params变量可以应用ratnums约束

conststruct snd\_pcm\_hw\_constraint\_ratnums \* r

struct snd\_ratnums constriants

**返回**

如果成功，则为零，如果失败，则为负面错误代码。

## int snd\_pcm\_hw\_constraint\_ratdens(struct snd\_pcm\_runtime \* runtime，unsigned int cond，snd\_pcm\_hw\_param\_t var，const struct snd\_pcm\_hw\_constraint\_ratdens \* r)

将ratdens约束应用于参数

**参数**

struct snd\_pcm\_runtime \* runtime

PCM运行时实例

unsigned int cond

条件位

snd\_pcm\_hw\_param\_t var

hw\_params变量可以应用ratdens约束

const struct snd\_pcm\_hw\_constraint\_ratdens \* r

struct snd\_ratdens constriants

**返回**

如果成功，则为零，如果失败，则为负面错误代码。

## int snd\_pcm\_hw\_constraint\_msbits(struct snd\_pcm\_runtime \* runtime，unsigned int cond，unsigned int width，unsigned int msbits)

添加一个硬件约束msbits规则

**参数**

struct snd\_pcm\_runtime \* runtime

PCM运行时实例

unsigned int cond

条件位

unsigned int width

样本位宽

unsigned int msbits

msbits宽度

**说明**

此约束将在选择指定宽度的采样格式时设置最高有效位数（msbits）。如果宽度设置为0，则将为宽度大于指定msbits的任何采样格式设置msbits。

**返回**

如果成功，则为零，如果失败，则为负面错误代码。

## int snd\_pcm\_hw\_constraint\_step(struct snd\_pcm\_runtime \* runtime，unsigned int cond，snd\_pcm\_hw\_param\_t var，unsigned long step)

添加一个硬件限制步骤规则

**参数**

struct snd\_pcm\_runtime \* runtime

PCM运行时实例

unsigned int cond

条件位

snd\_pcm\_hw\_param\_t var

hw\_params变量可以应用步骤约束

unsigned long step

步骤大小

**返回**

如果成功，则为零，如果失败，则为负面错误代码。

## int snd\_pcm\_hw\_constraint\_pow2(struct snd\_pcm\_runtime \* runtime，unsigned int cond，snd\_pcm\_hw\_param\_t var)

添加一个hw约束2的幂规则

**参数**

struct snd\_pcm\_runtime \* runtime

PCM运行时实例

unsigned int cond

条件位

snd\_pcm\_hw\_param\_t var

可以应用2的幂次方约束的hw\_params变量

**返回**

如果成功，则为零，如果失败，则为负面错误代码。

## int snd\_pcm\_hw\_rule\_noresample(struct snd\_pcm\_runtime \* runtime，unsigned int base\_rate)

添加一条规则，允许禁用hw重新采样

**参数**

struct snd\_pcm\_runtime \* runtime

PCM运行时实例

unsigned int base\_rate

硬件不重新采样的速率

**返回**

如果成功返回零，否则返回负错误代码。

## int snd\_pcm\_hw\_param\_value(const struct snd\_pcm\_hw\_params \* params, snd\_pcm\_hw\_param\_t var, int \* dir)

返回 params 字段 var 的值

**参数**

const struct snd\_pcm\_hw\_params \* params

hw\_params 实例

snd\_pcm\_hw\_param\_t var

要检索的参数

int \* dir

指向方向（-1、0、1）的指针或 NULL

**返回**

如果 var 在由 params 定义的配置空间中是固定的，则返回 var 字段的值。否则返回 -EINVAL。

## int snd\_pcm\_hw\_param\_first(struct snd\_pcm\_substream \* pcm, struct snd\_pcm\_hw\_params \* params, snd\_pcm\_hw\_param\_t var, int \* dir)

微调配置空间并返回最小值

**参数**

struct snd\_pcm\_substream \* pcm

PCM 实例

struct snd\_pcm\_hw\_params \* params

hw\_params 实例

snd\_pcm\_hw\_param\_t var

要检索的参数

int \* dir

指向方向（-1、0、1）的指针或 NULL

**说明**

在由 params 定义的配置空间中，从 var 移除所有 > 最小值的值。相应地减小配置空间。

**返回**

如果成功，返回最小值，否则返回负错误代码。

## int snd\_pcm\_hw\_param\_last(struct snd\_pcm\_substream \* pcm, struct snd\_pcm\_hw\_params \* params, snd\_pcm\_hw\_param\_t var, int \* dir)

微调配置空间并返回最大值

**参数**

struct snd\_pcm\_substream \* pcm

PCM 实例

struct snd\_pcm\_hw\_params \* params

hw\_params 实例

snd\_pcm\_hw\_param\_t var

要检索的参数

int \* dir

指向方向（-1、0、1）的指针或 NULL

**说明**

在由 params 定义的配置空间中，从 var 移除所有 < 最大值的值。相应地减小配置空间。

**返回**

如果成功，返回最大值，否则返回负错误代码。

## int snd\_pcm\_lib\_ioctl(struct snd\_pcm\_substream \* substream, unsigned int cmd, void \* arg)

通用 PCM ioctl 回调函数

**参数**

struct snd\_pcm\_substream \* substream

PCM substream 实例

unsigned int cmd

ioctl 命令

void \* arg

ioctl 参数

**说明**

处理 PCM 的通用 ioctl 命令。可以作为 PCM 操作的 ioctl 回调传递。

**返回**

如果成功，返回零，否则返回负错误代码。

## void snd\_pcm\_period\_elapsed(struct snd\_pcm\_substream \* substream)

更新下一个周期的 PCM 状态

**参数**

struct snd\_pcm\_substream \* substream

PCM substream 实例

**说明**

当 PCM 处理周期大小时，从中断处理程序调用此函数。它将更新当前指针、唤醒睡眠者等。

即使自上次调用以来经过了多个时期，您也只需调用此函数一次。

## int snd\_pcm\_add\_chmap\_ctls(struct snd\_pcm \* pcm, int stream, const struct snd\_pcm\_chmap\_elem \* chmap, int max\_channels, unsigned long private\_value, struct snd\_pcm\_chmap \*\* info\_ret)

创建通道映射控制元素

**参数**

struct snd\_pcm \* pcm

已分配的 PCM 实例

int stream

流方向

const struct snd\_pcm\_chmap\_elem \* chmap

通道映射元素（用于查询）

int max\_channels

流的最大通道数

unsigned long private\_value

传递给每个 kcontrol 的 private\_value 字段的值

struct snd\_pcm\_chmap \*\* info\_ret

如果非 NULL，则存储struct snd\_pcm\_chmap 实例

**说明**

创建分配给给定 PCM 流的通道映射控制元素。

**返回**

如果成功，返回零，否则返回负错误值。

## int snd\_hwdep\_new(struct snd\_card \* card, char \* id, int device, struct snd\_hwdep \*\* rhwdep)

创建新的 hwdep 实例

**参数**

struct snd\_card \* card

卡实例

char \* id

id 字符串

int device

设备索引（从零开始）

struct snd\_hwdep \*\* rhwdep

用于存储新 hwdep 实例的指针

**说明**

在卡上创建具有给定索引的新 hwdep 实例。必须手动设置返回的实例的回调函数（hwdep->ops）。

**返回**

如果成功，返回零，否则返回负错误代码。

## void snd\_pcm\_stream\_lock(struct snd\_pcm\_substream \*substream)

锁定 PCM 流

**参数**

struct snd\_pcm\_substream \*substream

PCM substream

**说明**

这将锁定 PCM 流的自旋锁或互斥锁，具体取决于给定 substream 的 nonatomic 标志。这也会获取全局链接 rw 锁（或 rw 信号量），以避免与链接的流竞争。

## void snd\_pcm\_stream\_unlock(struct snd\_pcm\_substream \*substream)

解除锁定 PCM 流

**参数**

struct snd\_pcm\_substream \*substream

PCM substream

**说明**

这将解锁通过 snd\_pcm\_stream\_lock() 锁定的 PCM 流。

## void snd\_pcm\_stream\_lock\_irq(struct snd\_pcm\_substream \* substream)

锁定 PCM 流

**参数**

struct snd\_pcm\_substream \* substream

PCM substream

**说明**

这将像 snd\_pcm\_stream\_lock() 一样锁定 PCM 流，并禁用本地 IRQ（仅在 nonatomic 为 false 时）。对于非原子操作的情况，这与 snd\_pcm\_stream\_lock() 相同。

## void snd\_pcm\_stream\_unlock\_irq(struct snd\_pcm\_substream \* substream)

解锁 PCM 流

**参数**

struct snd\_pcm\_substream \* substream

PCM substream

**说明**

这是 snd\_pcm\_stream\_lock\_irq() 的对应函数。

## void snd\_pcm\_stream\_unlock\_irqrestore(struct snd\_pcm\_substream \* substream, unsigned long flags)

解锁 PCM 流

**参数**

struct snd\_pcm\_substream \* substream

PCM substream

unsigned long flags

irq 标志

**说明**

这是 snd\_pcm\_stream\_lock\_irqsave() 的对应函数。

## int snd\_pcm\_stop(struct snd\_pcm\_substream \* substream, snd\_pcm\_state\_t state)

尝试停止 substream 组中的所有正在运行的流

**参数**

struct snd\_pcm\_substream \* substream

PCM substream 实例

snd\_pcm\_state\_t state

停止流后的 PCM 状态

**说明**

然后将每个流的状态无条件更改为给定状态。

**返回**

如果成功，返回零，否则返回负错误代码。

## int snd\_pcm\_stop\_xrun(struct snd\_pcm\_substream \* substream)

停止运行中的流作为XRUN

**参数**

struct snd\_pcm\_substream \* substream

PCM子流实例

**说明**

这个函数停止给定的运行子流（和所有链接的子流）作为XRUN。不像snd\_pcm\_stop()，这个函数自己占有子流锁。

**返回**

如果成功则返回0，否则返回负错误代码。

## int snd\_pcm\_suspend(struct snd\_pcm\_substream \* substream)

触发所有链接流的挂起

**参数**

struct snd\_pcm\_substream \* substream

PCM子流

**说明**

此调用后，所有流都变成SUSPENDED状态。

**返回**

如果成功（或substream是NULL），则返回0；如果失败，则返回负错误代码。

## int snd\_pcm\_suspend\_all(struct snd\_pcm \* pcm)

触发给定pcm中所有子流的挂起

**参数**

struct snd\_pcm \* pcm

PCM实例

**说明**

此调用后，所有流都变成SUSPENDED状态。

**返回**

如果成功（或pcm是NULL），则返回0；如果失败，则返回负错误代码。

## int snd\_pcm\_kernel\_ioctl(struct snd\_pcm\_substream \* substream, unsigned int cmd, void \* arg)

在内核空间执行PCM ioctl

**参数**

struct snd\_pcm\_substream \* substream

PCM子流

unsigned int cmd

IOCTL cmd

void \* arg

IOCTL参数

**说明**

此函数主要为OSS层和USB模拟设备驱动程序提供，并且它只允许有限的ioctl集合（hw\_params、sw\_params、prepare、start、drain、drop、forward）。

## int snd\_pcm\_lib\_default\_mmap(struct snd\_pcm\_substream \* substream, struct vm\_area\_struct \* area)

默认PCM数据mmap函数

**参数**

struct snd\_pcm\_substream \* substream

PCM子流

struct vm\_area\_struct \* area

VMA

**说明**

这是PCM数据的默认mmap处理程序。当mmap pcm\_ops为NULL时，隐式调用此函数。

## int snd\_pcm\_lib\_mmap\_iomem(struct snd\_pcm\_substream \* substream, struct vm\_area\_struct \* area)

默认的I/O mem PCM数据mmap函数

**参数**

struct snd\_pcm\_substream \* substream

PCM子流

struct vm\_area\_struct \* area

VMA

**说明**

当您的硬件将iomapped页面用作硬件缓冲区并希望将其映射到内存中时，请将此函数作为mmap pcm\_ops。请注意，这仅适用于有限体系结构。

## void \* snd\_malloc\_pages(size\_t size, gfp\_t gfp\_flags)

按给定大小分配页面

**参数**

size\_t size

要分配的大小（字节）

gfp\_t gfp\_flags

分配条件，GFP\_XXX

**说明**

分配给定大小的物理连续页面。

**返回**

缓冲区的指针，如果没有足够的内存，则为NULL。

## void snd\_free\_pages(void \* ptr, size\_t size)

释放分配的页面

**参数**

void \* ptr

要释放的缓冲区指针

size\_t size

分配的缓冲区大小

**说明**

通过snd\_malloc\_pages()释放已分配的缓冲区。

## int snd\_dma\_alloc\_pages(int type, struct device \* device, size\_t size, struct snd\_dma\_buffer \* dmab)

根据给定类型分配缓冲区区域

**参数**

int type

DMA缓冲区类型

struct device \* device

设备指针

size\_t size

要分配的缓冲区大小

struct snd\_dma\_buffer \* dmab

缓冲区分配记录，用于存储分配的数据

**说明**

为相应的缓冲区类型调用内存分配器函数。

**返回**

如果成功分配了给定大小的缓冲区，则为零，否则为负值。

## int snd\_dma\_alloc\_pages\_fallback(int type, struct device \* device, size\_t size, struct snd\_dma\_buffer \* dmab)

根据给定类型以回退方式分配缓冲区区域

**参数**

int type

DMA缓冲区类型

struct device \* device

设备指针

size\_t size

要分配的缓冲区大小

struct snd\_dma\_buffer \* dmab

缓冲区分配记录，用于存储分配的数据

**说明**

为相应的缓冲区类型调用内存分配器函数。当没有空间时，此函数会缩小大小并尝试重新分配。实际分配的大小存储在res\_size参数中。

**返回**

如果成功分配了给定大小的缓冲区，则为零，否则为负值。

## void snd\_dma\_free\_pages(struct snd\_dma\_buffer \* dmab)

释放已分配的缓冲区

**参数**

struct snd\_dma\_buffer \* dmab

要释放的缓冲区分配记录

**说明**

通过snd\_dma\_alloc\_pages()释放已分配的缓冲区。

# 缓冲库

帧缓冲驱动程序严重依赖于四个数据结构。这些结构在include/linux/fb.h中声明。它们是fb\_info、fb\_var\_screeninfo、fb\_fix\_screeninfo和fb\_monospecs。后三者可在用户空间和内核空间之间可用。

fb\_info定义特定视频卡的当前状态。在fb\_info内部，存在一个名为fb\_ops的结构，其中包含使fbdev和fbcon工作所需的函数。fb\_info仅对内核可见。

fb\_var\_screeninfo用于说明用户定义的视频卡功能。使用fb\_var\_screeninfo，可以定义分辨率等内容。

下一个结构是fb\_fix\_screeninfo。这定义了在设置模式时创建的卡的属性，并且不能以其他方式更改。一个很好的例子是帧缓冲存储器的开始。这“锁定”了帧缓冲存储器的地址，使其无法更改或移动。

最后一个结构是fb\_monospecs。在旧的API中，fb\_monospecs没有太大的重要性。这允许禁止诸如在固定频率监视器上设置800x600模式之类的事情。使用新API时，fb\_monospecs可以防止这些情况发生，如果使用正确，可以防止监视器被烧坏。在内核2.5.x之前，fb\_monospecs不会有用。

## 帧缓冲存储器

### int register\_framebuffer（struct fb\_info \*fb\_info）

注册帧缓冲设备

**参数**

struct fb\_info \*fb\_info

帧缓冲信息结构

**说明**

注册帧缓冲设备fb\_info。

返回负errno表示错误，返回零表示成功。

### int unregister\_framebuffer（struct fb\_info \*fb\_info）

释放帧缓冲设备

**参数**

struct fb\_info \*fb\_info

帧缓冲信息结构

**说明**

取消注册帧缓冲设备fb\_info。

返回负errno表示错误，返回零表示成功。

此函数还将通知帧缓冲控制台释放驱动程序。

这应该在驱动程序的module\_exit（）函数中调用。如果在module\_exit（）之外调用此操作，请确保驱动程序实现了fb\_open（）和fb\_release（）以检查是否有进程正在使用设备。

### void fb\_set\_suspend（struct fb\_info \*info，int state）

低级驱动程序发出挂起信号

**参数**

struct fb\_info \*info

受影响的帧缓冲

int state

0 =恢复，！= 0 =暂停

**说明**

这是由低级驱动程序用于向核心和客户端发出挂起/恢复信号的。必须使用控制台信号量调用。

## 帧缓冲区颜色图

### void fb\_dealloc\_cmap（struct fb\_cmap \*cmap）

取消分配颜色映射

**参数**

struct fb\_cmap \*cmap

帧缓冲颜色映射结构

**说明**

释放以前使用fb\_alloc\_cmap（）分配的颜色映射。

### int fb\_copy\_cmap（const struct fb\_cmap \*from，struct fb\_cmap \*to）

复制颜色映射

**参数**

const struct fb\_cmap \*from

帧缓冲颜色映射结构

struct fb\_cmap \*to

帧缓冲颜色映射结构

**说明**

将颜色映射的内容从from复制到to。

### int fb\_set\_cmap（struct fb\_cmap \*cmap，struct fb\_info \*info）

设置颜色映射表

**参数**

struct fb\_cmap \*cmap

帧缓冲颜色映射结构

struct fb\_info \*info

帧缓冲信息结构

**说明**

为设备info设置颜色映射cmap。

返回负errno表示错误，返回零表示成功。

### const struct fb\_cmap \*fb\_default\_cmap（int len）

获取默认颜色映射表

**参数**

int len

每个深度的调色板大小

**说明**

获取特定屏幕深度的默认颜色映射表。len是特定屏幕深度的调色板大小。

返回指向帧缓冲颜色映射结构的指针。

### void fb\_invert\_cmaps（void）

反转所有默认颜色映射表

**参数**

void

没有参数

**说明**

反转所有默认颜色映射表。

## 帧缓冲视频模式数据库

### int fb\_try\_mode（struct fb\_var\_screeninfo \*var，struct fb\_info \*info，const struct fb\_videomode \*mode，unsigned int bpp）

测试视频模式

**参数**

struct fb\_var\_screeninfo \*var

帧缓冲用户定义的显示部分

struct fb\_info \*info

帧缓冲信息结构

const struct fb\_videomode \*mode

帧缓冲视频模式结构

unsigned int bpp

每像素的色深（以位/像素为单位）

**说明**

尝试测试视频模式，以检查它是否在设备信息中有效。

返回1表示成功。

### void fb\_delete\_videomode（const struct fb\_videomode \*mode，struct list\_head \*head）

从模型列表中删除视频模式条目

**参数**

const struct fb\_videomode \*mode

要删除的视频模式

struct list\_head \*head

modelist的struct list\_head

**注释**

将删除所有匹配的模式条目

### int fb\_find\_mode（struct fb\_var\_screeninfo \*var，struct fb\_info \*info，const char \*mode\_option，const struct fb\_videomode \*db，unsigned int dbsize，const struct fb\_videomode \*default\_mode，unsigned int default\_bpp）

查找有效视频模式

**参数**

struct fb\_var\_screeninfo \*var

帧缓冲用户定义的显示部分

struct fb\_info \*info

帧缓冲信息结构

const char \*mode\_option

要查找的字符串视频模式

const struct fb\_videomode \*db

视频模式数据库

unsigned int dbsize

db的大小

const struct fb\_videomode \*default\_mode

回退的默认视频模式

unsigned int default\_bpp

默认每像素位深度

**说明**

查找合适的视频模式，从mode\_option中指定的模式开始，回退到默认模式。如果默认模式失败，则将尝试所有视频模式数据库中的模式。

mode\_option的有效模式指示符：

<xres> x <yres> [M] [R] [-<bpp>] [@<refresh>][i][p][m]

或者

<name>[-<bpp>][@<refresh>]

其中<xres>，<yres>，<bpp>和<refresh>十进制数字，<name>是字符串。

如果yres之后（如果有refresh / bpp，则为refresh / bpp之前）出现“M”，则函数将使用VESA(tm)协调的视频定时（CVT）计算定时。如果在“M”之后出现“R”，则将使用减少的间隙计算（用于平板电视）。如果存在“i”或“p”，则计算交替或逐行扫描模式。如果有“m”，则添加边距，等于xres的1.8％向下取整至8个像素和yres的1.8％。字符“i”，“p”和“m”必须在“M”和“R”之后。例如：

1024x768MR-8@60m-R带有60Hz的减少空白和边距。

返回零表示失败，返回1表示使用指定的mode\_option，返回2表示使用忽略的刷新速率指定的mode\_option，返回3表示使用默认模式，返回4表示回退到任何有效模式。

**注意**

所传递的结构变量未清除！这允许您提供例如灰度和accel\_flags字段的值。

### void fb\_var\_to\_videomode（struct fb\_videomode \*mode，const struct fb\_var\_screeninfo \*var）

将fb\_var\_screeninfo转换为fb\_videomode

**参数**

struct fb\_videomode \*mode

指向struct fb\_videomode的指针.

const struct fb\_var\_screeninfo \*var

指向struct fb\_var\_screeninfo的指针

### void fb\_videomode\_to\_var(struct fb\_var\_screeninfo \*var, const struct fb\_videomode \*mode)

将fb\_videomode转换为fb\_var\_screeninfo

**参数**

struct fb\_var\_screeninfo \*var

指向struct fb\_var\_screeninfo的指针

const struct fb\_videomode \*mode

指向struct fb\_videomode的指针

### int fb\_mode\_is\_equal(const struct fb\_videomode \*mode1, const struct fb\_videomode \*mode2)

比较 2 种视频模式

**参数**

const struct fb\_videomode \* mode1

第一视频模式

const struct fb\_videomode \* mode2

第二视频模式

**返回**

如果相等则为 1，如果不相等则为 0

### const struct fb\_videomode \*fb\_find\_best\_mode(const struct fb\_var\_screeninfo \*var, struct list\_head \*head)

找到最匹配的视频模式

**参数**

const struct fb\_var\_screeninfo \* var

指向结构 fb\_var\_screeninfo 的指针

struct list\_head \* head

指向 modelist 的 struct list\_head 的指针

**返回**

struct fb\_videomode，如果没有找到则为 NULL

重要提示：此函数假定 info->modelist 中的所有 modelist 条目都是有效的。

**注意**

找到与 var->xres 和 var->yres 具有相同或更大维度的最佳匹配视频模式。如果找到超过 1 个视频模式，将返回刷新率最高的视频模式

### const struct fb\_videomode \*fb\_find\_nearest\_mode(const struct fb\_videomode \*mode, struct list\_head \*head)

找到最近的视频模式

**参数**

const struct fb\_videomode \* mode

指向结构 fb\_videomode 的指针

struct list\_head \* head

指向模型师的指针

**说明**

找到最匹配的视频模式，尺寸更小或更大。如果找到超过 1 个视频模式，将返回具有最接近刷新率的视频模式。

### const struct fb\_videomode \*fb\_match\_mode(const struct fb\_var\_screeninfo \*var, struct list\_head \*head)

找到一个与 var 中的时间完全匹配的视频模式

**参数**

const struct fb\_var\_screeninfo \* var

指向结构 fb\_var\_screeninfo 的指针

struct list\_head \* head

指向 modelist 的 struct list\_head 的指针

**返回**

struct fb\_videomode，如果没有找到则为 NULL

### int fb\_add\_videomode(const struct fb\_videomode \*mode, struct list\_head \*head)

向模型师添加视频模式条目

**参数**

const struct fb\_videomode \* mode

要添加的视频模式

struct list\_head \* head

modelist 的 struct list\_head

**注意**

只会添加不匹配的模式条目

### void fb\_destroy\_modelist(struct list\_head \*head)

摧毁模型师

**参数**

struct list\_head \* head

modelist 的 struct list\_head

### void fb\_videomode\_to\_modelist(const struct fb\_videomode \*modedb, int num, struct list\_head \*head)

将模式数组转换为模式列表

**参数**

const struct fb\_videomode \* modedb

fb\_videomode 结构数组

int num

数组中的条目数

struct list\_head \* head

modelist 的 struct list\_head

## 帧缓冲区Macintosh视频模式数据库

### int mac\_vmode\_to\_var(int vmode, int cmode, struct fb\_var\_screeninfo \*var)

将 vmode/cmode 对转换为 var 结构

**参数**

int vmode

MacOS 视频模式

int cmode

MacOS 颜色模式

struct fb\_var\_screeninfo \* var

帧缓冲区视频模式结构

**说明**

将 MacOS vmode/cmode 对转换为帧缓冲区视频模式结构。

错误时返回负 errno，成功时返回零。

### int mac\_map\_monitor\_sense(int sense)

将监视器感知转换为 vmode

**参数**

int sense

Macintosh 显示器感应数

**描述**

将 Macintosh 监视器感知编号转换为 MacOS vmode 编号。

返回 MacOS vmode 视频模式编号

### int mac\_find\_mode(struct fb\_var\_screeninfo \*var, struct fb\_info \*info, const char \*mode\_option, unsigned int default\_bpp)

找到一个视频模式

**参数**

struct fb\_var\_screeninfo \* var

帧缓冲区用户定义的显示部分

struct fb\_info \* info

帧缓冲区信息结构

const char \* mode\_option

视频模式名称（参见 mac\_modedb[]）

unsigned int default\_bpp

默认颜色深度（以每像素位数为单位）

**说明**

找到合适的视频模式。尝试设置mode\_option指定的模式。如果所需模式的名称以“mac”开头，将使用 Mac 视频模式数据库，否则将回退到标准视频模式数据库。

**注意**

标记为 \_\_init 的函数只能在期间使用：

系统启动。

从 fb\_find\_mode 返回错误代码（参见 fb\_find\_mode 函数）。

## 帧缓冲区字体

有关详细信息，请参阅文件 lib/fonts/fonts.c。

# 电压和电流调节器API

## 介绍

此框架旨在提供标准内核接口，以控制电压和电流调节器。

其目的是允许系统动态地控制调节器功率输出，以节省电力并延长电池寿命。这适用于电压调节器（其输出电压可控制）和电流汇（其电流限制可控制）。

请注意，Linux内核源代码中有其他（目前更完整的）文档可供参考，位于Documentation/power/regulator下。

### 词汇表

调节器API使用许多可能不熟悉的术语

调节器

向其他设备提供电能的电子设备。大多数调节器可以启用和禁用其输出，有些还可以控制其输出电压或电流。

使用者

电子设备，消耗由调节器提供的电能。这些可能是静态的，只需要固定的电源，或者是动态的，需要在运行时对调节器进行积极管理。

电力领域

由给定调节器供电的电子电路，包括调节器和所有使用者设备。调节器的配置在电路中的所有组件之间共享。

电力管理集成电路（PMIC）

一个集成了许多稳压器和其他子系统的IC。在嵌入式系统中，主要的PMIC通常相当于台式机系统中的电源和南桥的组合。

## 消费驱动接口

这提供了类似于内核时钟框架的API。消费者驱动程序使用获取和放置操作来获取和释放稳压器。提供了函数来启用和禁用稳压器，并获取和设置稳压器的运行时参数。

当消费者请求稳压器时，他们使用其供应名称的符号名称，例如“Vcc”，这些符号名称由机器接口映射到实际的稳压器设备。

当稳压器框架未使用时，提供此API的存根版本，以最小化使用ifdefs的需要。

### 启用和禁用

稳压器API提供稳压器的引用计数启用和禁用。消费设备使用regulator\_enable（）和regulator\_disable（）函数来启用和禁用稳压器。对这两个函数的调用必须平衡。

请注意，由于多个消费者可能正在使用稳压器，而机器约束可能不允许禁用稳压器，因此不能保证调用regulator\_disable（）实际上会导致由稳压器提供的电源被禁用。消费者驱动程序应假定稳压器可能始终处于启用状态。

### 配置

某些消费者设备可能需要能够动态配置其供应。例如，MMC驱动程序可能需要为其卡选择正确的工作电压。这可以在启用或禁用稳压器时完成。

regulator\_set\_voltage（）和regulator\_set\_current\_limit（）函数提供了主要接口。两者都采用电压和电流范围，支持不需要特定值的驱动程序（例如，CPU频率缩放通常允许CPU在低频率下使用更宽的供应电压范围，但不需要降低供应电压）。必须相同。

### 回调

还可以为诸如规管故障之类的事件注册回调。

## 稳压器驱动程序接口

稳压器芯片的驱动程序通过向稳压器核心注册稳压器，提供操作结构给核心。通知器接口允许将错误条件报告给核心。

注册应由平台的显式设置触发，为稳压器提供包含约束和供应信息的struct regulator\_init\_data。

## 机器接口

该接口提供了一种定义在给定系统上如何将稳压器连接到消费者以及系统的有效操作参数的方法。

### 供应品

稳压器供应品使用struct regulator\_consumer\_supply指定。这是在驱动程序注册时作为机器约束的一部分完成的。

### 约束

除了定义连接之外，机器接口还提供了定义客户端允许执行的操作和可以设置的参数的约束。这是必需的，因为通常稳压器设备将提供比在给定系统上安全使用的灵活性更高的支持，例如支持更高的供应电压，而不适用于消费者的额定电压等级。

这在驱动程序注册时通过提供struct regulation\_constraints来完成。

约束也可以在约束中指定稳压器的初始配置，这对于静态消费者特别有用。

## API参考

由于内核文档框架和现有源代码布局的限制，在此处记录了整个稳压器API。

### struct pre\_voltage\_change\_data

与PRE\_VOLTAGE\_CHANGE事件一起发送的数据

**定义**

struct pre\_voltage\_change\_data {

unsigned long old\_uV;

unsigned long min\_uV;

unsigned long max\_uV;

};

**成员**

old\_uV

更改之前的电压。

min\_uV

我们将更改为的最小电压。

max\_uV

我们将更改为的最大电压。

### struct regulator\_bulk\_data

用于批量稳压器操作的数据。

**定义**

struct regulator\_bulk\_data {

const char \*supply;

struct regulator \*consumer;

};

**成员**

supply

供应的名称。在使用批量稳压器API之前由用户初始化。

consumer

供应品的稳压器消费者。这将由批量API管理。

**说明**

稳压器API提供了一系列regulator\_bulk\_（）API调用，作为需要多个供应的消费者的便利。此结构用于管理这些调用的数据。

### struct regulator\_state

低功耗系统状态下的稳压器状态

**定义**

struct regulator\_state {

int uV;

int min\_uV;

int max\_uV;

unsigned int mode;

int enabled;

bool changeable;

};

**成员**

uV

挂起期间的默认操作电压，可以在<min\_uV，max\_uV>之间进行调整。

min\_uV

可以设置最小挂起电压。

max\_uV

可以设置最大挂起电压。

mode

挂起期间的操作模式。

enabled

挂起期间的操作。 - DO\_NOTHING\_IN\_SUSPEND - DISABLE\_IN\_SUSPEND - ENABLE\_IN\_SUSPEND

changeable

此状态可以在启用/禁用之间切换，

**说明**

这说明了在系统宽低功率状态下调节器的状态。必须设置启用或禁用之一才能应用配置。

### struct regulation\_constraints

调节器操作约束。

**定义**

struct regulation\_constraints {

const char \*name;

int min\_uV;

int max\_uV;

int uV\_offset;

int min\_uA;

int max\_uA;

int ilim\_uA;

int system\_load;

u32 \*max\_spread;

int max\_uV\_step;

unsigned int valid\_modes\_mask;

unsigned int valid\_ops\_mask;

int input\_uV;

struct regulator\_state state\_disk;

struct regulator\_state state\_mem;

struct regulator\_state state\_standby;

struct notification\_limit over\_curr\_limits;

struct notification\_limit over\_voltage\_limits;

struct notification\_limit under\_voltage\_limits;

struct notification\_limit temp\_limits;

suspend\_state\_t initial\_state;

unsigned int initial\_mode;

unsigned int ramp\_delay;

unsigned int settling\_time;

unsigned int settling\_time\_up;

unsigned int settling\_time\_down;

unsigned int enable\_time;

unsigned int active\_discharge;

unsigned always\_on:1;

unsigned boot\_on:1;

unsigned apply\_uV:1;

unsigned ramp\_disable:1;

unsigned soft\_start:1;

unsigned pull\_down:1;

unsigned over\_current\_protection:1;

unsigned over\_current\_detection:1;

unsigned over\_voltage\_detection:1;

unsigned under\_voltage\_detection:1;

unsigned over\_temp\_detection:1;

};

**成员**

name

约束说明名称，用于显示目的。

min\_uV

使用者可以设置的最小电压。

max\_uV

使用者可以设置的最大电压。

uV\_offset

从使用者到调节器的电压的偏移量。

min\_uA

使用者可以设置的最小电流。

max\_uA

使用者可以设置的最大电流。

ilim\_uA

最大输入电流。

system\_load

未被任何使用者请求捕获的负载。

max\_spread

耦合调节器之间的最大可能差距

max\_uV\_step

电压最大可能的步进变化

valid\_modes\_mask

使用者可以配置的模式掩码。

valid\_ops\_mask

使用者可以执行的操作。

input\_uV

由另一个调节器提供电源时的调节器输入电压。

state\_disk

系统以磁盘模式挂起时调节器的状态。

state\_mem

系统以存储器模式挂起时调节器的状态。

state\_standby

系统以待机模式挂起时调节器的状态。

over\_curr\_limits

超过电流限制的限制。

over\_voltage\_limits

超过电压限制的限制。

under\_voltage\_limits

在电压限制下的限制。

temp\_limits

温度限制的限制。

initial\_state

默认设置的挂起状态。

initial\_mode

在启动时设置的模式。

ramp\_delay

电压变化后稳定下来的时间（单位uV / us）

settling\_time

电压变化时稳定下来的时间（当电压变化是非线性时）（单位微秒）。

settling\_time\_up

电压上升后稳定下来的时间（当电压变化是非线性时）（单位微秒）。

settling\_time\_down

电压降低后稳定下来的时间（当电压变化是非线性时）（单位微秒）。

enable\_time

线路开启时间（单位微秒）

active\_discharge

启用/禁用主动放电。enum调节器活动放电值用于初始化。

always\_on

如果调节器永远不应被禁用，则设置。

boot\_on

如果系统最初启动时启用调节器，则设置。如果调节器未被硬件或引导加载程序启用，则在应用约束时将启用它。

apply\_uV

在初始化时应用电压约束。

ramp\_disable

在初始化或设置电压时禁用斜坡延迟。

soft\_start

启用软起动以使电压缓慢上升。

pull\_down

当调节器被禁用时启用下拉。

over\_current\_protection

在超过电流事件时自动禁用。

over\_current\_detection

配置超过电流限制。

over\_voltage\_detection

配置过高电压限制。

under\_voltage\_detection

配置低于电压限制。

over\_temp\_detection

配置超过温度限制。

**说明**

此结构说明调节器和板/机器特定的约束。

### struct regulation\_constraints

供应- >设备映射

**定义**

struct regulator\_consumer\_supply {

const char \*dev\_name;

const char \*supply;

};

**成员**

dev\_name

使用者的dev\_name（）结果。

supply

供应名称。

**说明**

这将一个供应名称映射到一个设备。使用dev\_name允许支持晚回收struct device的总线，如I2C。

### struct regulator\_init\_data

调节器平台初始化数据。

**定义**

struct regulator\_init\_data {

const char \*supply\_regulator;

struct regulation\_constraints constraints;

int num\_consumer\_supplies;

struct regulator\_consumer\_supply \*consumer\_supplies;

int (\*regulator\_init)(void \*driver\_data);

void \*driver\_data;

};

**成员**

supply\_regulator

父调节器。使用调节器在sysfs中的名称字段中显示的调节器名称指定，该名称可以使用约束字段“名称”显式设置。

constraints

限制。必须指定这些约束才能使用调节器。

num\_consumer\_supplies

使用者设备供应的数量。

consumer\_supplies

使用者设备供应配置。

regulator\_init

调节器注册后调用的回调。

driver\_data

传递给regulator\_init的数据。

**说明**

初始化约束，我们的供应和使用者供应。

### struct regulator\_linear\_range

指定线性电压范围

**定义**

struct regulator\_linear\_range {

unsigned int min\_uV;

unsigned int min\_sel;

unsigned int max\_sel;

unsigned int uV\_step;

};

**成员**

min\_uV

范围中的最低电压

min\_sel

范围的最低选择器

max\_sel

范围的最高选择器

uV\_step

步长

**说明**

为regulator\_map\_linear\_range（）和regulator\_list\_linear\_range（）指定一个电压范围。

### struct regulator\_ops

调节器操作。

**定义**

struct regulator\_ops {

int (\*list\_voltage) (struct regulator\_dev \*, unsigned selector);

int (\*set\_voltage) (struct regulator\_dev \*, int min\_uV, int max\_uV, unsigned \*selector);

int (\*map\_voltage)(struct regulator\_dev \*, int min\_uV, int max\_uV);

int (\*set\_voltage\_sel) (struct regulator\_dev \*, unsigned selector);

int (\*get\_voltage) (struct regulator\_dev \*);

int (\*get\_voltage\_sel) (struct regulator\_dev \*);

int (\*set\_current\_limit) (struct regulator\_dev \*, int min\_uA, int max\_uA);

int (\*get\_current\_limit) (struct regulator\_dev \*);

int (\*set\_input\_current\_limit) (struct regulator\_dev \*, int lim\_uA);

int (\*set\_over\_current\_protection)(struct regulator\_dev \*, int lim\_uA, int severity, bool enable);

int (\*set\_active\_discharge)(struct regulator\_dev \*, bool enable);

int (\*enable) (struct regulator\_dev \*);

int (\*disable) (struct regulator\_dev \*);

int (\*is\_enabled) (struct regulator\_dev \*);

int (\*set\_mode) (struct regulator\_dev \*, unsigned int mode);

unsigned int (\*get\_mode) (struct regulator\_dev \*);

int (\*get\_error\_flags)(struct regulator\_dev \*, unsigned int \*flags);

int (\*enable\_time) (struct regulator\_dev \*);

int (\*set\_ramp\_delay) (struct regulator\_dev \*, int ramp\_delay);

int (\*set\_voltage\_time) (struct regulator\_dev \*, int old\_uV, int new\_uV);

int (\*set\_voltage\_time\_sel) (struct regulator\_dev \*,unsigned int old\_selector, unsigned int new\_selector);

int (\*set\_soft\_start) (struct regulator\_dev \*);

int (\*get\_status)(struct regulator\_dev \*);

unsigned int (\*get\_optimum\_mode) (struct regulator\_dev \*, int input\_uV, int output\_uV, int load\_uA);

int (\*set\_load)(struct regulator\_dev \*, int load\_uA);

int (\*set\_bypass)(struct regulator\_dev \*dev, bool enable);

int (\*get\_bypass)(struct regulator\_dev \*dev, bool \*enable);

int (\*set\_suspend\_voltage) (struct regulator\_dev \*, int uV);

int (\*set\_suspend\_enable) (struct regulator\_dev \*);

int (\*set\_suspend\_disable) (struct regulator\_dev \*);

int (\*set\_suspend\_mode) (struct regulator\_dev \*, unsigned int mode);

int (\*resume)(struct regulator\_dev \*rdev);

int (\*set\_pull\_down) (struct regulator\_dev \*);

};

**成员**

list\_voltage

返回受支持的电压之一，以微伏为单位； 如果选择器指示的电压在此系统上无法使用，则为零； 或者负errno。选择器的范围从零到少于regulator\_desc .n\_voltages。电压可以以任何顺序报告。

set\_voltage

设置规定范围内的调整器电压。 驱动程序应选择最接近min\_uV的电压。

map\_voltage

将电压转换为选择器

set\_voltage\_sel

使用指定的选择器设置调整器电压。

get\_voltage

返回调整器的当前配置的电压； 如果调节器无法在启动时读取且尚未设置，则返回-ENOTRECOVERABLE。

get\_voltage\_sel

返回调整器的当前配置电压选择器； 如果调节器无法在启动时读取且尚未设置，则返回-ENOTRECOVERABLE。

set\_current\_limit

为限流调节器配置限制。 驱动程序应选择最接近max\_uA的电流。

get\_current\_limit

获取限流调节器的配置限制。

set\_input\_current\_limit

配置输入限制。

set\_over\_current\_protection

支持启用和设置超电流情况检测的限制。

set\_active\_discharge

设置调节器的主动放电启用/禁用。

enable

将调节器配置为已启用。

disable

将调节器配置为已禁用。

is\_enabled

如果调节器已启用，则返回1，否则返回0。 还可以返回负errno。

set\_mode

设置调节器的配置操作模式。

get\_mode

获取调节器的配置操作模式。

get\_error\_flags

获取调节器的当前错误。

enable\_time

启用后调节器输出电压稳定所需的时间，单位为微秒。

set\_ramp\_delay

设置调节器的渐变延迟。 驱动程序应选择小于（最接近的）ramp\_delay的渐变延迟。

set\_voltage\_time

将调整器电压输出电压设置为新值后必须稳定的时间，单位为μs。 函数接收from和to电压作为输入，应返回最坏情况。

set\_voltage\_time\_sel

将调整器电压输出电压设置为新值后必须稳定的时间，单位为μs。 函数接收from和to电压选择器作为输入，应返回最坏情况。

set\_soft\_start

为调整器启用软启动。

get\_status

以REGULATOR\_STATUS值（或负errno）返回调整器的实际（非配置）状态

get\_optimum\_mode

获取调整器在指定参数下运行时最有效的操作模式。

set\_load

设置调节器的负载。

set\_bypass

设置调节器为旁路模式。

get\_bypass

获取调节器旁路模式状态。

set\_suspend\_voltage

在系统暂停时设置调节器的电压。

set\_suspend\_enable

当系统挂起时，将调节器标记为启用。

set\_suspend\_disable

当系统挂起时，将调节器标记为禁用。

set\_suspend\_mode

设置系统挂起时调节器的运行模式。

set\_pull\_down

配置调节器在禁用时下拉。

**说明**

此结构说明了调节器芯片驱动程序可以实现的调节器操作。

### struct regulator\_desc

struct调节器说明符

**定义**

struct regulator\_desc {

const char \*name;

const char \*supply\_name;

const char \*of\_match;

const char \*regulators\_node;

int (\*of\_parse\_cb)(struct device\_node \*,const struct regulator\_desc \*, struct regulator\_config \*);

int id;

unsigned int continuous\_voltage\_range:1;

unsigned n\_voltages;

const struct regulator\_ops \*ops;

int irq;

enum regulator\_type type;

struct module \*owner;

unsigned int min\_uV;

unsigned int uV\_step;

unsigned int linear\_min\_sel;

int fixed\_uV;

unsigned int ramp\_delay;

int min\_dropout\_uV;

const struct regulator\_linear\_range \*linear\_ranges;

int n\_linear\_ranges;

const unsigned int \*volt\_table;

unsigned int vsel\_reg;

unsigned int vsel\_mask;

unsigned int csel\_reg;

unsigned int csel\_mask;

unsigned int apply\_reg;

unsigned int apply\_bit;

unsigned int enable\_reg;

unsigned int enable\_mask;

unsigned int enable\_val;

unsigned int disable\_val;

bool enable\_is\_inverted;

unsigned int bypass\_reg;

unsigned int bypass\_mask;

unsigned int bypass\_val\_on;

unsigned int bypass\_val\_off;

unsigned int active\_discharge\_on;

unsigned int active\_discharge\_off;

unsigned int active\_discharge\_mask;

unsigned int active\_discharge\_reg;

unsigned int soft\_start\_reg;

unsigned int soft\_start\_mask;

unsigned int soft\_start\_val\_on;

unsigned int pull\_down\_reg;

unsigned int pull\_down\_mask;

unsigned int pull\_down\_val\_on;

unsigned int enable\_time;

unsigned int off\_on\_delay;

unsigned int (\*of\_map\_mode)(unsigned int mode);

};

**成员**

name

标识调节器的名称。

supply\_name

标识调节器的电源。

of\_match

用于在DT中标识调节器的名称。

regulators\_node

在DT中包含调节器定义的节点名称。

of\_parse\_cb

如果存在 of\_match，则可选回调。在 init\_data 解析期间为 DT 中解析的每个调节器调用。传递给回调函数的 regulator\_config 参数将是仅对本特定调用有效的 regulator\_register 传递的配置的副本。回调函数可以自由更改配置，但不能将其存储用于以后的使用。回调应返回 0 表示成功，或返回负 ERRNO 表示失败。

id

调节器的数字标识符。

continuous\_voltage\_range

指示调节器是否可以在限制范围内设置任何电压。

n\_voltages

可用于 ops.list\_voltage() 的选择器数量。

ops

调节器操作表。

irq

调节器的中断号。

type

指示调节器是电压调节器或电流调节器。

owner

提供调节器的模块，用于引用计数。

min\_uV

最低选择器所带的电压（如果是线性映射）。

uV\_step

每个选择器所增加的电压（如果是线性映射）。

linear\_min\_sel

开始线性映射的最小选择器。

fixed\_uV

固定轨道的电压。

ramp\_delay

电压变化后稳定的时间（单位uV/us）。

min\_dropout\_uV

此调节器可以处理的最小掉电电压。

linear\_ranges

可能的电压范围的常量表。

n\_linear\_ranges

线性范围（如果使用）表中的条目数。

volt\_table

电压映射表（如果基于表格的映射）。

vsel\_reg

使用 regulator\_map\_\*\_voltage\_\* 时的选择器寄存器。

vsel\_mask

用于选择器的寄存器位字段的掩码。

csel\_reg

使用 regmap set\_current\_limit 的当前限制选择器的寄存器。

csel\_mask

用于当前限制选择器的寄存器位字段的掩码。

apply\_reg

当使用 regulator\_set\_voltage\_sel\_regmap 时，用于在输出上启动电压变化的寄存器。

apply\_bit

使用 regulator\_set\_voltage\_sel\_regmap 时用于在输出上启动电压变化的寄存器位字段。

enable\_reg

控制 regmap enable/disable ops 时使用的寄存器。

enable\_mask

用于控制 regmap enable/disable ops 的寄存器位字段。

enable\_val

控制 regmap enable/disable ops 时的启用值。

disable\_val

控制 regmap enable/disable ops 时的禁用值。

enable\_is\_inverted

一个标志，用于指示在使用 regulator\_enable\_regmap 和 friends APIs 时将 enable\_mask 位设置为禁用。

bypass\_reg

使用 regmap set\_bypass 时使用的控制寄存器。

bypass\_mask

用于使用 regmap set\_bypass 时的控制寄存器位字段的掩码。

bypass\_val\_on

使用 regmap set\_bypass 时的启用值。

bypass\_val\_off

使用 regmap set\_bypass 时的禁用值。

bypass\_val\_off

使用regmap set\_bypass时的禁用控制值

active\_discharge\_on

使用regmap set\_active\_discharge时的禁用控制值

active\_discharge\_off

使用regmap set\_active\_discharge时的启用控制值

active\_discharge\_mask

使用regmap set\_active\_discharge时的掩码控制值

active\_discharge\_reg

使用regmap set\_active\_discharge时的寄存器控制值

soft\_start\_reg

使用regmap set\_soft\_start时的寄存器控制值

soft\_start\_mask

使用regmap set\_soft\_start时的掩码控制值

soft\_start\_val\_on

使用regmap set\_soft\_start时的启用控制值

pull\_down\_reg

使用regmap set\_pull\_down时的寄存器控制值

pull\_down\_mask

使用regmap set\_pull\_down时的掩码控制值

pull\_down\_val\_on

使用regmap set\_pull\_down时的启用控制值

enable\_time

调节器初始启用需要的时间（以微秒为单位）。

off\_on\_delay

在重新启用调节器之前的保护时间（以微秒为单位）

poll\_enabled\_time

在检查调节器实际启用时使用的轮询间隔（以微秒为单位）。最多可达enable\_time。

of\_map\_mode

将DeviceTree中定义的硬件模式映射到标准模式

**说明**

每个注册到核心的调节器均由此类型的结构和regulator\_config结构来说明。该结构包含调节器说明的非变量部分。

### struct regulator\_config

动态调节器说明符

**定义**

struct regulator\_config {

struct device \*dev;

const struct regulator\_init\_data \*init\_data;

void \*driver\_data;

struct device\_node \*of\_node;

struct regmap \*regmap;

bool ena\_gpio\_initialized;

int ena\_gpio;

struct gpio\_desc \*ena\_gpiod;

unsigned int ena\_gpio\_invert:1;

unsigned int ena\_gpio\_flags;

};

**成员**

dev

监管机构的结构设备

init\_data

平台提供初始化数据，由驱动传递

driver\_data

私人监管机构数据

of\_node

用于解析设备树绑定的 OpenFirmware 节点（可能为 NULL）。

regmap

如果不足，则用于核心 regmap 助手的 regmap dev\_get\_regmap()。

ena\_gpio\_initialized

GPIO 控制调节器启用已正确初始化，这意味着 >= 0 是有效的 gpio 标识符，< 0 是不存在的 gpio。

ena\_gpio

GPIO 控制稳压器使能。

ena\_gpiod

GPIO 描述符控制调节器使能。

ena\_gpio\_invert

感知 GPIO 使能控制。

ena\_gpio\_flags

调用时使用的标志[gpio\_request\_one()](https://www.kernel.org/doc/html/v4.19/driver-api/gpio/legacy.html" \l "c.gpio_request_one" \o "gpio_request_one)

**说明**

每个注册到核心的调节器均由此类型的结构和regulator\_desc结构来说明。该结构包含调节器说明的运行时变量部分。

### struct regulator \*regulator\_get(struct device \*dev, const char \*id)

查找并获得对调节器的引用。

**参数**

struct device \*dev

调节器“使用者”所在的设备

const char \*id

提供名称或调节器ID。

**说明**

返回对应于调节器生产方的struct regulator，或包含errno的IS\_ERR()条件。

强烈建议使用通过set\_consumer\_device\_supply（）配置的供应名称。建议使用的供应名称应与数据表中所用的名称和/或相关设备引脚的名称相匹配。

### struct regulator \*regulator\_get\_exclusive(struct device \*dev, const char \*id)

获得对调节器的独占访问。

**参数**

struct device \*dev

调节器“使用者”所在的设备

const char \*id

提供名称或调节器ID。

**说明**

返回对应于调节器生产方的struct regulator，或包含errno的IS\_ERR()条件。在持有此引用并且调节器的使用计数已初始化以反映调节器当前状态的情况下，其他使用者将无法获得此调节器。

这适用于无法容忍调节器共享使用的使用者，例如需要强制关闭调节器以正确操作其控制的硬件的使用者。

强烈建议使用通过set\_consumer\_device\_supply（）配置的供应名称。建议使用的供应名称应与数据表中所用的名称和/或相关设备引脚的名称相匹配。

### struct regulator \*regulator\_get\_optional(struct device \*dev, const char \*id)

获得对调节器的可选访问。

**参数**

struct device \*dev

调节器“使用者”所在的设备

const char \*id

提供名称或调节器ID。

**说明**

返回对应于调节器生产方的struct regulator，或包含errno的IS\_ERR()条件。

这适用于可以在正常使用中断开某些供电的设备的使用者，例如某些MMC设备。它可以允许调节器核心通过使用正常的regulator\_get（）调用来提供其他供电的存根提供，而无需干扰可以处理缺失供电的驱动程序的操作。

强烈建议使用通过set\_consumer\_device\_supply（）配置的供应名称。建议使用的供应名称应与数据表中所用的名称和/或相关设备引脚的名称相匹配。

### void regulator\_put（struct regulator \*调整器）

“释放”调节器源

**参数**

struct regulator \*regulator

调节器源

**注意事项**

驱动程序必须确保在调用此函数之前对此调节器源进行的所有regulator\_enable调用都应以regulator\_disable调用平衡。

### int regulator\_register\_supply\_alias(struct device \*dev, const char \*id, struct device \*alias\_dev, const char \*alias\_id)

为供电线路提供设备别名查找

**参数**

struct device \*dev

将作为调节器“使用者”提供给该设备的设备

const char \*id

提供名称或调节器ID

struct device \*alias\_dev

应该使用的设备来查找供应

const char \*alias\_id

应该用于查找供应的供应名称或稳压器ID

**说明**

在alias\_dev上查找id的所有查找都将改为在alias\_id上替代的dev。

### void regulator\_unregister\_supply\_alias(struct device \*dev，const char \*id)

删除设备别名

参量

struct device \*dev

将作为稳压器“消费者”赠送的设备

const char \*id

供应名称或稳压器ID

**说明**

如果id在dev上存在查找别名，则删除该别名。

### int regulator\_bulk\_register\_supply\_alias（struct device \*dev，const char \*const \*id，struct device \*alias\_dev，const char \*const \*alias\_id，int num\_id）

注册多个别名

**参数**

struct device \*dev

作为稳压器“消费者”赠送的设备

const char \*const \*id

供应名称列表或将作为稳压器ID的设备

struct device \*alias\_dev

应该用于查找供应的设备

const char \*const \*alias\_id

应该用于查找供应的供应或稳压器ID的名称列表

int num\_id

要注册的别名数

**说明**

成功返回0，失败时返回errno。

此辅助函数允许驱动程序在一个操作中注册多个供应别名。如果任何别名无法注册，则在返回给调用者之前将删除所有已注册的别名。

### void regulator\_bulk\_unregister\_supply\_alias（struct device \*dev，const char \*const \*id，int num\_id）

取消注册多个别名

**参数**

struct device \*dev

将作为稳压器“消费者”赠送的设备

const char \*const \*id

供应名称或稳压器ID列表

int num\_id

要取消注册的别名数

**说明**

此辅助函数允许驱动程序在一个操作中注销多个供应别名。

### int regulator\_enable（struct regulator \* regulator）

启用稳压器输出

**参数**

struct regulator \* regulator

调节器需要的电源

**说明**

请求使用预定义的电压或电流值启用稳压器输出。 regulator\_enable（）的调用必须与regulator\_disable（）的调用平衡。

**注意**

输出值可以由其他驱动程序，引导加载程序或可以在稳压器中硬连线来设置。

### int regulator\_disable（struct regulator \* regulator）

禁止稳压器输出

**参数**

struct regulator \* regulator

调节器需要的电源

**说明**

禁用稳压器输出电压或电流。 regulator\_enable（）的调用必须与regulator\_disable（）的调用平衡。

**注意**

只有当没有其他使用者设备启用时，此操作才会关闭稳压器输出，稳压器设备支持禁用并且机器约束允许执行此操作。

### int regulator\_force\_disable（struct regulator \* regulator）

强制禁用稳压器输出

**参数**

struct regulator \* regulator

调节器需要的电源

**说明**

强制禁用稳压器输出电压或电流。

**注意**

即使其他使用者设备启用它，这也会禁用稳压器输出。 如果不禁用稳压器，则可能会导致设备损坏的情况下应使用此功能（例如，超温）。

### int regulator\_disable\_deferred（struct regulator \* regulator，int ms）

延迟禁用稳压器输出

**参数**

struct regulator \* regulator

调节器需要的电源

int ms

直到稳压器被禁用的毫秒数

**说明**

在延迟之后执行regulator\_disable（）对稳压器的操作。 这适用于需要一些时间来静音的设备。

**注意**

只有当没有其他使用者设备启用时，此操作才会禁用稳压器输出，稳压器设备支持禁用并且机器约束允许执行此操作。

### int regulator\_is\_enabled（struct regulator \* regulator）

稳压器输出是否已启用

**参数**

struct regulator \* regulator

调节器需要的电源

**说明**

如果支持客户端的稳压器驱动程序已请求启用设备，则返回正数，否则返回零，否则返回负的errno代码。

请注意，支持此稳压器手柄的设备可以有多个用户，因此即使从未为此特定源调用regulator\_enable（），它也可能已启用。

### int regulator\_count\_voltages（struct regulator \* regulator）

计算稳压器列表电压（）选择器

**参数**

struct regulator \* regulator

调节器需要的电源

**说明**

返回选择器的数量或负errno。 选择器从零开始编号，通常对应于硬件寄存器中的位字段。

### int regulator\_list\_voltage（struct regulator \* regulator，unsigned selector）

枚举支持的电压

**参数**

struct regulator \* regulator

调节器需要的电源

unsigned selector

识别要列出的电压

**上下文**

可以睡觉

**说明**

返回电压，该电压可以传递给regulator\_set\_voltage（），如果此选择器代码无法在此系统上使用，则返回零，否则返回负errno。

### int regulator\_get\_hardware\_vsel\_register（struct regulator \* regulator，unsigned \* vsel\_reg，unsigned \* vsel\_mask）

获取硬件电压选择寄存器

**参数**

struct regulator \* regulator

调节器需要的电源

unsigned \* vsel\_reg

电压选择器寄存器，输出参数

unsigned \* vsel\_mask

电压选择器位域的掩码，输出参数

**说明**

返回用于设置稳压器电压的硬件寄存器偏移量和位掩码。 例如，在配置电压缩放硬件或固件可以使I2C请求绕过内核时，这可能很有用。

成功时，将填入输出参数vsel\_reg和vsel\_mask，并返回0，否则返回负errno。

### int regulator\_list\_hardware\_vsel（struct regulator \*regulator，unsigned selector）

获取选择器的硬件特定寄存器值

**参数**

struct regulator \*regulator

调节器来源

unsigned selector

电压标识

**说明**

将选择器转换为硬件特定的电压选择器，可直接写入调节器寄存器。可以通过调用regulator\_get\_hardware\_vsel\_register来确定电压寄存器的地址。

发生错误时返回负的errno。

### unsigned int regulator\_get\_linear\_step（struct regulator \*regulator）

返回VSEL值之间的电压步长大小

**参数**

struct regulator \*regulator

调节器来源

**说明**

返回线性调节器的VSEL值之间的电压步长大小，如果调节器不是线性调节器，则返回0。

### int regulator\_is\_supported\_voltage（struct regulator \*regulator，int min\_uV，int max\_uV）

检查是否可以支持电压范围

**参数**

struct regulator \*regulator

要检查的调节器。

int min\_uV

所需最小电压（微伏）。

int max\_uV

所需最大电压（微伏）。

**说明**

返回布尔值或负错误代码。

### int regulator\_set\_voltage（struct regulator \*regulator，int min\_uV，int max\_uV）

设置调节器输出电压

**参数**

struct regulator \*regulator

调节器来源

int min\_uV

所需最小电压（微伏）

int max\_uV

可接受的最高电压（微伏）

**说明**

将电压调节器设置为所需的输出电压。这可以在任何调节器状态下设置。也就是说，调节器可以被禁用或启用。

如果调节器已启用，则电压将立即改变为新值，否则，如果调节器被禁用，则在启用时调节器将输出新电压。

**注意**

如果调节器在多个设备之间共享，则将使用满足系统限制的最低请求电压。调节器系统限制必须在调用此功能之前设置为该调节器，否则此调用将失败。

### int regulator\_set\_voltage\_time（struct regulator \*regulator，int old\_uV，int new\_uV）

获取提高/降低时间

**参数**

struct regulator \*regulator

调节器来源

int old\_uV

起始电压（毫伏）

int new\_uV

目标电压（毫伏）

**说明**

提供起始和结束电压，此函数尝试计算到达该新电压所需的微秒时间。

### int regulator\_set\_voltage\_time\_sel（struct regulator\_dev \*rdev，unsigned int old\_selector，unsigned int new\_selector）

获取提高/降低时间

**参数**

结构调节器\_dev \* rdev

调节器源设备

unsigned int old\_selector

用于起始电压的选择器

unsigned int new\_selector

用于目标电压的选择器

**说明**

提供起始和目标电压选择器，此函数返回到达此新电压所需的时间（以微秒为单位）

提供ramp\_delay的驱动器可以将其用作set\_voltage\_time\_sel（）操作。

### int regulator\_sync\_voltage（struct regulator \*regulator）

重新应用上次调节器输出电压

**参数**

结构调节器\*调节器

调节器来源

**说明**

重新应用上次配置的电压。这旨在用于某些外部控制源与消费者合作导致配置的电压发生更改的情况。

### int regulator\_get\_voltage（struct regulator \*regulator）

获取调节器输出电压

**参数**

结构调节器\*调节器

调节器来源

**说明**

以微伏为单位返回当前调节器电压。

**注意**

如果调节器已禁用，则返回电压值。此函数不应用于确定调节器状态。

### int regulator\_set\_current\_limit（struct regulator \*regulator，int min\_uA，int max\_uA）

设置调节器输出电流限制

**参数**

struct regulator \*regulator

调节器来源

int min\_uA

支持的最小电流（微安）

int max\_uA

支持的最大电流（微安）

**说明**

将电流汇流处设置为所需的输出电流。这可以在任何调节器状态下设置。也就是说，调节器可以被禁用或启用。

如果调节器已启用，则电流将立即更改为新值，否则，如果调节器被禁用，则在启用时调节器将输出新电流。

**注**

调节器系统限制必须在调用此函数之前设置为该调节器，否则此调用将失败。

### int regulator\_get\_current\_limit（struct regulator \*regulator）

获取调节器输出电流

**参数**

struct regulator \*regulator

调节器来源

**说明**

以微安为单位返回指定电流汇流处提供的电流。

**注**

如果调节器已禁用，则返回电流值。此函数不应用于确定调节器状态。

### int regulator\_set\_mode（struct regulator \*regulator，unsigned int mode）

设置调节器工作模式

**参数**

struct regulator \*regulator

调节器来源

unsigned int mode

操作模式- REGULATOR\_MODE常量之一

**说明**

将调节器工作模式设置为提高调节器效率或提高调节性能。

**注**

调节器系统限制必须在调用此函数之前设置为该调节器，否则此调用将失败。

### unsigned int regulator\_get\_mode（struct regulator \*regulator）

获取调节器工作模式

**参数**

struct regulator \*regulator

调节器来源

**说明**

获取当前的调节器工作模式。

### int regulator\_get\_error\_flags（struct regulator \*regulator，unsigned int \*flags）

获取调节器错误信息

**参数**

struct regulator \*regulator

调节器来源

unsigned int \* flags

存储错误标志的指针

**说明**

获取当前的调节器误差信息。

### int regulator\_set\_load(struct regulator \* regulator，int uA\_load)

设置调节器负载。

**参数**

struct regulator \* regulator

调节器源

int uA\_load

负载电流

**说明**

向调节器内核通知新的设备负载。然后，DRMS（如果由约束启用）使用这个负载来设置新调节器负载的最有效的调节器工作模式。

当消费者设备改变操作状态和电源状态时，它们会通知其供应调节器它们将要使用的最大功率（可以从设备数据表中获取功耗表）。影响电源消耗的操作状态变化的实例包括：

1. 设备打开/关闭。
2. 设备的IO即将开始或已经完成。
3. 设备闲置工作。

此信息也通过sysfs导出给用户空间。

如果平台约束允许，DRMS将总请求数量加总并更改为最有效的操作模式。

出错时，会返回负errno。

### int regulator\_allow\_bypass(struct regulator \* regulator，bool enable)

允许调节器进入旁路模式。

**参数**

struct regulator \* regulator

要配置的调节器

bool enable

启用或禁用旁路模式。

**说明**

如果所有其他调节器使用者也启用旁路模式并且机器约束允许，允许调节器进入旁路模式。旁路模式意味着调节器仅将输入直接传递到输出，而不进行调节。

### int regulator\_register\_notifier(struct regulator \* regulator，struct notifier\_block \* nb)

注册调节器事件通知器。

**参数**

struct regulator \* regulator

调节器源

struct notifier\_block \* nb

通知块

**说明**

注册通知器块以接收调节器事件。

### int regulator\_unregister\_notifier(struct regulator \* regulator，struct notifier\_block \* nb)

注销调节器事件通知器。

**参数**

struct regulator \* regulator

调节器源

struct notifier\_block \* nb

通知块

**说明**

注销调节器事件通知器块。

### int regulator\_bulk\_get(struct device \* dev，int num\_consumers，struct regulator\_bulk\_data \* consumers)

获取多个调节器使用者。

**参数**

struct device \* dev

设备供应

int num\_consumers

要注册的使用者数量

struct regulator\_bulk\_data \* consumers

消费者配置；客户端存储在此处。

**说明**

在失败时返回0，在失败时返回errno。

该帮助程序函数允许驱动程序在一个操作中获取多个调节器使用者。如果无法获取任何调节器，则返回调用者之前分配的所有调节器将被释放。

### int regulator\_**bulk**\_enable(int num\_consumers，struct regulator\_bulk\_data \* consumers)

启用多个调节器使用者。

**参数**

int num\_consumers

消费者数量

struct regulator\_bulk\_data \* consumers

消费数据；客户端存储在此处。成功返回0，失败返回errno

**说明**

此方便API允许使用者在单个API调用中启用多个调节器客户端。如果无法启用任何使用者，则在返回之前任何其他已启用的使用者将被再次禁用。

### int regulator\_bulk\_disable(int num\_consumers，struct regulator\_bulk\_data \* consumers)

禁用多个调节器使用者。

**参数**

int num\_consumers

消费者数量

struct regulator\_bulk\_data \* consumers

消费数据；客户端存储在此处。成功返回0，失败返回errno

**说明**

此方便API允许消费者在单个API调用中禁用多个调节器客户端。如果无法禁用任何使用者，则在返回之前任何其他已禁用的使用者将被再次启用。

### int regulator\_bulk\_force\_disable(int num\_consumers，struct regulator\_bulk\_data \* consumers)

强制禁用多个调节器使用者。

**参数**

int num\_consumers

消费者数量

struct regulator\_bulk\_data \* consumers

消费数据；客户端存储在此处。成功返回0，失败返回errno

**说明**

此方便API允许消费者在单个API调用中强制禁用多个调节器客户端。

注当设备热度高时会造成设备损坏，应使用此选项禁用调节器。虽然调节器\_force\_disable函数调用某些使用者可能会返回错误编号，但会为所有使用者调用该函数。

### void regulator\_bulk\_free(int num\_consumers，struct regulator\_bulk\_data \* consumers)

释放多个调节器使用者。

**参数**

int num\_consumers

消费者数量

struct regulator\_bulk\_data \* consumers

消费数据；客户端存储在此处。

**说明**

此方便API允许消费者在单个API调用中免费多个调节器客户端。

### int regulator\_notifier\_call\_chain(struct regulator\_dev \* rdev，unsigned long event，void \* data)

调用调节器事件通知器。

**参数**

struct regulator\_dev \* rdev

调节器源

unsigned long event

通知块

void \* data

回调特定数据。

**说明**

由调节器驱动程序调用以通知客户端发生了调节器事件。

### int regulator\_mode\_to\_status(unsigned int mode)

将调节器模式转换为状态

**参数**

unsigned int mode

转换模式

**说明**

将一个稳压器模式转换为状态。

### struct regulator\_dev \*regulator\_register(const struct regulator\_desc \*regulator\_desc, const struct regulator\_config \*cfg)

注册稳压器

**参数**

const struct regulator\_desc \*regulator\_desc

要注册的稳压器

const struct regulator\_config \*cfg

用于稳压的运行时配置

**说明**

稳压器驱动程序调用此函数注册一个regu。成功时返回指向struct “regulator\_dev”的有效指针，而在错误的情况下返回ERR\_PTR()。

### void regulator\_unregister(struct regulator\_dev \*rdev)

注销稳压器

**参数**

struct regulator\_dev \*rdev

要注销的稳压器

**说明**

稳压器驱动程序调用此函数注销一个稳压器。

### void regulator\_has\_full\_constraints(void)

系统有完全规范化的约束条件

**参数**

无参数

**说明**

调用此函数将导致稳压器API禁用所有计数为0且没有始终启用约束条件的稳压器，该函数为late\_initcall。意图是这将成为未来内核发布中的默认行为，因此鼓励用户立即使用此功能。

### void \*rdev\_get\_drvdata(struct regulator\_dev \*rdev)

获取rdev稳压器驱动程序数据

**参数**

struct regulator\_dev \*rdev

稳压器

**说明**

获取rdev稳压器驱动程序的私有数据。可以在稳压器驱动程序上下文中使用此调用。

### void \*regulator\_get\_drvdata(struct regulator \*regulator)

获取稳压器驱动程序数据

**参数**

struct regulator \*regulator

稳压器

**说明**

获取稳压器驱动程序的私有数据。当需要调用非API稳压器特定函数时，可以在使用者驱动程序上下文中使用此调用。

### void regulator\_set\_drvdata(struct regulator \*regulator, void \*data)

设置稳压器驱动程序数据

**参数**

struct regulator \*regulator

稳压器

void \*data

数据

### int rdev\_get\_id(**struct** regulator\_dev \*rdev)

获取稳压器ID

**参数**

struct regulator\_dev \*rdev

稳压器

# 工业输入/输出

## 介绍

工业输入/输出子系统（IIO）的主要目的是为某种程度上执行模拟到数字转换（ADC）或数字到模拟转换（DAC）或两者都执行的设备提供支持。目的是填补类似的hwmon和输入子系统之间的差距。Hwmon针对低采样率的传感器，用于监视和控制系统本身，例如风扇速度控制或温度测量。输入就如其名称所示，专注于人机交互输入设备（键盘，鼠标，触摸屏）。在某些情况下，这些和IIO之间存在显着的重叠。

属于此类别的设备包括：

模数转换器（ADC）

加速度计

电容数字转换器（CDC）

数字模拟转换器（DAC）

陀螺仪

惯性测量单元（IMU）

颜色和光传感器

磁力计

压力传感器

接近传感器

温度传感器

通常，这些传感器通过SPI或I2C连接。传感器设备的常见用例是具有组合功能（例如光加距离传感器）。

## 核心元素

工业I / O核心提供了编写许多不同类型的嵌入式传感器驱动程序的统一框架，以及用于操作传感器的用户空间应用程序的标准接口。实现可以在drivers / iio / industrialio-\*下找到

### 工业I / O设备

struct iio\_dev - 工业I / O设备

iio\_device\_alloc() - 从驱动程序中分配iio\_dev

iio\_device\_free() - 从驱动程序释放iio\_dev

iio\_device\_register() - 在IIO子系统中注册设备

iio\_device\_unregister() - 在IIO子系统中注销设备

IIO设备通常对应于单个硬件传感器，并且它提供驱动程序处理设备所需的所有信息。首先让我们查看嵌入在IIO设备中的功能，然后我们将展示设备驱动程序如何使用IIO设备。

用户空间应用程序与IIO驱动程序交互的方式有两种。

1）/ sys / bus / iio / iiodeviceX /，这表示一个硬件传感器，并将同一芯片的数据通道分组。

2）/ dev / iiodeviceX，用于缓冲数据传输和事件信息检索的字符设备节点界面。

典型的IIO驱动程序将以I2C或SPI驱动程序的形式进行注册，并将创建两个例程，即probe和remove。

在探测中：

1）调用iio\_device\_alloc()，该函数为IIO设备分配内存。

2）使用驱动程序特定信息（例如设备名称，设备通道）初始化IIO设备字段。

3）调用iio\_device\_register()，这将在IIO核心中注册设备。此调用后，设备准备好接受来自用户空间应用程序的请求。

在删除时，我们以相反的顺序释放在探测中分配的资源

1）iio\_device\_unregister()，将设备从IIO核心注销。

2）iio\_device\_free()，释放为IIO设备分配的内存。

#### IIO设备sysfs接口

属性是用于公开芯片信息并允许应用程序设置各种配置参数的sysfs文件。对于索引X的设备，属性可以在/sys/bus/iio/iio:deviceX/目录下找到。常见属性包括

1. 名称，物理芯片的说明。
2. dev，显示与/dev/iiodeviceX节点相关的majorminor对。
3. sampling\_frequency\_available，可用离散的采样频率值集，用于设备。
4. IIO设备的可用标准属性在Linux内核源代码的Documentation/ABI/testing/sysfs-bus-iio文件中说明。

#### IIO设备通道

struct iio\_chan\_spec - 单个通道的规范。

IIO设备通道是数据通道的表示。 IIO设备可以具有一个或多个通道。例如：

1. 温度计传感器具有一个代表温度测量的通道。
2. 具有两个通道的光传感器指示可见和红外光谱中的测量结果。
3. 加速度计可以具有最多3个通道，表示沿X，Y和Z轴的加速度。

IIO通道由struct iio\_chan\_spec说明。例如，上述温度传感器的温度计驱动程序需要将其通道说明如下:

static const struct iio\_chan\_spec temp\_channel[] = {

{

.type = IIO\_TEMP,

.info\_mask\_separate = BIT(IIO\_CHAN\_INFO\_PROCESSED),

},

};

Channel sysfs属性向用户空间公开，以位掩码形式指定。基于它们的共享信息，属性可以在以下掩码之一中设置：

info\_mask\_separate，属性将具体适用于此通道;

info\_mask\_shared\_by\_type，属性由相同类型的所有通道共享。

info\_mask\_shared\_by\_dir，属性由相同方向的所有通道共享。

info\_mask\_shared\_by\_all，属性由所有通道共享。

当每种通道类型有多个数据通道时，我们有两种方法可以区分：

1. 它们将iio\_chan\_spec的.modified字段设置为1。修饰符是使用相同的iio\_chan\_spec结构的.channel2字段指定的，用于指示频道的物理唯一特征，例如其方向或光谱响应。例如，光传感器可以具有两个通道，一个用于红外光，一个用于红外和可见光。
2. 将iio\_chan\_spec .indexed字段设置为1.在这种情况下，通道只是采用.channel字段指定的索引的另一个实例。

以下是如何使用渠道的修改器:

static const struct iio\_chan\_spec light\_channels[] = {

{

.type = IIO\_INTENSITY,

.modified = 1,

.channel2 = IIO\_MOD\_LIGHT\_IR,

.info\_mask\_separate = BIT(IIO\_CHAN\_INFO\_RAW),

.info\_mask\_shared = BIT(IIO\_CHAN\_INFO\_SAMP\_FREQ),

},

{

.type = IIO\_INTENSITY,

.modified = 1,

.channel2 = IIO\_MOD\_LIGHT\_BOTH,

.info\_mask\_separate = BIT(IIO\_CHAN\_INFO\_RAW),

.info\_mask\_shared = BIT(IIO\_CHAN\_INFO\_SAMP\_FREQ),

},

{

.type = IIO\_LIGHT,

.info\_mask\_separate=BIT(IIO\_CHAN\_INFO\_PROCESSED),

.info\_mask\_shared = BIT(IIO\_CHAN\_INFO\_SAMP\_FREQ),

},

}

该通道的定义将生成两个用于原始数据检索的单独的sysfs文件：

/sys/bus/iio/iiodeviceX / in\_intensity\_ir\_raw

/ sys / bus / iio / iiodeviceX / in\_intensity\_both\_raw

一个用于处理数据的文件:

/sys/bus/iio/iiodeviceX / in\_illuminance\_input

用于采样频率的一个共享sysfs文件:

/sys/bus/iio/iiodeviceX / sampling\_frequency。

以下是如何使用通道的索引:

static const struct iio\_chan\_spec light\_channels[] = {

{

.type = IIO\_VOLTAGE,

.indexed = 1,

.channel = 0,

.info\_mask\_separate = BIT(IIO\_CHAN\_INFO\_RAW),

},

{

.type = IIO\_VOLTAGE,

.indexed = 1,

.channel = 1,

.info\_mask\_separate = BIT(IIO\_CHAN\_INFO\_RAW),

},

}

这将为原始数据检索生成两个单独的属性文件：

/sys/bus/iio/devices / iiodeviceX / in\_voltage0\_raw，表示通道0的电压测量；

/sys/bus/iio/devices/iiodeviceX / in\_voltage1\_raw，表示通道1的电压测量。

#### 更多详细信息

**struct iio\_chan\_spec\_ext\_info**

扩展通道信息属性

1）定义

struct iio\_chan\_spec\_ext\_info {

const char \*name;

enum iio\_shared\_by shared;

ssize\_t (\*read)(struct iio\_dev \*, uintptr\_t private, struct iio\_chan\_spec const \*, char \*buf);

ssize\_t (\*write)(struct iio\_dev \*, uintptr\_t private,struct iio\_chan\_spec const \*, const char \*buf, size\_t len);

uintptr\_t private;

};

2）成员

name

信息属性名称

shared

此属性是否在所有通道之间共享。

read

信息属性的读取回调，可以为NULL

write

信息属性的写回调，可以为NULL

private

数据专用于驱动程序

**struct iio\_enum Enum**

通道信息属性

1）定义

struct iio\_enum {

const char \* const \*items;

unsigned int num\_items;

int (\*set)(struct iio\_dev \*, const struct iio\_chan\_spec \*, unsigned int);

int (\*get)(struct iio\_dev \*, const struct iio\_chan\_spec \*);

};

2）成员

items

字符串数组。

num\_items

项目数组的长度。

set

设置回调函数，可以为NULL。

get

获取回调函数，可以为 NULL。

1. 说明
2. iio\_enum 结构可用于实现枚举样式的通道属性。枚举样式属性是那些具有一组映射到无符号整数值的字符串的属性。IIO 枚举帮助程序代码负责值和字符串之间的映射，并生成一个包含所有可用项目列表的“\_available”文件。更新属性时将调用设置回调。最后一个参数是新激活项目的索引。get 回调将用于查询当前活动的项目，并应该为其返回索引。

**IIO\_ENUM( \_name , \_shared , \_e )**

初始化枚举扩展通道属性

1）参数

\_name

属性名称

\_shared

该属性是否在所有通道之间共享

\_e

指向 iio\_enum 结构的指针

2）说明

这通常应该与一起使用[IIO\_ENUM\_AVAILABLE()](https://www.kernel.org/doc/html/v4.19/driver-api/iio/core.html" \l "c.IIO_ENUM_AVAILABLE" \o "IIO_ENUM_AVAILABLE)

**IIO\_ENUM\_AVAILABLE( \_name , \_e )**

初始化枚举可用的扩展通道属性

1）参数

\_name

属性名称（“\_available”将附加到名称中）

\_e

指向 iio\_enum 结构的指针

2）说明

创建一个只读属性，它在空格分隔的列表中列出所有可用的枚举项。这通常应该与一起使用[IIO\_ENUM()](https://www.kernel.org/doc/html/v4.19/driver-api/iio/core.html" \l "c.IIO_ENUM" \o "IIO_ENUM)

**struct iio\_mount\_matrix**

iio装配矩阵

1）定义

struct iio\_mount\_matrix {

const char \*rotation[9];

};

2）成员

rotation

三维空间旋转矩阵，定义传感器与主要硬件的对齐方式

**IIO\_MOUNT\_MATRIX**

IIO\_MOUNT\_MATRIX（\_shared，\_get）

初始化安装矩阵扩展通道属性

1）参数

\_shared

属性是否在所有通道之间共享

\_get

指向iio\_get\_mount\_matrix\_t访问器的指针

**struct iio\_event\_spec**

通道事件的规格

1）定义

struct iio\_event\_spec {

enum iio\_event\_type type;

enum iio\_event\_direction dir;

unsigned long mask\_separate;

unsigned long mask\_shared\_by\_type;

unsigned long mask\_shared\_by\_dir;

unsigned long mask\_shared\_by\_all;

};

2）成员

type

事件类型

dir

事件的方向

mask\_separate

枚举iio\_event\_info值的位掩码。在此掩码中设置的属性将按通道注册。

mask\_shared\_by\_type

枚举iio\_event\_info值的位掩码。在此掩码中设置的属性将由通道类型共享。

mask\_shared\_by\_dir

枚举iio\_event\_info值的位掩码。在此掩码中设置的属性将由通道类型和方向共享。

mask\_shared\_by\_all

枚举iio\_event\_info值的位掩码。在此掩码中设置的属性将由所有通道共享。

**struct iio\_chan\_spec**

单个通道的规格

**定义**

struct iio\_chan\_spec {

enum iio\_chan\_type type;

int channel;

int channel2;

unsigned long address;

int scan\_index;

struct {

char sign;

u8 realbits;

u8 storagebits;

u8 shift;

u8 repeat;

enum iio\_endian endianness;

} scan\_type;

long info\_mask\_separate;

long info\_mask\_separate\_available;

long info\_mask\_shared\_by\_type;

long info\_mask\_shared\_by\_type\_available;

long info\_mask\_shared\_by\_dir;

long info\_mask\_shared\_by\_dir\_available;

long info\_mask\_shared\_by\_all;

long info\_mask\_shared\_by\_all\_available;

const struct iio\_event\_spec \*event\_spec;

unsigned int num\_event\_specs;

const struct iio\_chan\_spec\_ext\_info \*ext\_info;

const char \*extend\_name;

const char \*datasheet\_name;

unsigned modified:1;

unsigned indexed:1;

unsigned output:1;

unsigned differential:1;

};

2）成员

type

通道正在进行何种类型的测量

channel

我们希望分配给通道的数字

channel2

如果有差分通道的第二个数字，则为该数字。如果设置了修改符，则此处的值指定修饰符。

address

与驱动程序特定的标识符。

scan\_index

调用缓冲区中的扫描顺序的单调索引。

scan\_type

说明扫描类型的结构

scan\_type.sign

“s”或“u”以指定有符号或无符号

scan\_type.realbits

数据有效位数

scan\_type.storagebits

Realbits + padding

scan\_type.shift

在遮罩掉realbits之前向右移动此数量。

scan\_type.repeat

实体/存储位重复的次数。当重复元素大于1时，sysfs中的类型元素将显示重复值。否则，省略重复次数。

scan\_type.endianness

小端或大端

info\_mask\_separate

要导出的与此通道特定的信息。

info\_mask\_separate\_available

要导出的特定于此通道的可用性信息。

info\_mask\_shared\_by\_type

应由所有同一类型的通道共享的要导出的信息。

info\_mask\_shared\_by\_type\_available

应由所有同一类型的通道共享的要导出的可用性信息。

info\_mask\_shared\_by\_dir

应由所有具有相同方向的通道共享的要导出的信息。

info\_mask\_shared\_by\_dir\_available

应由所有具有相同方向的通道共享的要导出的可用性信息。

info\_mask\_shared\_by\_all

应由所有通道共享的要导出的信息。

info\_mask\_shared\_by\_all\_available

应由所有通道共享的要导出的可用性信息。

event\_spec

应为此通道注册的事件数组。

num\_event\_specs

event\_spec数组的大小。

ext\_info

此通道的扩展信息属性数组。数组以NULL结尾，最后一个元素应将其名称字段设置为NULL。

extend\_name

允许使用传递性名称标记通道属性。请注意，与修饰符不同，这对代码等没有任何影响。

datasheet\_name

在内核映射通道中使用的名称。它应对应于在数据表中第一个引用通道的名称（例如，IND），或最近可能的化合物名称（例如，IND-INC）。

modified

此通道是否应用修饰符。这些因通道类型而异。修饰符设置为channel2。例如，关于“X”轴的轴向传感器的IIO\_MOD\_X。

indexed

指定通道具有数字索引。如果没有，则将抑制sysfs属性的通道索引号，但不会抑制事件代码。

output

通道是输出。

differential

通道是差分的。

**bool iio\_channel\_has\_info（const struct iio\_chan\_spec \* chan，enum iio\_chan\_info\_enum type）**

检查通道是否支持信息属性

1）参数

const struct iio\_chan\_spec \* chan

要查询的通道

enumiio\_chan\_info\_enum type

要检查的信息属性类型

2）说明

如果通道支持报告给定信息属性类型的值，则返回true，否则返回false。

**bool iio\_channel\_has\_available(const struct iio\_chan\_spec \*chan, enum iio\_chan\_info\_enum type)**

检查通道是否有可用属性

1）参数

const struct iio\_chan\_spec \*chan

要查询的通道

enum iio\_chan\_info\_enum type

要检查的可用属性的类型

2）说明

如果通道支持报告给定属性类型的可用值，则返回true，否则返回false。

**struct iio\_info**

设备的常量信息

1）定义

struct iio\_info {

const struct attribute\_group \*event\_attrs;

const struct attribute\_group \*attrs;

int (\*read\_raw)(struct iio\_dev \*indio\_dev,struct iio\_chan\_spec const \*chan,int \*val,int \*val2, long mask);

int (\*read\_raw\_multi)(struct iio\_dev \*indio\_dev,struct iio\_chan\_spec const \*chan,int max\_len,int \*vals,int \*val\_len, long mask);

int (\*read\_avail)(struct iio\_dev \*indio\_dev,struct iio\_chan\_spec const \*chan,const int \*\*vals,int \*type,int \*length, long mask);

int (\*write\_raw)(struct iio\_dev \*indio\_dev,struct iio\_chan\_spec const \*chan,int val,int val2, long mask);

int (\*read\_label)(struct iio\_dev \*indio\_dev,struct iio\_chan\_spec const \*chan, char \*label);

int (\*write\_raw\_get\_fmt)(struct iio\_dev \*indio\_dev,struct iio\_chan\_spec const \*chan, long mask);

int (\*read\_event\_config)(struct iio\_dev \*indio\_dev,const struct iio\_chan\_spec \*chan,enum iio\_event\_type type, enum iio\_event\_direction dir);

int (\*write\_event\_config)(struct iio\_dev \*indio\_dev,const struct iio\_chan\_spec \*chan,enum iio\_event\_type type,enum iio\_event\_direction dir, int state);

int (\*read\_event\_value)(struct iio\_dev \*indio\_dev,const struct iio\_chan\_spec \*chan,enum iio\_event\_type type,enum iio\_event\_direction dir, enum iio\_event\_info info, int \*val, int \*val2);

int (\*write\_event\_value)(struct iio\_dev \*indio\_dev,const struct iio\_chan\_spec \*chan,enum iio\_event\_type type,enum iio\_event\_direction dir, enum iio\_event\_info info, int val, int val2);

int (\*validate\_trigger)(struct iio\_dev \*indio\_dev, struct iio\_trigger \*trig);

int (\*update\_scan\_mode)(struct iio\_dev \*indio\_dev, const unsigned long \*scan\_mask);

int (\*debugfs\_reg\_access)(struct iio\_dev \*indio\_dev,unsigned reg, unsigned writeval, unsigned \*readval);

int (\*fwnode\_xlate)(struct iio\_dev \*indio\_dev, const struct fwnode\_reference\_args \*iiospec);

int (\*hwfifo\_set\_watermark)(struct iio\_dev \*indio\_dev, unsigned val);

int (\*hwfifo\_flush\_to\_buffer)(struct iio\_dev \*indio\_dev, unsigned count);

};

2）成员

event\_attrs

事件控制属性

attrs

通用设备属性

read\_raw

从设备请求值的函数。掩码指定哪个值。注意0表示查询所讨论的通道。返回将指定设备返回的值类型。val和val2将包含组成返回的元素。

read\_raw\_multi

从设备返回的函数。掩码指定哪个值。注意0表示查询所讨论的通道。返回将指定设备返回的值类型。vals指针包含组成返回的元素。max\_len指定vals指针可以包含的最大元素数。val\_len用于返回vals中有效元素的长度。

read\_avail

从设备返回可用值的函数。掩码指定哪个值。注意0表示查询所讨论的通道的可用值。返回指定vals中是否返回IIO\_AVAIL\_LIST或IIO\_AVAIL\_RANGE。返回类型在type中返回，并且值的数量在length中返回。对于范围而言，始终返回三个vals；最小值、步长和最大值。对于列表，enum所有可能的值。

write\_raw

向设备写入值的函数。参数与read\_raw相同。

read\_label

请求指定标签的标签名称的函数，以更好地识别通道。

write\_raw\_get\_fmt

查询预期格式/精度的回调函数。如果驱动程序未设置，则write\_raw返回IIO\_VAL\_INT\_PLUS\_MICRO。

read\_event\_config

了解事件是否启用。

write\_event\_config

设置事件是否启用。

read\_event\_value

读取与事件关联的配置值。

write\_event\_value

为事件写入配置值。

validate\_trigger

当更改当前触发器时，验证触发器的函数。

update\_scan\_mode

配置设备和扫描缓冲区，当通道已更改时

debugfs\_reg\_access

读取或写入设备的寄存器值的函数

fwnode\_xlate

基于fwnode的函数指针，用于获取通道指示器索引。与of\_xlate功能相同。

hwfifo\_set\_watermark

函数指针，用于设置当前硬件fifo水印级别；有关硬件fifo的操作，请参阅Documentation/ABI/testing/sysfs-bus-iio中的hwfifo\_\*条目

hwfifo\_flush\_to\_buffer

函数指针，用于将存储在硬件fifo中的样本刷新到设备缓冲区。驱动程序不应刷新超过计数的样本。该函数必须返回刷新的样本数，如果未刷新样本，则返回0，如果未刷新样本并且存在错误，则返回负整数。

**struct iio\_buffer\_setup\_ops**

缓冲区设置相关的回调函数

1）定义

struct iio\_buffer\_setup\_ops {

int (\*preenable)(struct iio\_dev \*);

int (\*postenable)(struct iio\_dev \*);

int (\*predisable)(struct iio\_dev \*);

int (\*postdisable)(struct iio\_dev \*);

bool (\*validate\_scan\_mask)(struct iio\_dev \*indio\_dev, const unsigned long \*scan\_mask);

};

2）成员

preenable

[驱动程序]在标记缓冲区已启用之前运行的函数

postenable

[驱动程序]在标记缓冲区已启用之后运行的函数

predisable

[驱动程序] 在标记缓冲区禁用之前运行的函数

postdisable

[驱动程序] 在标记缓冲区禁用之后运行的函数

validate\_scan\_mask

[驱动程序] 用于检查给定扫描掩码是否对设备有效的回调函数

**struct iio\_dev**

工业输入输出设备

1）定义

struct iio\_dev {

int id;

struct module \*driver\_module;

int modes;

int currentmode;

struct device dev;

struct iio\_event\_interface \*event\_interface;

struct iio\_buffer \*buffer;

struct list\_head buffer\_list;

int scan\_bytes;

struct mutex mlock;

const unsigned long \*available\_scan\_masks;

unsigned masklength;

const unsigned long \*active\_scan\_mask;

bool scan\_timestamp;

unsigned scan\_index\_timestamp;

struct iio\_trigger \*trig;

bool trig\_readonly;

struct iio\_poll\_func \*pollfunc;

struct iio\_poll\_func \*pollfunc\_event;

struct iio\_chan\_spec const \*channels;

int num\_channels;

struct list\_head channel\_attr\_list;

struct attribute\_group chan\_attr\_group;

const char \*name;

const struct iio\_info \*info;

clockid\_t clock\_id;

struct mutex info\_exist\_lock;

const struct iio\_buffer\_setup\_ops \*setup\_ops;

struct cdev chrdev;

#define IIO\_MAX\_GROUPS 6;

const struct attribute\_group \*groups[IIO\_MAX\_GROUPS + 1];

int groupcounter;

unsigned long flags;

#if defined(CONFIG\_DEBUG\_FS);

struct dentry \*debugfs\_dentry;

unsigned cached\_reg\_addr;

#endif;

};

2）成员

id

[INTERN] 用于在内部识别设备

driver\_module

[INTERN] 曾经让削弱用户变得更加困难

modes

[DRIVER] 设备支持的操作模式

currentmode

[DRIVER] 当前运行模式

dev

[DRIVER] 设备结构，应分配父级和所有者

event\_interface

[实习生] 与中断线关联的事件 chrdev

buffer

[驱动程序] 存在任何缓冲区

buffer\_list

[INTERN] 当前附加的所有缓冲区列表

scan\_bytes

[实习生] 捕获的字节数将馈送到缓冲区多路分解器

mlock

[DRIVER] 锁用于防止同时设备状态更改

available\_scan\_masks

[DRIVER] 可选的允许位掩码数组

masklength

[实习生] 从渠道建立的面具的长度

active\_scan\_mask

[实习生] 缓冲区请求的所有扫描掩码的联合

scan\_timestamp

[INTERN] 设置是否有任何缓冲区请求时间戳

scan\_index\_timestamp

[INTERN] 索引到时间戳的缓存

trig

[INTERN] 当前设备触发器（缓冲模式）

trig\_readonly

[INTERN] 将当前触发器标记为不可变

pollfunc

[DRIVER] 函数在接收到触发器时运行

pollfunc\_event

[DRIVER] 函数在接收到事件触发器时运行

channels

[DRIVER]通道规格结构表

num\_channels

[DRIVER] channels中指定的通道数。

channel\_attr\_list

[实习生] 跟踪自动创建的频道属性

chan\_attr\_group

[INTERN] 基本目录中所有属性的组

name

[DRIVER] 设备名称。

info

[DRIVER] 来自驱动程序的回调和常量信息

clock\_id

[实习生] 时间戳时钟 posix 标识符

info\_exist\_lock

[INTERN] 锁定以防止在移除期间使用

setup\_ops

[DRIVER] 在缓冲区启用/禁用之前和之后调用的回调

chrdev

[INTERN] 关联字符设备

groups

[实习生] 属性组

groupcounter

[INTERN] 下一个属性组的索引

flags

[INTERN] 文件操作相关标志，包括忙碌标志。

debugfs\_dentry

[实习生] 设备特定的 debugfs 目录。

cached\_reg\_addr

[INTERN] 用于 debugfs 读取的缓存寄存器地址。

**iio\_device\_register**

iio\_device\_register (indio\_dev)

在 IIO 子系统中注册一个设备

1）参数

indio\_dev

由设备驱动程序填充的设备结构

**devm\_iio\_device\_register**

devm\_iio\_device\_register (dev, indio\_dev)

资源管理的 iio\_device\_register()

1）参数

dev

要为其分配 iio\_dev 的设备

indio\_dev

由设备驱动程序填充的设备结构

2）说明

管理的 iio\_device\_register。使用此函数注册的 IIO 设备在驱动程序取消加载时会自动注销。此函数在内部调用 iio\_device\_register()。有关更多信息，请参见该函数。

3）返回

成功时为 0，在失败时为负的错误数。

**void iio\_device\_put(struct iio\_dev \*indio\_dev)**

struct 设备的引用计数释放

1）参数

struct iio\_dev \*indio\_dev

包含设备的 IIO 设备结构

**clockid\_t iio\_device\_get\_clock(const struct iio\_dev \* indio\_dev)**

检索设备的当前时间戳时钟

1）参数

const struct iio\_dev \* indio\_dev

包含设备的 IIO 设备结构

**sstruct iio\_dev \*dev\_to\_iio\_dev(struct device \*dev)**

从设备结构中获取 IIO 设备结构

1）参数

struct device \*dev

嵌入在 IIO 设备中的设备

2）注意

设备必须是 IIO 设备，否则结果是未定义的。

**struct iio\_dev \*iio\_device\_get(struct iio\_dev \*indio\_dev)**

增加设备的引用计数

1）参数

struct iio\_dev \*indio\_dev

IIO 设备结构

2）返回

所传递的 IIO 设备

**void iio\_device\_set\_drvdata(struct iio\_dev \*indio\_dev, void \*data)**

设置设备驱动程序数据

1）参数

struct iio\_dev \*indio\_dev

IIO 设备结构

void \*data

驱动程序特定数据

2）说明

允许将任意指针附加到 IIO 设备，稍后可以使用 iio\_device\_get\_drvdata() 检索它。

**void \*iio\_device\_get\_drvdata(const struct iio\_dev \*indio\_dev)**

获取设备驱动程序数据

1）参数

const struct iio\_dev \*indio\_dev

IIO 设备结构

2）说明

返回之前使用 iio\_device\_set\_drvdata() 设置的数据

**bool iio\_buffer\_enabled(struct iio\_dev \* indio\_dev**)

帮助函数，用于测试缓冲区是否已启用

1）参数

struct iio\_dev \* indio\_dev

设备的 IIO 设备结构

**struct dentry \*iio\_get\_debugfs\_dentry(struct iio\_dev \*indio\_dev)**

获取 debugfs\_dentry 的帮助函数

1）参数

struct iio\_dev \*indio\_dev

设备的 IIO 设备结构

**IIO\_DEGREE\_TO\_RAD**

IIO\_DEGREE\_TO\_RAD (deg)

将角度转换为弧度

1）参数

deg

角度值

2）说明

返回将给定值从角度转换为弧度的结果

**IIO\_RAD\_TO\_DEGREE**

IIO\_RAD\_TO\_DEGREE (rad)

将弧度转换为角度

1）参数

rad

弧度值

2）说明

返回将给定值从弧度转换为角度的结果

**IIO\_G\_TO\_M\_S\_2**

IIO\_G\_TO\_M\_S\_2 (g)

将 g 转换为米/秒平方

1）参数

g

g 值

2）说明

返回将给定值从 g 转换为米/秒平方的结果

**IIO\_M\_S\_2\_TO\_G**

IIO\_M\_S\_2\_TO\_G (ms2)

将米/秒平方转换为 g

1）参数

ms2

米/秒平方的值

2）说明

返回将给定值从米/秒平方转换为 g 的结果

**s64 iio\_get\_time\_ns(const struct iio\_dev \*indio\_dev)**

用于获取事件时间戳等的实用函数

1）参数

const struct iio\_dev \*indio\_dev

设备

**unsigned int iio\_get\_time\_res(const struct iio\_dev \* indio\_dev)**

用于获取纳秒为单位的时间戳时钟分辨率的实用函数。

1）参数

const struct iio\_dev \* indio\_dev

设备

**int iio\_read\_mount\_matrix(struct device \*dev, struct iio\_mount\_matrix \*matrix)**

从设备“mount-matrix”属性检索 iio 设备的安装矩阵

1）参数

struct device \*dev

被分配装配矩阵属性的设备

struct iio\_mount\_matrix \*matrix

用于存储检索到的矩阵

2）说明

如果设备没有被分配装配矩阵属性，则会填充默认的3x3身份矩阵。

3）返回

成功返回0，失败返回负错误代码。

**ssize\_t iio\_format\_value(char \*buf, unsigned int type, int size, int \*vals)**

将IIO值格式化为其字符串表示形式

1）参数

char \*buf

被写入格式化值的缓冲区，假定足够大（即PAGE\_SIZE）。

unsigned int type

IIO\_VAL\_\*常量之一。这决定了如何格式化val和val2参数。

int size

vals中包含的IIO值条目数

int \*vals

指向值的指针，确切的含义取决于type参数。

3）返回

默认为0，失败返回负数，对于属于IIO\_VAL\_\*常量的类型，返回写入的字符总数。

**int iio\_str\_to\_fixpoint(const char \*str, int fract\_mult, int \*integer, int \*fract)**

从字符串中解析一个定点数

1）参数

const char \*str

要解析的字符串

int fract\_mult

第一个小数位的乘数，应该是10的幂

int \*integer

数字的整数部分

int \*fract

数字的小数部分

2）说明

如果成功则返回0，否则返回负错误代码。

**struct iio\_dev \*iio\_device\_alloc(struct device \*parent, int sizeof\_priv)**

从驱动程序中分配一个iio\_dev

1）参数

struct device \*parent

父设备。

int sizeof\_priv

为私有结构分配的空间。

**void iio\_device\_free(struct iio\_dev \*dev)**

从驱动程序中释放iio\_dev

1）参数

struct iio\_dev \*dev

与设备相关联的iio\_dev

**struct iio\_dev \*devm\_iio\_device\_alloc(struct device \*parent, int sizeof\_priv)**

资源管理的iio\_device\_alloc()

1）参数

struct device \*parent

用于分配iio\_dev的设备，并用于该IIO设备的父级

int sizeof\_priv

为私有结构分配的空间。

2）说明

托管iio\_device\_alloc。使用此函数分配的iio\_dev会在驱动程序分离时自动释放。

3）返回

成功时返回分配的iio\_dev指针，失败时返回NULL。

**void devm\_iio\_device\_free(struct device \* dev, struct iio\_dev \* iio\_dev)**

资源管理的iio\_device\_free()

1）参数

struct device \* dev

此iio\_dev所属的设备

struct iio\_dev \* iio\_dev

与设备相关联的iio\_dev

2）说明

释放使用devm\_iio\_device\_alloc()分配的iio\_dev。

**void iio\_device\_unregister(struct iio\_dev \*indio\_dev)**

从IIO子系统注销设备

1）参数

struct iio\_dev \*indio\_dev

表示该设备的设备结构。

**void devm\_iio\_device\_unregister(struct device \* dev, struct iio\_dev \* indio\_dev)**

资源管理的iio\_device\_unregister()

1）参数

struct device \* dev

此iio\_dev所属的设备

struct iio\_dev \* indio\_dev

与设备相关联的iio\_dev

2）说明

注销使用devm\_iio\_device\_register()注册的iio\_dev。

**int iio\_device\_claim\_direct\_mode(struct iio\_dev \*indio\_dev)**

将设备保持在直接模式

1）参数

struct iio\_dev \*indio\_dev

与设备相关联的iio\_dev

2）说明

如果设备处于直接模式，则保证该设备将保持在该模式，直到调用iio\_device\_release\_direct\_mode()。

与iio\_device\_release\_direct\_mode()一起使用

3）返回

成功返回0，失败返回-EBUSY。

**void iio\_device\_release\_direct\_mode(struct iio\_dev \*indio\_dev)**

释放对直接模式的控制

1）参数

struct iio\_dev \*indio\_dev

与设备相关联的iio\_dev

2）说明

释放控制权，设备不再保证保持在直接模式中。

与iio\_device\_claim\_direct\_mode()一起使用

## 缓冲

struct iio\_buffer — 通用缓冲区结构

iio\_validate\_scan\_mask\_onehot() — 验证仅有一个通道被选中

iio\_buffer\_get() — 获取对缓冲区的引用

iio\_buffer\_put() — 释放对缓冲区的引用

工业I/O核心提供了一种基于触发源的连续数据捕获方式。可以从/dev/iio:deviceX字符设备节点同时读取多个数据通道，从而减少CPU负载。

### IIO缓冲区sysfs接口

IIO缓冲区有一个关联的属性目录，在/sys/bus/iio/iio:deviceX/buffer/下有一些现有的属性

length，可以由缓冲区存储的数据样本（容量）总数。

enable，激活缓冲区捕获。

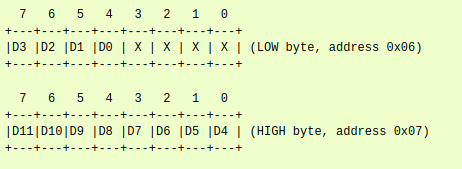
### IIO缓冲区设置

放置在缓冲区中的通道读取的元信息称为扫描元素。配置扫描元素的重要位暴露给用户空间应用程序，通过/sys/bus/iio/iio:deviceX/scan\_elements/目录。该目录包含以下形式的属性：

enable，用于启用通道。仅在其属性为非零时，触发捕获才会包含此通道的数据样本。

type，缓冲区中的扫描元素数据存储的说明，因此也决定了从用户空间读取它的形式。格式为[be|le][s|u]bits/storagebits[Xrepeat][>>shift]。be或le，指定大端或小端。s或u，指定是否为有符号（2的补码）或无符号。bits，有效数据位数。storagebits，它在缓冲区中占用的位数（填充后）。重复指定了位数/存储位数的重复次数。当重复元素为0或1时，重复值被省略。如果指定shift，则需要在掩码未使用的位之前应用的移位。

例如，具有12位分辨率的3轴加速度计的驱动程序，其中数据存储在两个8位寄存器中，如下所示：:



将具有以下每个轴的扫描元素类型:

$ cat /sys/bus/iio/devices/iio:device0/scan\_elements/in\_accel\_y\_type

le:s12/16>>4

用户空间应用程序将从缓冲区中读取的数据样本解释为两个字节的小端有符号数据，需要在掩码掉12个有效数据位之前右移4个位。

为实现缓冲区支持，驱动程序应初始化iio\_chan\_spec定义中的以下字段

struct iio\_chan\_spec {

/\* other 成员 \*/

int scan\_index

struct {

char sign;

u8 realbits;

u8 storagebits;

u8 shift;

u8 repeat;

enum iio\_endian endianness;

} scan\_type;

};

以上述加速度计实现的驱动程序将具有以下通道定义:

struct iio\_chan\_spec accel\_channels[] = {

{

.type = IIO\_ACCEL,

.modified = 1,

.channel2 = IIO\_MOD\_X,

/\* other stuff here \*/

.scan\_index = 0,

.scan\_type = {

.sign = 's',

.realbits = 12,

.storagebits = 16,

.shift = 4,

.endianness = IIO\_LE,

},

}

/\* similar for Y (with channel2 = IIO\_MOD\_Y, scan\_index = 1)

\* and Z (with channel2 = IIO\_MOD\_Z, scan\_index = 2) axis

\*/

}

在此，scan\_index定义了启用通道放置于缓冲区内的顺序。具有较低scan\_index的通道将位于具有较高索引的通道之前。每个通道都需要具有唯一的scan\_index。

将scan\_index设置为-1可用于指示特定通道不支持缓冲区捕获。在这种情况下，在scan\_elements目录中不会为通道创建任何条目。

### 更多细节

#### int iio\_push\_to\_buffers\_with\_timestamp(struct iio\_dev \*indio\_dev, void \*data, int64\_t timestamp)

将数据和时间戳推送到缓冲区

**参数**

struct iio\_dev \*indio\_dev

设备的iio\_dev结构。

void \*data

样本数据

int64\_t timestamp

样本数据的时间戳

**说明**

将数据推送到IIO设备的缓冲区中。如果设备启用了时间戳，函数将在将其推送到设备缓冲区之前将提供的时间戳存储为样本数据缓冲区中的最后一个元素。样本数据缓冲区需要足够大以容纳附加的时间戳（通常应该是indio->scan\_bytes字节大）。

成功返回0，否则返回负错误代码。

#### void iio\_buffer\_set\_attrs( struct iio\_buffer \*  buffer , const struct attribute \*\*  attrs )

设置缓冲区特定属性

**参数**

struct iio\_buffer \* buffer

我们正在为其设置属性的缓冲区

const struct attribute \*\* attrs

指向指向属性的指针的空终止列表的指针

#### bool iio\_validate\_scan\_mask\_onehot(struct iio\_dev \*indio\_dev, const unsigned long \*mask)

验证只选择了一个通道

**参数**

struct iio\_dev \*indio\_dev

iio设备

const unsigned long \*mask

要检查的扫描掩码

**说明**

当只能为一个通道激活采样时，返回true，否则返回false。

#### int iio\_push\_to\_buffers(struct iio\_dev \*indio\_dev, const void \*data)

推送到已注册的缓冲区。

**参数**

struct iio\_dev \*indio\_dev

设备的iio\_dev结构。

const void \*data

完整扫描。

#### struct iio\_buffer \*iio\_buffer\_get(struct iio\_buffer \*buffer)

获取对缓冲区的引用

**参数**

struct iio\_buffer \*buffer

要为其获取引用的缓冲区，可以为NULL

**说明**

返回传递到函数中的缓冲区的指针。

#### void iio\_buffer\_put(struct iio\_buffer \*buffer)

释放对缓冲区的引用

**参数**

struct iio\_buffer \*buffer

要释放引用的缓冲区，可以为NULL

#### int iio\_device\_attach\_buffer(struct iio\_dev \*indio\_dev, struct iio\_buffer \*buffer)

将缓冲区附加到IIO设备

**参数**

struct iio\_dev \*indio\_dev

缓冲区应附加到的设备

struct iio\_buffer \*buffer

要附加到设备的缓冲区

**说明**

如果成功则返回0，否则为负数。

此函数将缓冲区附加到IIO设备。缓冲区将附加到设备，直到释放设备。由于遗留原因，第一个附加的缓冲区也将分配给“indio\_dev->buffer”。在此处分配的数组将通过iio\_device\_detach\_buffers()调用释放，该调用由iio\_device\_free()处理。

## 触发器

struct iio\_trigger – 工业I/O触发器设备

devm\_iio\_trigger\_alloc() – 资源管理iio\_trigger\_alloc

devm\_iio\_trigger\_register() – 资源管理iio\_trigger\_register iio\_trigger\_unregister

iio\_trigger\_validate\_own\_device() – 检查触发器和IIO设备是否属于同一个设备

在许多情况下，驱动程序能够基于一些外部事件（触发器）而不是周期性地轮询数据来捕获数据非常有用（触发器）。 IIO触发器可以由具有基于硬件生成的事件（例如数据准备好或阈值超出）的IIO设备的设备驱动程序或由独立于中断源的独立驱动程序（例如连接到某些外部系统的GPIO线，定时器中断或用户空间写入sysfs中的特定文件）提供。触发器可能会为多个传感器初始化数据捕捉，而且可能完全与传感器本身无关。

### IIO 触发器 sysfs 接口

与触发器相关的 sysfs 中有两个位置：

1）/sys/bus/iio/devices/triggerY/\*，这个文件是在 IIO 触发器向 IIO 核心注册后创建的，对应于索引为 Y 的触发器。由于触发器可以因类型而异，因此我们可以在这里说明一些标准属性；

name，触发器名称，可以后续用于与设备关联。

sampling\_frequency，一些基于定时器的触发器使用此属性来指定触发调用的频率。

2）/sys/bus/iio/devices/iio:deviceX/trigger/\*，此目录是在设备支持触发缓冲时创建的。我们可以通过在 current\_trigger 文件中写入触发器名称来将触发器与我们的设备关联起来。

### IIO 触发器设置

让我们看一个设置触发器供驱动程序使用的简单示例:

struct iio\_trigger\_ops trigger\_ops = {

.set\_trigger\_state = sample\_trigger\_state,

.validate\_device = sample\_validate\_device,

}

struct iio\_trigger \*trig;

/\* first, allocate memory for our trigger \*/

trig = iio\_trigger\_alloc(dev, "trig-%s-%d", name, idx);

/\* setup trigger operations field \*/

trig->ops = &trigger\_ops;

/\* now register the trigger with the IIO core \*/

iio\_trigger\_register(trig);

### IO 触发器操作

struct iio\_trigger\_ops — iio\_trigger 的操作struct 。

请注意，触发器附有一组操作：

set\_trigger\_state，按需开启/关闭触发器。

validate\_device，验证当前触发器更改时设备的函数。

### 更多详情

#### struct iio\_trigger\_ops

iio\_trigger 的操作struct 。

**定义**

struct iio\_trigger\_ops {

int (\*set\_trigger\_state)(struct iio\_trigger \*trig, bool state);

void (\*reenable)(struct iio\_trigger \*trig);

int (\*validate\_device)(struct iio\_trigger \*trig, struct iio\_dev \*indio\_dev);

};

**成员**

set\_trigger\_state

按需开启/关闭触发器

reenable

重新启用触发器的函数（使用计数为零时可能为 NULL）

validate\_device

验证当前触发器更改时设备的函数。

**说明**

这通常是驱动程序内静态常量，并由给定设备的实例共享。

#### struct iio\_trigger

工业输入/输出触发器设备。

**定义**

struct iio\_trigger {

const struct iio\_trigger\_ops \*ops;

struct module \*owner;

int id;

const char \*name;

struct device dev;

struct list\_head list;

struct list\_head alloc\_list;

atomic\_t use\_count;

struct irq\_chip subirq\_chip;

int subirq\_base;

struct iio\_subirq subirqs[CONFIG\_IIO\_CONSUMERS\_PER\_TRIGGER];

unsigned long pool[BITS\_TO\_LONGS(CONFIG\_IIO\_CONSUMERS\_PER\_TRIGGER)];

struct mutex pool\_lock;

bool attached\_own\_device;

};

**成员**

ops

[驱动程序] 操作struct

owner

[INTERN] 驱动程序模块的所有者

id

[INTERN] 唯一的 ID 号码

name

[驱动程序] 唯一的名称

dev

[驱动程序] 关联的设备（如果相关）

list

[INTERN] 用于维护全局触发器列表

alloc\_list

[驱动程序] 用于驱动程序特定的触发器列表。

use\_count

[INTERN] 触发器的使用计数。

subirq\_chip

[INTERN] 关联的“虚拟”中断控制器

subirq\_base

[INTERN] 触发器提供的中断的基础编号。

subirqs

[INTERN] 关于“子”中断的信息。

pool

[INTERN] 当前使用的中断的位图。

pool\_lock

[INTERN] 对中断池进行保护。

attached\_own\_device

[INTERN] 如果我们自己使用自己的设备作为触发器，即如果我们向同一设备注册了轮询功能，就是在使用自己的设备提供触发器。

#### void iio\_trigger\_set\_drvdata(struct iio\_trigger \*trig, void \*data)

设置触发器驱动程序数据

**参数**

struct iio\_trigger \*trig

IIO 触发器struct

void \*data

驱动程序特定的数据

**说明**

允许将任意指针附加到 IIO 触发器，稍后可以使用 iio\_trigger\_get\_drvdata() 检索该指针。

#### void \*iio\_trigger\_get\_drvdata(struct iio\_trigger \*trig)

获取触发器驱动程序数据

**参数**

struct iio\_trigger \*trig

IIO 触发器struct

**说明**

返回以前使用 iio\_trigger\_set\_drvdata() 设置的数据。

#### int iio\_trigger\_register(struct iio\_trigger \*trig\_info)

向 IIO 核心注册触发器

**参数**

struct iio\_trigger \*trig\_info

要注册的触发器

#### void iio\_trigger\_unregister(struct iio\_trigger \*trig\_info)

从核心注销触发器

**参数**

struct iio\_trigger \*trig\_info

要注销的触发器

#### int iio\_trigger\_set\_immutable(struct iio\_dev \*indio\_dev, struct iio\_trigger \*trig)

在目标上设置一个不可变触发器

**参数**

struct iio\_dev \*indio\_dev

包含设备的 IIO 设备结构。

struct iio\_trigger \*trig

要分配给设备的触发器

#### void iio\_trigger\_poll(struct iio\_trigger \*trig)

在触发器出现时调用

**参数**

struct iio\_trigger \*trig

发生的触发器

**说明**

通常在相关硬件中断处理程序中调用。

#### bool iio\_trigger\_using\_own(struct iio\_dev \*indio\_dev)

告诉我们自己是否使用自己的硬件触发器

**参数**

struct iio\_dev \*indio\_dev

要检查的设备

#### struct iio\_trigger \* devm\_iio\_trigger\_alloc(struct device \* dev, const char \* fmt, ...)

受资源管理的 iio\_trigger\_alloc()

**参数**

struct device \* dev

要为其分配 iio\_trigger 的设备

const char \* fmt

触发器名称格式。如果包括格式说明符，则在结果字符串中插入和格式化其后的附加参数，以替换它们的相应说明符。

...

可变参数

**说明**

管理的 iio\_trigger\_alloc。使用此函数分配的 iio\_trigger 将在驱动程序分离时自动释放。

如果需要单独释放使用此函数分配的 iio\_trigger，则必须使用 devm\_iio\_trigger\_free()。

**返回**

成功时分配的 iio\_trigger 的指针，失败时为 NULL。

#### void devm\_iio\_trigger\_free(struct device \* dev, struct iio\_trigger \* iio\_trig)

资源管理的iio\_trigger\_free（）

**参数**

struct device \* dev

此iio\_dev所属的设备

struct iio\_trigger \* iio\_trig

与设备关联的iio\_trigger

**说明**

释放使用devm\_iio\_trigger\_alloc（）分配的iio\_trigger。

#### int devm\_iio\_trigger\_register（struct device \* dev，struct iio\_trigger \* trig\_info）

资源管理iio\_trigger\_register（）

**参数**

struct device \* dev

为其分配此触发器的设备

struct iio\_trigger \* trig\_info

要注册的触发器

**说明**

管理iio\_trigger\_register（）。使用此函数注册的IIO触发器在驱动程序取消加载时自动取消注册。此函数在内部调用iio\_trigger\_register（）。有关更多信息，请参见该功能。

**返回**

成功时为0，失败时为负错误号。

#### void devm\_iio\_trigger\_unregister（struct device \* dev，struct iio\_trigger \* trig\_info）

资源管理iio\_trigger\_unregister（）

**参数**

struct device \* dev

此iio\_trigger所属的设备

struct iio\_trigger \* trig\_info

与设备关联的触发器

**说明**

注销使用devm\_iio\_trigger\_register（）注册的触发器。

#### int iio\_trigger\_validate\_own\_device（struct iio\_trigger \* trig，struct iio\_dev \* indio\_dev）

检查触发器和IIO设备是否属于同一设备

**参数**

struct iio\_trigger \* trig

要检查的IIO触发器

struct iio\_dev \* indio\_dev

要检查的IIO设备

**说明**

此函数可用作只能附加到其自己设备的触发器的validate\_device回调。

**返回**

如果触发器和IIO设备都属于同一个设备，则为0，否则为-EINVAL。

## 触发的缓冲区

既然我们知道了缓冲区和触发器是什么，让我们看看它们如何一起工作。

## IIO触发缓冲区设置

iio\_triggered\_buffer\_setup（） - 设置触发缓冲区和轮询函数

iio\_triggered\_buffer\_cleanup（） - 释放iio\_triggered\_buffer\_setup（）分配的资源

struct iio\_buffer\_setup\_ops - 缓冲区设置相关回调

典型的触发缓冲区设置如下：

const struct iio\_buffer\_setup\_ops sensor\_buffer\_setup\_ops = {

.preenable = sensor\_buffer\_preenable,

.postenable = sensor\_buffer\_postenable,

.postdisable = sensor\_buffer\_postdisable,

.predisable = sensor\_buffer\_predisable,

};

irqreturn\_t sensor\_iio\_pollfunc(int irq, void \*p)

{

pf->timestamp = iio\_get\_time\_ns((struct indio\_dev \*)p);

return IRQ\_WAKE\_THREAD;

}

irqreturn\_t sensor\_trigger\_handler(int irq, void \*p)

{

u16 buf[8];

int i = 0;

/\* read data for each active channel \*/

for\_each\_set\_bit(bit, active\_scan\_mask, masklength)

buf[i++] = sensor\_get\_data(bit)

iio\_push\_to\_buffers\_with\_timestamp(indio\_dev, buf, timestamp);

iio\_trigger\_notify\_done(trigger);

return IRQ\_HANDLED;

}

/\* setup triggered buffer, usually in probe function \*/

iio\_triggered\_buffer\_setup(indio\_dev, sensor\_iio\_polfunc,

sensor\_trigger\_handler,

sensor\_buffer\_setup\_ops);

要注意的重要事项是：

iio\_buffer\_setup\_ops，调用在缓冲区配置序列中预定义点的缓冲区设置功能（例如，在启用之前，禁用之后）。如果未指定，则IIO核心使用默认的iio\_triggered\_buffer\_setup\_ops。

sensor\_iio\_pollfunc，将用作轮询函数顶部的函数。它应该尽可能少地进行处理，因为它在中断上下文中运行。最常见的操作是记录当前时间戳，出于这个原因，可以使用IIO核心定义的iio\_pollfunc\_store\_time()函数。

sensor\_trigger\_handler，将用作轮询函数的底部函数。这在内核线程的上下文中运行，所有处理都在这里进行。它通常从设备中读取数据并将其与在顶部记录的时间戳一起存储在内部缓冲区中。

### 更多细节

#### int iio\_triggered\_buffer\_setup（struct iio\_dev \* indio\_dev，irqreturn\_t（\* h）（int irq，void \* p，irqreturn\_t（\* thread）（int irq，void \* p， const struct iio\_buffer\_setup\_ops \* setup\_ops）

设置触发缓冲区和轮询函数

**参数**

struct iio\_dev \* indio\_dev

IIO设备struct

irqreturn\_t（\*）（int irq，void \* p） h

将用作轮询函数顶部的函数

irqreturn\_t（\*）（int irq，void \* p） thread

将用作轮询函数底部的函数

const struct iio\_buffer\_setup\_ops \* setup\_ops

用于此设备的缓冲区设置功能。如果为NULL，则使用触发缓冲区的默认设置功能。

**说明**

此函数结合了在设置触发缓冲区时通常执行的一些常见任务。它将分配缓冲区和轮询函数。

在调用此函数之前，indio\_devstruct 应该已经完全初始化，但尚未注册。实际上，这意味着应该在iio\_device\_register（）右侧调用此函数。

要释放此函数分配的资源，请调用iio\_triggered\_buffer\_cleanup（）。

#### void iio\_triggered\_buffer\_cleanup（struct iio\_dev \* indio\_dev）

释放iio\_triggered\_buffer\_setup\_ext（）分配的资源

**参数**

struct iio\_dev \* indio\_dev

IIO设备struct

## 硬件消费者

IIO设备可以直接连接到硬件中的另一个设备。在这种情况下，IIO提供程序和IIO消费者之间的缓冲区由硬件处理。工业I / O HW消费者提供了一种无需软件缓冲区来绑定这些IIO设备的方法。实现可在drivers / iio / buffer / hw-consumer.c中找到。

struct iio\_hw\_consumer-硬件消费者结构

iio\_hw\_consumer\_alloc() — 分配IIO硬件消费者

iio\_hw\_consumer\_free() — 释放IIO硬件消费者

iio\_hw\_consumer\_enable() — 启用IIO硬件消费者

iio\_hw\_consumer\_disable() — 禁用IIO硬件消费者

### 硬件消费者设置

作为标准的IIO设备，实现基于IIO提供者/消费者。一个典型的IIO HW消费者设置如下:

static struct iio\_hw\_consumer \*hwc;

static const struct iio\_info adc\_info = {

.read\_raw = adc\_read\_raw,

};

static int adc\_read\_raw(struct iio\_dev \*indio\_dev,

struct iio\_chan\_spec const \*chan, int \*val,

int \*val2, long mask)

{

ret = iio\_hw\_consumer\_enable(hwc);

/\* Acquire data \*/

ret = iio\_hw\_consumer\_disable(hwc);

}

static int adc\_probe(struct platform\_device \*pdev)

{

hwc = devm\_iio\_hw\_consumer\_alloc(&iio->dev);

}

### 更多细节

#### struct iio\_hw\_consumer \*iio\_hw\_consumer\_alloc(struct device \*dev)

分配IIO硬件消费者

**参数**

struct device \*dev

指向消费者设备的指针。

**说明**

成功时返回有效的iio\_hw\_consumer，失败则返回ERR\_PTR()。

#### void iio\_hw\_consumer\_free(struct iio\_hw\_consumer \*hwc)

释放IIO硬件消费者

**参数**

struct iio\_hw\_consumer \*hwc

要释放的硬件消费者。

#### struct iio\_hw\_consumer \*devm\_iio\_hw\_consumer\_alloc(struct device \*dev)

受管理的iio\_hw\_consumer\_alloc()

**参数**

struct device \*dev

指向消费者设备的指针。

**说明**

管理的iio\_hw\_consumer\_alloc。用此函数分配的iio\_hw\_consumer将在驱动程序分离时自动释放。

成功时返回已分配的iio\_hw\_consumer指针，失败则返回NULL。

#### void devm\_iio\_hw\_consumer\_free(struct device \*dev, struct iio\_hw\_consumer \*hwc)

受管理的iio\_hw\_consumer\_free()

**参数**

struct device \*dev

指向消费者设备的指针。

struct iio\_hw\_consumer \*hwc

要释放的iio\_hw\_consumer。

**说明**

释放由devm\_iio\_hw\_consumer\_alloc()分配的iio\_hw\_consumer。

#### int iio\_hw\_consumer\_enable(struct iio\_hw\_consumer \*hwc)

启用IIO硬件消费者

**参数**

struct iio\_hw\_consumer \*hwc

要启用的iio\_hw\_consumer。

**说明**

成功时返回0。

#### void iio\_hw\_consumer\_disable(struct iio\_hw\_consumer \*hwc)

禁用IIO硬件消费者

**参数**

struct iio\_hw\_consumer \*hwc

要禁用的iio\_hw\_consumer。

# 输入子系统

## 输入核心

### struct input\_value

输入值表示

**定义**

struct input\_value {

\_\_u16 type;

\_\_u16 code;

\_\_s32 value;

};

**成员**

type

值类型（EV\_KEY，EV\_ABS等）

code

值代码

value

值

### struct input\_dev

表示输入设备

**定义**

struct input\_dev {

const char \*name;

const char \*phys;

const char \*uniq;

struct input\_id id;

unsigned long propbit[BITS\_TO\_LONGS(INPUT\_PROP\_CNT)];

unsigned long evbit[BITS\_TO\_LONGS(EV\_CNT)];

unsigned long keybit[BITS\_TO\_LONGS(KEY\_CNT)];

unsigned long relbit[BITS\_TO\_LONGS(REL\_CNT)];

unsigned long absbit[BITS\_TO\_LONGS(ABS\_CNT)];

unsigned long mscbit[BITS\_TO\_LONGS(MSC\_CNT)];

unsigned long ledbit[BITS\_TO\_LONGS(LED\_CNT)];

unsigned long sndbit[BITS\_TO\_LONGS(SND\_CNT)];

unsigned long ffbit[BITS\_TO\_LONGS(FF\_CNT)];

unsigned long swbit[BITS\_TO\_LONGS(SW\_CNT)];

unsigned int hint\_events\_per\_packet;

unsigned int keycodemax;

unsigned int keycodesize;

void \*keycode;

int (\*setkeycode)(struct input\_dev \*dev,const struct input\_keymap\_entry \*ke, unsigned int \*old\_keycode);

int (\*getkeycode)(struct input\_dev \*dev, struct input\_keymap\_entry \*ke);

struct ff\_device \*ff;

unsigned int repeat\_key;

struct timer\_list timer;

int rep[REP\_CNT];

struct input\_mt \*mt;

struct input\_absinfo \*absinfo;

unsigned long key[BITS\_TO\_LONGS(KEY\_CNT)];

unsigned long led[BITS\_TO\_LONGS(LED\_CNT)];

unsigned long snd[BITS\_TO\_LONGS(SND\_CNT)];

unsigned long sw[BITS\_TO\_LONGS(SW\_CNT)];

int (\*open)(struct input\_dev \*dev);

void (\*close)(struct input\_dev \*dev);

int (\*flush)(struct input\_dev \*dev, struct file \*file);

int (\*event)(struct input\_dev \*dev, unsigned int type, unsigned int code, int value);

struct input\_handle \_\_rcu \*grab;

spinlock\_t event\_lock;

struct mutex mutex;

unsigned int users;

bool going\_away;

struct device dev;

struct list\_head h\_list;

struct list\_head node;

unsigned int num\_vals;

unsigned int max\_vals;

struct input\_value \*vals;

bool devres\_managed;

};

**成员**

name

设备名称

phys

系统层次结构中设备的物理路径

uniq

设备的唯一标识代码（如果设备具有此代码）

id

设备的ID（struct input\_id）

propbit

设备属性和怪癖的位图

evbit

设备支持的事件类型的位图（EV\_KEY，EV\_REL等）

keybit

此设备具有的按键/按钮的位图

relbit

设备的相对轴的位图

absbit

设备的绝对轴的位图

mscbit

设备支持的其他事件的位图

ledbit

位于设备上的LED的位图

sndbit

设备支持的声音效果的位图

ffbit

设备支持的力反馈效果的位图

swbit

设备上存在的开关的位图

hint\_events\_per\_packet

设备在一个数据包中生成的事件的平均数（在EV\_SYN/SYN\_REPORT事件之间）。事件处理程序使用此值来估计需要包含事件的缓冲区的大小。

keycodemax

键码表的大小

keycodesize

键码表元素的大小

keycode

此设备的扫描码到键码的映射

setkeycode

可选的方法来更改当前键映射，用于实现稀疏键映射。如果未提供默认机制，则使用默认机制。此方法在持有event\_lock的同时被调用，因此不得睡眠

getkeycode

可选的传统方法，用于检索当前键映射。

ff

如果设备支持力反馈效果，则与设备关联的力反馈结构

repeat\_key

存储上次按下的键的键码；用于实现软件自动重复

timer

软件自动重复计时器

rep

用于自动重复参数（延迟、速率）的当前值

mt

指向多点触控状态的指针

absinfo

包含有关绝对轴（当前值、最小值、最大值、平坦度、模糊度、分辨率）的信息的struct input\_absinfo数组

key

反映设备的按键/按钮的当前状态

led

反映设备 LED 的当前状态

snd

反映声音效果的当前状态

sw

反映设备开关的当前状态

open

当第一个用户调用 input\_open\_device() 时调用此方法。驱动程序必须准备设备以开始生成事件（启动轮询线程、请求IRQ、提交URB等）。

close

当最后一个用户调用 input\_close\_device() 时调用此方法。

flush

清除设备。通常用于在从设备断开连接时清除加载到设备中的力反馈效果

event

事件处理程序，用于发送到设备的事件，如 EV\_LED 或 EV\_SND。设备应执行请求的动作（开启 LED、播放声音等）。调用受 event\_lock 保护，不得睡眠

grab

当前拥有设备的输入句柄（通过 EVIOCGRAB ioctl）。当句柄抓取设备时，它成为来自设备的所有输入事件的唯一接收者

event\_lock

在 input\_event() 中，当输入核心接收和处理设备的新事件时，将使用此自旋锁锁定代码。在设备已经在输入核心注册后，访问和/或修改设备的参数（例如 keymap 或 absmin、absmax、absfuzz 等）的代码必须获取此锁

mutex

序列化对 open()、close() 和 flush() 方法的调用

users

存储打开此设备的用户（输入处理程序）数量。它由 input\_open\_device() 和 input\_close\_device() 使用，以确保只有在第一个用户打开设备时才调用 dev->open()，而在最后一个用户关闭设备时调用 dev->close()

going\_away

标记正在注销的设备，并导致 input\_open\_device\*() 返回 -ENODEV。

dev

此设备的驱动程序模型视图

h\_list

与该设备关联的输入句柄列表。在访问列表时，必须使用 dev->mutex

node

用于将设备放置到 input\_dev\_list 上

num\_vals

当前帧中排队的值的数量

max\_vals

在一帧中排队的值的最大数量

vals

在当前帧中排队的值的数组

devres\_managed

指示设备是使用 devres 框架进行管理的，并且无需显式注销或释放。

### struct input\_handler

实现输入设备的接口之一

**定义**

struct input\_handler {

void \*private;

void (\*event)(struct input\_handle \*handle, unsigned int type, unsigned int code, int value);

void (\*events)(struct input\_handle \*handle, const struct input\_value \*vals, unsigned int count);

bool (\*filter)(struct input\_handle \*handle, unsigned int type, unsigned int code, int value);

bool (\*match)(struct input\_handler \*handler, struct input\_dev \*dev);

int (\*connect)(struct input\_handler \*handler, struct input\_dev \*dev, const struct input\_device\_id \*id);

void (\*disconnect)(struct input\_handle \*handle);

void (\*start)(struct input\_handle \*handle);

bool legacy\_minors;

int minor;

const char \*name;

const struct input\_device\_id \*id\_table;

struct list\_head h\_list;

struct list\_head node;

};

**成员**

private

驱动程序特定的数据

event

事件处理程序。输入核心在中断禁用和 dev->event\_lock 自旋锁保持的情况下调用此方法，因此它可能不会睡眠

events

事件序列处理程序。输入核心在中断禁用和 dev->event\_lock 自旋锁保持的情况下调用此方法，因此它可能不会睡眠

filter

类似于 event；将普通事件处理程序与“过滤器”分开。

match

在将设备的 ID 与处理程序的 ID 表进行比较后调用，以在设备和处理程序之间执行细粒度匹配

connect

将处理程序附加到输入设备时调用

disconnect

从输入设备断开连接处理程序

start

为给定的输入句柄启动处理程序。此函数在调用 connect() 方法之后由输入核心立即调用，以及处理“抓取”设备的进程释放它时调用

legacy\_minors

由使用传统次要值范围的驱动程序设置为 true

minor

该驱动程序可以提供的 32 个传统次要值范围的开始

name

处理程序的名称，将显示在 /proc/bus/input/handlers 中

id\_table

可以处理该驱动程序的 input\_device\_ids 表的指针

h\_list

与处理程序关联的输入句柄列表

node

用于将驱动程序放置到 input\_handler\_list 上

**说明**

输入处理程序附加到输入设备并创建输入句柄。在同一时间可能有几个处理程序附加到同一输入设备上。它们都将获得由设备生成的输入事件的副本。

相同的结构用于实现输入过滤器。输入核心允许先运行过滤器，如果任何过滤器表明应过滤事件（通过从其 filter() 方法返回 true），则不会将事件传递给常规处理程序。

请注意，输入核心序列化对 connect() 和 disconnect() 方法的调用。

### struct input\_handle

将输入设备与输入处理程序链接起来

**定义**

ct input\_handle {

void \*private;

int open;

const char \*name;

struct input\_dev \*dev;

struct input\_handler \*handler;

struct list\_head d\_node;

struct list\_head h\_node;

};

**成员**

private

特定于处理程序的数据

open

显示句柄是否“打开”的计数器，即应该从其设备传递事件

name

由创建它的处理程序赋予句柄的名称

dev

手柄连接到的输入设备

handler

通过此句柄与设备一起工作的处理程序

d\_node

用于将句柄放在设备的附加句柄列表中

h\_node

用于将句柄放在处理程序的句柄列表中，从中获取事件

### void input\_set\_events\_per\_packet(struct [input\_dev](https://docs.kernel.org/driver-api/input.html" \l "c.input_dev" \o "input_dev) \*dev, int n\_events)

告诉处理程序有关驱动程序事件率

**参数**

struct input\_dev \* dev

驱动程序使用的输入设备

int n\_events

调用之间的平均事件数input\_sync()

**说明**

如果从设备发送的事件率异常大，请使用此函数设置预期事件率。这将允许处理程序为事件流设置适当的缓冲区大小，以最大限度地减少信息丢失。

### struct ff\_device

输入设备的力反馈部分

**定义**

struct ff\_device {

int (\*upload)(struct input\_dev \*dev, struct ff\_effect \*effect, struct ff\_effect \*old);

int (\*erase)(struct input\_dev \*dev, int effect\_id);

int (\*playback)(struct input\_dev \*dev, int effect\_id, int value);

void (\*set\_gain)(struct input\_dev \*dev, u16 gain);

void (\*set\_autocenter)(struct input\_dev \*dev, u16 magnitude);

void (\*destroy)(struct ff\_device \*);

void \*private;

unsigned long ffbit[BITS\_TO\_LONGS(FF\_CNT)];

struct mutex mutex;

int max\_effects;

struct ff\_effect \*effects;

struct file \*effect\_owners[];

};

**成员**

upload

调用以将新效果上传到设备

erase

调用以从设备中删除效果

playback

调用请求设备开始播放指定效果

set\_gain

调用设置指定增益

set\_autocenter

调用自动居中设备

destroy

当父输入设备被销毁时由输入核心调用

private

驱动程序特定的数据，将被自动释放

ffbit

设备真正支持的力反馈功能的位图（不像 input\_dev->ffbit 中的那样模拟）

mutex

用于序列化对设备的访问的互斥体

max\_effects

设备支持的最大效果数

effects

指向当前加载到设备中的效果数组的指针

effect\_owners

一系列效果所有者；当拥有效果的文件句柄关闭时，效果会自动删除

**说明**

每个力反馈设备都必须实现upload()和playback() 方法；erase()是可选的。set\_gain()并且set\_autocenter()仅在驱动程序设置了 FF\_GAIN 和 FF\_AUTOCENTER 位时才需要实施。

请注意playback()，set\_gain()和set\_autocenter()是在 dev->event\_lock 自旋锁保持和中断关闭的情况下调用的，因此可能不会休眠。

### void input\_event(struct [input\_dev](https://docs.kernel.org/driver-api/input.html" \l "c.input_dev" \o "input_dev) \*dev, unsigned int type, unsigned int code, int value)

报告新的输入事件

**参数**

struct input\_dev \* dev

产生事件的设备

unsigned int type

事件的类型

unsigned int code

事件代码

int value

事件的价值

**说明**

实现各种输入设备的驱动程序应该使用此函数来报告输入事件。另见[input\_inject\_event()](https://www.kernel.org/doc/html/v4.19/driver-api/input.html" \l "c.input_inject_event" \o "输入_注入_事件)。

**注意**

[input\_event()](https://www.kernel.org/doc/html/v4.19/driver-api/input.html" \l "c.input_event" \o "输入事件)可以在输入设备分配给 之后安全地使用[input\_allocate\_device()](https://www.kernel.org/doc/html/v4.19/driver-api/input.html" \l "c.input_allocate_device" \o "输入分配设备)，甚至在它注册之前就可以安全使用[input\_register\_device()](https://www.kernel.org/doc/html/v4.19/driver-api/input.html" \l "c.input_register_device" \o "输入寄存器设备)，但事件不会到达任何输入处理程序。这种早期调用[input\_event()](https://www.kernel.org/doc/html/v4.19/driver-api/input.html" \l "c.input_event" \o "输入事件)可用于“播种”开关的初始状态或绝对轴的初始位置等。

### void input\_inject\_event(struct [input\_handle](https://docs.kernel.org/driver-api/input.html" \l "c.input_handle" \o "input_handle) \*handle, unsigned int type, unsigned int code, int value)

从输入处理程序发送输入事件

**参数**

struct input\_handle \* handle

通过输入句柄发送事件

unsigned int type

事件的类型

unsigned int code

事件代码

int value

事件的价值

**说明**

类似于[input\_event()](https://www.kernel.org/doc/html/v4.19/driver-api/input.html" \l "c.input_event" \o "输入事件)但如果设备被“抢夺”并且处理注入事件不是拥有该设备的事件，则会忽略事件。

### void input\_alloc\_absinfo(struct [input\_dev](https://docs.kernel.org/driver-api/input.html" \l "c.input_dev" \o "input_dev) \*dev)

分配 input\_absinfo 结构数组

**参数**

struct input\_dev \* dev

输入设备发出绝对事件

**说明**

如果调用者要求的 absinfo 结构已经分配，​​这个函数将不会做任何事情。

### int input\_grab\_device(struct [input\_handle](https://docs.kernel.org/driver-api/input.html" \l "c.input_handle" \o "input_handle) \*handle)

专用抓取器

**参数**

struct input\_handle \* handle

想要拥有设备的输入句柄

**说明**

当设备被输入句柄抓取时，设备生成的所有事件都只传递给这个句柄。当设备被抓取时，其他输入句柄注入的事件也会被忽略。

### **v**oid input\_release\_device(struct [input\_handle](https://docs.kernel.org/driver-api/input.html" \l "c.input_handle" \o "input_handle) \*handle)

释放先前抓取的设备。

**参数**

struct input\_handle \*handle

拥有该设备的输入控制器。

**说明**

释放先前抓取的设备，以便其他输入控制器可以开始接收输入事件。在释放后，所有连接到该设备的处理程序都会调用其start()方法，以便它们可以将设备状态与系统其余部分同步。

### int input\_open\_device(struct input\_handle \*handle)

打开输入设备。

**参数**

struct input\_handle \*handle

通过该句柄访问设备。

**说明**

当输入控制器想要从给定的输入设备开始接收事件时，应调用此函数。

### void input\_close\_device(struct input\_handle \*handle)

关闭输入设备。

**参数**

struct input\_handle \*handle

通过该句柄访问设备。

**说明**

当输入控制器想要停止从给定的输入设备接收事件时，应调用此函数。

### int input\_scancode\_to\_scalar(const struct input\_keymap\_entry \*ke, unsigned int \*scancode)

将struct input\_keymap\_entry中的扫描码转换成标量形式。

**参数**

const struct input\_keymap\_entry \*ke

包含要转换的扫描码的按键映射表条目。

unsigned int \*scancode

所转换的扫描码应存储的位置的指针。

**说明**

此函数用于将存储在struct keymap\_entry中的扫描码转换成传统按键映射处理方法所理解的标量形式。这些方法期望将扫描码表示为“unsigned int”。

### int input\_get\_keycode(struct input\_dev \*dev, struct input\_keymap\_entry \*ke)

检索为给定扫描码映射的按键码

**参数**

struct input\_dev \*dev

查询其键映射的输入设备

struct input\_keymap\_entry \*ke

键映射表条目

**说明**

任何有兴趣检索当前键映射的人都应调用此函数。目前evdev处理程序正在使用它。

### int input\_set\_keycode(struct input\_dev \*dev, const struct input\_keymap\_entry \*ke)

为给定扫描码分配按键码

**参数**

struct input\_dev \*dev

要更新其键映射的输入设备

const struct input\_keymap\_entry \*ke

新键映射表条目

**说明**

任何需要更新当前键映射的人都应调用此函数。目前键盘和evdev处理程序正在使用它。

### void input\_reset\_device(struct input\_dev \*dev)

重置/恢复输入设备的状态

**参数**

struct input\_dev \*dev

需要重置状态的输入设备

**说明**

此函数尝试重置打开的输入设备的状态，并将内部状态与硬件状态同步。我们将所有键标记为已释放，恢复LED状态，重复速率等。

### struct input\_dev \*input\_allocate\_device(void)

为新的输入设备分配内存。

**参数**

void

无需参数

**说明**

返回准备好的struct input\_dev或NULL。

**注意**

使用input\_free\_device()来释放尚未注册的设备；对于已注册的设备应使用input\_unregister\_device()。

### struct input\_dev \*devm\_input\_allocate\_device(struct device \*dev)

分配受管理的输入设备。

**参数**

struct device \*dev

拥有正在创建的输入设备的设备

**说明**

返回准备好的struct input\_dev或NULL。

管理的输入设备不需要显式注销或释放，因为当所有者设备取消与其驱动程序的绑定（或绑定失败）时，它将自动完成。一旦分配了托管输入设备，它就可以像常规输入设备一样进行设置和注册。如果您需要它们，没有特殊的devm\_input\_device\_[un]register()变体，常规的变体可以处理托管和未托管的设备。但是，在大多数情况下，无需显式注销或释放托管的输入设备。

**注意**

所有者设备被设置为输入设备的父级，用户不应该覆盖它。

### void input\_free\_device(struct input\_dev \*dev)

释放输入\_dev结构占用的内存。

**参数**

struct input\_dev \*dev

要释放的输入设备

**说明**

如果尚未调用input\_register\_device()或者调用失败，则应仅使用此函数。一旦设备被注册使用input\_unregister\_device()，一旦对设备的最后引用被删除，内存将被释放。

设备应由input\_allocate\_device()分配。

**注意**

如果存在对输入设备的引用，则在最后一个引用被删除之前，内存不会被释放。

### void input\_set\_capability(struct input\_dev \*dev, unsigned int type, unsigned int code)

将设备标记为能够发出或接受某个事件。

**参数**

struct input\_dev \*dev

能够发出或接受事件的设备

unsigned int type

事件的类型（EV\_KEY、EV\_REL等）

unsigned int code

事件代码

**说明**

除了在适当的能力位图中设置相应的位之外，该函数还会调整dev->evbit。

### void input\_enable\_softrepeat(struct input\_dev \*dev, int delay, int period)

启用软件自动重复。

**参数**

struct input\_dev \*dev

输入设备

int delay

重复延迟时间

int period

重复周期时间

**说明**

在输入设备上启用软件自动重复。

### int input\_register\_device(struct input\_dev \*dev)

向输入核心注册设备

**参数**

struct input\_dev \*dev

要注册的设备

**说明**

此函数将设备注册到输入核心。在注册之前，设备必须使用input\_allocate\_device()进行分配，并设置好所有的功能。如果函数失败，则必须使用input\_free\_device()释放设备。设备成功注册后，可以使用input\_unregister\_device()注销设备；此时不应调用input\_free\_device()。

请注意，此函数还用于注册托管输入设备（使用devm\_input\_allocate\_device()分配的设备）。这些托管的输入设备不需要明确注销或释放，其拆除是由devres基础设施控制的。还值得注意的是，托管输入设备的拆除在内部是一个两步过程先注销已注册的托管输入设备，但仍然保留在内存中，仍然可以处理input\_event()调用（尽管事件不会被传递到任何地方）。设备的释放将在以设备分配为基础的devres堆栈解开时进行。

### void input\_unregister\_device(struct input\_dev \*dev)

注销先前注册的设备

**参数**

struct input\_dev \*dev

要注销的设备

**说明**

此函数注销输入设备。设备注销后，调用者不应尝试访问它，因为它可能随时被释放。

### int input\_register\_handler(struct input\_handler \*handler)

注册新的输入处理程序

**参数**

struct input\_handler \*handler

要注册的处理程序

**说明**

此函数为系统中的输入设备注册新的输入处理程序（接口），并将其连接到与处理程序兼容的所有输入设备上。

### void input\_unregister\_handler(struct input\_handler \*handler)

注销输入处理程序

**参数**

struct input\_handler \*handler

要注销的处理程序

**说明**

此函数从其输入设备中断开处理程序，并将其从已知处理程序列表中删除。

### int input\_handler\_for\_each\_handle(struct input\_handler \*handler, void \*data, int (\*fn)(struct input\_handle\*, void\*))

处理句柄迭代器

**参数**

struct input\_handler \*handler

要迭代的输入处理程序

void \*data

回调函数的数据

int (\*fn)(struct input\_handle \*, void \*)

要为每个句柄调用的函数

**说明**

迭代总线的设备列表，并为每个设备调用fn，传递数据并在fn返回非零值时停止。该函数使用RCU来遍历列表，因此可能在原子环境中使用。fn回调从RCU临界节中调用，因此不能休眠。

### int input\_register\_handle(struct input\_handle \*handle)

注册新的输入句柄

**参数**

struct input\_handle \*handle

要注册的句柄

**说明**

此函数将新的输入句柄放入设备和处理程序的列表中，以便在使用input\_open\_device()打开后可以通过它流动事件。

这个函数应该从处理器的connect()方法中调用。

### void input\_unregister\_handle(struct input\_handle \*handle)

注销输入句柄

**参数**

struct input\_handle \*handle

要注销的句柄

**说明**

此函数从设备和处理程序的列表中删除输入句柄。

这个函数应该从处理器的disconnect()方法中调用。

### int input\_get\_new\_minor(int legacy\_base, unsigned int legacy\_num, bool allow\_dynamic)

分配新的输入次要编号

**参数**

int legacy\_base

要搜索的旧范围的起始位置

unsigned int legacy\_num

旧范围的大小

bool allow\_dynamic

是否可以从动态范围中获取ID

**说明**

此函数为从输入主要命名空间分配新设备次要编号。调用者可以通过指定legacy\_base和legacy\_num参数请求传统的次要编号，以及在传统范围内没有空闲ID时是否可以从动态范围中分配ID。

### void input\_free\_minor(unsigned int minor)

释放以前分配的次要编号

**参数**

unsigned int minor

要释放的次要编号

**说明**

此函数释放以前分配的输入次要编号，以便稍后可以重用它。

### int input\_ff\_upload(struct input\_dev \*dev, struct ff\_effect \*effect, struct file \*file)

将效果上传到力反馈设备

**参数**

struct input\_dev \*dev

输入设备

struct ff\_effect \*effect

要上传的效果

struct file \*file

效果的所有者

### int input\_ff\_erase(struct input\_dev \*dev, int effect\_id, struct file \*file)

从设备中删除力反馈效果

**参数**

struct input\_dev \*dev

要从中删除效果的输入设备

int effect\_id

要删除的效果的id

struct file \*file

请求的所有者

**说明**

此函数从指定的设备中删除力反馈效果。只有在使用相同的文件句柄上传的效果才会被删除。

### int input\_ff\_event(struct input\_dev \*dev, unsigned int type, unsigned int code, int value)

力反馈事件的通用处理程序

**参数**

struct input\_dev \*dev

要将效果发送到的输入设备

unsigned int type

事件类型（除EV\_FF外的任何东西都会被忽略）

unsigned int code

事件代码

int value

事件值

### int input\_ff\_create（struct input\_dev \*dev，unsigned int max\_effects）

创建力反馈设备

**参数**

struct input\_dev \*dev

支持力反馈的输入设备

unsigned int max\_effects

设备支持的最大效果数

**说明**

此函数为输入设备的力反馈部分分配所有必要的内存，并安装所有默认处理程序。在调用此功能之前，dev->ffbit应该已经设置好。创建ff设备后，您需要设置其上传、擦除、播放和其他处理程序，然后才能注册输入设备。

### void input\_ff\_destroy（struct input\_dev \*dev）

释放输入设备的力反馈部分

**参数**

struct input\_dev \*dev

支持力回馈的输入设备

**说明**

此功能仅在错误路径中需要，因为当设备被销毁时，输入核心将自动释放力回馈结构。

### int input\_ff\_create\_memless（struct input\_dev \*dev、void \*data，int（\*play\_effect）（struct input\_dev\*，void\*，struct ff\_effect\*））

创建无内存力反馈设备

**参数**

struct input\_dev \*dev

支持力反馈的输入设备

void \*data

传递到play\_effect的驱动程序特定数据

int（\*play\_effect）（struct input\_dev\*，void\*，struct ff\_effect\*）

驱动程序特定的播放FF效果的方法

## 多点触控库

### struct input\_mt\_slot

表示输入MT插槽的状态

**定义**

struct input\_mt\_slot {

int abs[ABS\_MT\_LAST - ABS\_MT\_FIRST + 1];

unsigned int frame;

unsigned int key;

};

**成员**

abs

在此插槽中保存ABS\_MT轴的当前值

frame

调用input\_mt\_report\_slot\_state（）的最后一帧

key

此插槽的可选驱动程序指定

### struct input\_mt

跟踪联系的状态

**定义**

struct input\_mt {

int trkid;

int num\_slots;

int slot;

unsigned int flags;

unsigned int frame;

int \*red;

struct input\_mt\_slot slots[];

};

**成员**

trkid

为下一个联系存储MT跟踪ID

num\_slots

设备使用的MT插槽数

slot

当前正在传输的MT槽

flags

输入\_mt操作标志

frame

每次调用input\_mt\_sync\_frame（）时增加

red

内核内跟踪的降低成本矩阵

slots

保持跟踪联系当前值的插槽数组

### struct input\_mt\_pos

联系位置

**定义**

struct input\_mt\_pos {

s16 x, y;

};

**成员**

x——水平坐标

y——垂直坐标

### int input\_mt\_init\_slots（struct input\_dev \*dev，unsigned int num\_slots，unsigned int flags）

初始化MT输入插槽

**参数**

struct input\_dev \*dev

支持MT事件和手指跟踪的输入设备

unsigned int num\_slots

设备使用的插槽数

unsigned int flags

在核心中处理的mt任务

**说明**

此函数为输入设备中的MT插槽处理分配所有必要的内存，准备ABS\_MT\_SLOT和ABS\_MT\_TRACKING\_ID事件供使用，并设置适当的缓冲区。根据设置的标志，它还执行指针模拟和帧同步。

可以重复调用。如果尝试重新初始化不同数目的插槽，返回-EINVAL。

### void input\_mt\_destroy\_slots（struct input\_dev \*dev）

释放输入设备的MT插槽

**参数**

struct input\_dev \*dev

具有已分配MT插槽的输入设备

**说明**

此功能仅在错误路径中需要，因为当设备被销毁时，输入核心将自动释放MT插槽。

### bool input\_mt\_report\_slot\_state（struct input\_dev \*dev，unsigned int tool\_type，bool active）

报告联系状态

**参数**

struct input\_dev \*dev

具有已分配MT插槽的输入设备

unsigned int tool\_type

在此插槽中使用的工具类型

布尔值活跃的

如果联系处于活动状态，则为true；否则为false

**说明**

通过ABS\_MT\_TRACKING\_ID报告联系，如果需要，还通过ABS\_MT\_TOOL\_TYPE报告。如果活动为true且槽当前处于非活动状态，或者更改了工具类型，则向插槽分配新的跟踪ID。仅在设置了相应的absbit字段时才报告工具类型。

如果联系处于活动状态，则返回true。

### void input\_mt\_report\_finger\_count(struct input\_dev \*dev，int count)

报告联系数

**参数**

struct input\_dev \*dev

具有已分配MT插槽的输入设备

int count

联系数

**说明**

通过BTN\_TOOL\_FINGER、BTN\_TOOL\_DOUBLETAP、BTN\_TOOL\_TRIPLETAP和BTN\_TOOL\_QUADTAP报告联系计数。

输入核心确保只有为此设备设置的KEY事件会产生输出。

### void input\_mt\_report\_pointer\_emulation(struct input\_dev \*dev，bool use\_count)

常见的指针模拟

**参数**

struct input\_dev \*dev

具有已分配MT插槽的输入设备

bool use\_count

将活动联系数报告为手指计数

**说明**

通过BTN\_TOUCH、ABS\_X、ABS\_Y和ABS\_PRESSURE执行传统的指针模拟。如果use\_count为真，则模拟触摸板手指计数。

输入核心确保只有为此设备设置的KEY和ABS轴会产生输出。

### void input\_mt\_drop\_unused(struct input\_dev \*dev)

未在此帧中看到的不活动插槽

**参数**

struct input\_dev \*dev

具有已分配MT插槽的输入设备

**说明**

举起自上次调用此函数以来未看到的所有插槽。

### void input\_mt\_sync\_frame(struct input\_dev \*dev)

同步mt帧

**参数**

struct input\_dev \*dev

具有已分配MT插槽的输入设备

**说明**

关闭帧并为新帧准备内部状态。根据标志，标记未使用的插槽为不活动并执行指针模拟。

### int input\_mt\_assign\_slots(struct input\_dev \*dev, int \*slots, const struct input\_mt\_pos \*pos, int num\_pos, int dmax)

执行最佳匹配分配

**参数**

struct input\_dev \*dev

带有已分配 MT 插槽的输入设备

int \*slots

要填充的插槽分配

const struct input\_mt\_pos \*pos

要匹配的位置数组

int num\_pos

位置数

int dmax

ABS\_MT\_POSITION 的最大位移（无限则为零）

**说明**

执行针对当前触点的最佳匹配，并返回插槽分配列表。新的触点分配到未使用的插槽上。

分配是平衡的，以使所有坐标位移低于欧式距离 dmax。如果找不到这样的分配，则将某些触点分配到未使用的插槽上。

成功返回零，否则返回负错误。

### int input\_mt\_get\_slot\_by\_key(struct input\_dev \*dev, int key)

返回与 key 匹配的插槽

**参数**

struct input\_dev \*dev

带有已分配 MT 插槽的输入设备

int key

所寻找插槽的键

**说明**

如果存在，则返回给定键的插槽，否则将键设置为第一个未使用的插槽并返回。

如果找不到可用插槽，则返回 -1。请注意，为了使此函数正常工作，必须在每个帧上调用 input\_mt\_sync\_frame()。

## 轮询输入设备

### struct input\_polled\_dev

简单轮询输入设备

**定义**

vstruct input\_polled\_dev {

void \*private;

void (\*open)(struct input\_polled\_dev \*dev);

void (\*close)(struct input\_polled\_dev \*dev);

void (\*poll)(struct input\_polled\_dev \*dev);

unsigned int poll\_interval;

unsigned int poll\_interval\_max;

unsigned int poll\_interval\_min;

struct input\_dev \*input;

};

**成员**

private

驱动程序数据的私有部分。

open

准备轮询输入设备的驱动程序提供的方法（启用设备并可能清除设备状态）。

close

轮询结束时调用的驱动程序提供的方法。用于将设备置于低功耗模式。

poll

轮询设备并发布输入事件的驱动程序提供的方法（必需）。

poll\_interval

指定 poll() 方法应调用的频率。如果在注册设备时没有覆盖，则默认为 500 毫秒。

poll\_interval\_max

指定轮询间隔的上限。默认为 poll\_interval 的初始值。

poll\_interval\_min

指定轮询间隔的下限。默认为 0。

input

与轮询设备相关联的输入设备结构。必须由驱动程序正确初始化（id、名称、物理、位数）。

**说明**

轮询输入设备提供了一个骨架，用于支持不引发中断但必须定期扫描或轮询以检测其状态变化的简单输入设备。

### struct input\_polled\_dev \* input\_allocate\_polled\_device(void)

分配轮询设备的内存

**参数**

void

无参数

**说明**

该函数为轮询设备和与该轮询设备相关联的输入设备分配内存。

### struct input\_polled\_dev \*devm\_input\_allocate\_polled\_device(struct device \*dev)

分配托管轮询设备

**参数**

struct device \*dev

拥有正在创建的轮询设备的设备

**说明**

返回准备好的 struct input\_polled\_dev 或 NULL。

托管轮询输入设备无需显式注销或释放，因为在所有者设备解除绑定其驱动程序（或绑定失败）时将自动完成。一旦分配了这样的托管轮询设备，它就可以像常规轮询输入设备一样进行设置和注册（使用 input\_register\_polled\_device() 函数）。

如果要手动注销和释放这些托管轮询设备，仍然可以通过调用 input\_unregister\_polled\_device() 和 input\_free\_polled\_device() 来完成，尽管这很少需要。

**注意**

所有者设备会设置为输入设备的父设备，用户不应覆盖它。

### void input\_free\_polled\_device(struct input\_polled\_dev \* dev)

释放分配给轮询设备的内存

**参数**

struct input\_polled\_dev \* dev

要释放的设备

**说明**

该函数释放为轮询设备分配的内存并丢弃对关联输入设备的引用。

### int input\_register\_polled\_device(struct input\_polled\_dev \* dev)

注册轮询设备

**参数**

struct input\_polled\_dev \* dev

要注册的设备

**说明**

该函数将之前初始化的轮询输入设备注册到输入层。应使用 input\_allocate\_polled\_device() 调用为设备分配。调用者还应设置 poll() 方法并设置相应的 input\_dev 结构的功能（id、name、phys、bits）。

### void input\_unregister\_polled\_device(struct input\_polled\_dev \* dev)

注销轮询设备

**参数**

struct input\_polled\_dev \* dev

要注销的设备

**说明**

该函数从输入层注销之前注册的轮询输入设备。轮询停止，设备可以通过调用 input\_free\_polled\_device() 自由释放。

## 矩阵键盘/键盘

### struct matrix\_keymap\_data

矩阵键盘的按键映射

**定义**

struct matrix\_keymap\_data {

const uint32\_t \*keymap;

unsigned int keymap\_size;

};

**成员**

keymap

指向用 KEY() 宏编码的 uint32 值数组表示按键映射的指针

keymap\_size

此键映射中的条目数量（已初始化）

**说明**

该结构应由平台代码使用，以向实现矩阵式键盘/键盘的驱动程序提供按键映射。

### struct matrix\_keypad\_platform\_data

平台依赖的键盘数据

**定义**

struct matrix\_keypad\_platform\_data {

const struct matrix\_keymap\_data \*keymap\_data;

const unsigned int \*row\_gpios;

const unsigned int \*col\_gpios;

unsigned int num\_row\_gpios;

unsigned int num\_col\_gpios;

unsigned int col\_scan\_delay\_us;

unsigned int debounce\_ms;

unsigned int clustered\_irq;

unsigned int clustered\_irq\_flags;

bool active\_low;

bool wakeup;

bool no\_autorepeat;

bool drive\_inactive\_cols;

};

**成员**

keymap\_data

指向 matrix\_keymap\_data 的指针

row\_gpios

指向表示行的 GPIO 号码数组的指针

col\_gpios

指向表示列的 GPIO 号码数组的指针

num\_row\_gpios

设备实际使用的行 GPIO 数量

num\_col\_gpios

设备实际使用的列 GPIO 数量

col\_scan\_delay\_us

在激活列 GPIO 后，需要等待的延迟时间，以微秒为单位

debounce\_ms

消抖间隔时间，以毫秒为单位

clustered\_irq

可以指定如果所有行/列 GPIO 的中断被捆绑到一个单一的 irq 上

clustered\_irq\_flags

捆绑 irq 所需的标志

active\_low

GPIO 极性

wakeup

控制设备是否应设置为唤醒源

no\_autorepeat

禁用键自动重复

drive\_inactive\_cols

在扫描期间驱动非活动列，而不是将它们设置为输入。

**说明**

该结构表示矩阵键盘驱动程序使用的特定于平台的数据，以执行适当的初始化。

## 稀疏密钥映射支持

### struct key\_entry

用于稀疏密钥映射中的键映射条目

**定义**

struct key\_entry {

int type;

u32 code;

union {

u16 keycode;

struct {

u8 code;

u8 value;

} sw;

};

};

**成员**

type

键输入的类型 ( KE\_KEY，KE\_SW，KE\_VSW，KE\_END); 驱动程序允许扩展该列表，添加自己的私有定义。

code

标识按钮/开关的设备特定数据

unnamed\_union

未命名联合

keycode

分配给键/按钮的 KEY\_\* 代码

sw.code

分配给开关的 SW\_\* 代码

sw.value

KE\_SW 开关被切换时应在输入事件中发送的值。KE\_VSW 开关忽略此字段，并期望驱动程序为事件提供值。

**说明**

该结构定义了稀疏密钥映射中的条目，用于某些输入设备，传统的基于表格的方法不适用。

### struct key\_entry \*sparse\_keymap\_entry\_from\_scancode(struct input\_dev \*dev, unsigned int code)

执行稀疏密钥映射查找

**参数**

struct input\_dev \*dev

使用稀疏密钥映射的输入设备

unsigned int code

扫描代码

**说明**

此函数用于在使用稀疏键映射的输入设备中执行结构 key\_entry 查找。

### struct key\_entry \*sparse\_keymap\_entry\_from\_keycode(struct input\_dev \*dev, unsigned int keycode)

执行稀疏密钥映射查找

**参数**

struct input\_dev \*dev

使用稀疏密钥映射的输入设备

unsigned int keycode

键代码

**说明**

此函数用于在使用稀疏键映射的输入设备中执行结构 key\_entry 查找。

### int sparse\_keymap\_setup(struct input\_dev \*dev, const struct key\_entry \*keymap, int (\*setup)(struct input\_dev\*, struct key\_entry\*))

为输入设备设置稀疏密钥映射

**参数**

struct input\_dev \*dev

输入设备

const struct key\_entry \*keymap

以 key\_entry 结构的数组形式结束的键映射

### void sparse\_keymap\_report\_entry(struct input\_dev \*dev, const struct key\_entry \*ke, unsigned int value, bool autorelease)

报告与给定键条目相应的事件

**参数**

struct input\_dev \*dev

要报告事件的输入设备

const struct key\_entry \*ke

说明事件的键条目

unsigned int value

应报告的值 (KE\_SW 条目忽略此字段)

bool autorelease

发出 KE\_KEY 条目的按下事件报告后是否应发出释放事件，忽略所有其他条目

**说明**

此函数用于报告给定结构 key\_entry 说明的输入事件。

### bool sparse\_keymap\_report\_event(struct input\_dev \*dev, unsigned int code, unsigned int value, bool autorelease)

报告与给定扫描代码相应的事件

**参数**

struct input\_dev \*dev

使用稀疏密钥映射的输入设备

unsigned int code

扫描代码

unsigned int value

应报告的值 (KE\_SW 条目忽略此字段)

bool autorelease

发出 KE\_KEY 条目的按下事件报告后是否应发出释放事件，忽略所有其他条目

**说明**

此函数用于执行键映射查找，并报告相应事件在使用稀疏密钥映射的输入设备中。如果查找成功，则返回 true，否则返回 false。

# Linux USB API

## Linux-USB主机端API

### Linux上的USB介绍

通用串行总线 (USB) 用于将主机 (例如 PC 或工作站) 连接到多个外围设备。USB 使用一个树形结构，主机作为根 (系统的主节点)，集线器作为内部节点，外围设备作为叶子 (从节点)。现代计算机支持几个这样的 USB 设备树，通常有几个 USB 3.0 (5 GBit/s) 或 USB 3.1 (10 GBit/s) 以及一些遗留的 USB 2.0 (480 MBit/s) 总线以备不时之需。

这种主/从不对称性是基于多种原因设计的，其中之一是易于使用。在类型 C 插头上，物理上不可能混淆上游和下游或者它们被内置到外围设备中。此外，主机软件不需要处理分布式自动配置，因为预先指定的主节点管理所有这些工作。

内核开发者在2.2内核系列早期添加了对Linux的USB支持，并自那以后不断发展。除了支持每一代新的USB，各种主机控制器也得到了支持，添加了新的外围驱动程序，并引入了用于延迟测量和改进电源管理的高级特性。Linux不仅可以在控制设备的主机上运行，还可以在USB设备内运行。但是，在这些外围设备内运行的USB设备驱动程序不做与在主机上运行的驱动程序相同的事情，因此它们被赋予了一个不同的名称小工具驱动程序。本文档不涉及小工具驱动程序。

### USB主机端API模型

用于USB设备的主机端驱动程序与“usbcore” API交互。其中有两个。其中之一适用于通用驱动程序（通过驱动程序框架公开），另一个适用于作为核心部分的驱动程序。这样的核心驱动程序包括集线器驱动程序（管理USB设备的树）和若干种不同类型的主机控制器驱动程序，用于控制单个总线。

USB驱动程序看到的设备模型相对复杂。

1. USB支持四种数据传输类型（控制、大容量、中断和等时）。其中两种（控制和大容量）使用可用带宽，而另外两种（中断和等时）被安排为提供保证带宽。
2. 设备说明模型包括每个设备的一个或多个配置，但一次仅活动一个。设备应该能够以低于其最高速度的速度运行，并可以提供显示其完全运行的最低速度的BOS说明符。
3. 从USB 3.0开始，配置具有一个或多个“功能”，它们提供常用功能，并为功率管理目的而分组。
4. 配置或功能具有一个或多个“接口”，每个接口可能具有“备用设置”。接口可以由USB“类”规范进行标准化，也可以特定于供应商或设备。
5. 实际上，USB设备驱动程序绑定到接口而不是设备。将它们视为“接口驱动程序”，尽管您可能不会看到许多区别很重要的设备。大多数USB设备都很简单，只有一个功能、一个配置、一个接口和一个备用设置。
6. 接口具有一个或多个“端点”，每个端点支持一种类型和数据传输方向，例如“大容量输出”或“中断输入”。整个配置在每个方向上最多可以有十六个端点，可以根据所有接口的需要分配。
7. USB上的数据传输是分组的；每个端点有一个最大数据包大小。驱动程序通常必须了解惯例，例如使用“短”（包括零长度）数据包标记大容量传输的结束。
8. Linux USB API支持同步调用用于控制和大容量消息。它还支持用于所有数据传输类型的异步调用，使用称为“URB”（USB请求块）的请求结构。

因此，暴露给设备驱动程序的USB核心API涵盖了相当广泛的领域。您可能需要查阅USB 3.0规范（可在www.usb.org上免费在线获取），以及类别或设备规范。

实际上触摸硬件（读/写寄存器、处理IRQ等）的唯一主机端驱动程序是HCDs。从理论上讲，所有HCD都通过同一个API提供相同的功能。实际上，这越来越成为事实，但仍然存在差异，尤其是在不太常见的控制器上的故障处理方面。不同的控制器不一定报告相同的故障方面，并且从故障（包括像解除URB一样的软件引起的故障）的恢复还不完全一致。设备驱动程序作者应该注意使用每个不同的主机控制器驱动程序进行断开测试（在设备处于活动状态时），以确保驱动程序没有自己的错误，并确保它们不依赖于某些特定HCD行为。

### USB标准类型

在include/uapi/linux/usb/ch9.h中，您将找到在USB规范第9章中定义的USB数据类型。这些数据类型在整个USB中使用，在API（包括这个主机端API、小工具API、USB字符设备和debugfs接口）中使用。

#### const char \*usb\_speed\_string(enum usb\_device\_speed speed)

返回人类可读名称的速度

**参数**

enum usb\_device\_speed speed

要返回人类可读名称的速度。如果它不是在usb\_device\_speedenum中定义的任何速度，则将返回USB\_SPEED\_UNKNOWN的字符串。

#### enum usb\_device\_speed usb\_get\_maximum\_speed(struct device \*dev)

获取给定USB控制器的最大请求速度。

**参数**

struct device \*dev

指向给定USB控制器设备的指针。

**说明**

该函数从属性“maximum-speed”中获取最大速度字符串，并返回相应的enumusb\_device\_speed。

#### const char \*usb\_state\_string(enum usb\_device\_state state)

返回状态的可读名称。

**参数**

enumusb\_device\_state state

要返回可读名称的状态。如果它不是usb\_device\_state\_stringenum中的状态设备之一，则将返回字符串UNKNOWN。

### 主机端数据类型和宏

主机端API为驱动程序暴露了几个层，其中一些比其他层更为必要。它们支持主机侧驱动程序和设备的生命周期模型，并支持通过usbcore将缓冲区传递到执行设备驱动程序的某个HCD。

#### struct usb\_host\_endpoint

主机端点说明符和队列

**定义**

struct usb\_host\_endpoint {

struct usb\_endpoint\_descriptor desc;

struct usb\_ss\_ep\_comp\_descriptor ss\_ep\_comp;

struct usb\_ssp\_isoc\_ep\_comp\_descriptor ssp\_isoc\_ep\_comp;

struct list\_head urb\_list;

void \*hcpriv;

struct ep\_device \*ep\_dev;

unsigned char \*extra;

int extralen;

int enabled;

int streams;

};

**成员**

desc

此端点的说明符，wMaxPacketSize以本机字节序

ss\_ep\_comp

此端点的超级速度补充说明符

ssp\_isoc\_ep\_comp

此端点的超级速度加等时同步补充说明符

urb\_list

排队到此端点的URB；由usbcore维护

hcpriv

供HCD使用；通常包含硬件DMA队列头(QH)，其中包含一个或多个传输说明符(TD) per URB

ep\_dev

sysfs信息的ep\_device

extra

在配置中跟随此端点的说明符

extralen

“extra”中有多少字节是有效的

enabled

URB可以提交到此端点

streams

在端点上分配的USB-3流的数量

**说明**

USB请求始终排队到给定端点，该端点由给定USB配置中的活动接口中的说明符标识。

#### struct usb\_interface

USB设备驱动程序交互的接口

**定义**

struct usb\_interface {

struct usb\_host\_interface \*altsetting;

struct usb\_host\_interface \*cur\_altsetting;

unsigned num\_altsetting;

struct usb\_interface\_assoc\_descriptor \*intf\_assoc;

int minor;

enum usb\_interface\_condition condition;

unsigned sysfs\_files\_created:1;

unsigned ep\_devs\_created:1;

unsigned unregistering:1;

unsigned needs\_remote\_wakeup:1;

unsigned needs\_altsetting0:1;

unsigned needs\_binding:1;

unsigned resetting\_device:1;

unsigned authorized:1;

struct device dev;

struct device \*usb\_dev;

struct work\_struct reset\_ws;

};

**成员**

altsetting

接口结构的数组，每个备用设置都可能被选择。每个都包括一组端点配置。它们将没有特定的顺序。

cur\_altsetting

当前的备用设置。

num\_altsetting

定义的备用设置数。

intf\_assoc

接口关联说明符

minor

如果此接口绑定到使用USB主要编号的驱动程序，则分配给此接口的副本号码。如果此接口不使用USB主要要素，则此字段应未使用。驱动程序应在驱动程序的probe()函数中设置此值，在调用usb\_register\_dev()时，由USB核心分配了一个次要编号后。

condition

接口的绑定状态未绑定、绑定(probe()中)、绑定到驱动程序或解绑(disconnect()中)。

sysfs\_files\_created

sysfs属性存在

ep\_devs\_created

端点子虚拟设备存在

unregistering

在注销接口时设置的标志

needs\_remote\_wakeup

驱动程序需要远程唤醒能力以进行自动挂起。

needs\_altsetting0

如果已延迟对备用设置0的设置接口请求，则设置标志。

needs\_binding

当驱动程序应在不支持的复位或挂起操作后重新探测或解绑时，设置标志。

resetting\_device

USB核心重置设备，因此使用alt设置0作为当前设置；重置后需要带宽分配。

authorized

与设备授权相比，这允许(de)在单个接口授权。

dev

该设备的驱动程序模型视图

usb\_dev

如果接口绑定到USB主要，则这将指向该设备的sysfs表示形式。

reset\_ws

用于从原子上下文调度重置。

**说明**

USB设备驱动程序附加到物理设备上的接口。每个接口封装单个高级功能，例如将音频流提供给扬声器或报告音量控制的更改。许多USB设备仅具有一个接口。用于与接口端点通信的协议可以由usb“类”规范或产品的供应商定义。默认控制端点是每个接口的一部分，但从未列在接口的说明符中。

绑定到接口的驱动程序可以使用标准驱动程序模型调用，例如对该struct 的dev成员调用dev\_get\_drvdata()。

每个接口可以具有备用设置。设备的初始配置设置为altsetting 0，但驱动程序可以使用usb\_set\_interface()更改该设置。通常使用备用设置来控制周期性端点的使用，例如让不同的端点使用不同数量的保留USB带宽。使用等时端点的所有符合标准的USB设备将在非默认设置中使用它们。

USB规范说备用设置编号必须从0到少一个备用设置的总备用设置数。但是，一些设备会搞砸这个，而且struct 不一定按数值顺序存储。使用usb\_altnum\_to\_altsetting()根据其编号在altsetting数组中查找备用设置。

#### struct usb\_interface\_cache

设备界面的长期表示

**定义**

struct usb\_interface\_cache {

unsigned num\_altsetting;

struct kref ref;

struct usb\_host\_interface altsetting[0];

};

**成员**

num\_altsetting

定义的altsettings数量。

ref

引用计数器。

altsetting

接口结构的可变长度数组，每个可选的备用设置都有一个。每个备用设置包括一组端点配置。它们不会有特定的顺序。

**说明**

与struct usb\_interface不同（只在其配置安装的时间内存在），这些结构在usb\_device的生命周期内保持不变。可以随时通过这些结构访问altsetting数组，允许比较配置并为/sys/kernel/debug/usb/devices伪文件提供支持。

#### struct usb\_host\_config

设备配置的表示

**定义**

struct usb\_host\_config {

struct usb\_config\_descriptor desc;

char \*string;

struct usb\_interface\_assoc\_descriptor \*intf\_assoc[USB\_MAXIADS];

struct usb\_interface \*interface[USB\_MAXINTERFACES];

struct usb\_interface\_cache \*intf\_cache[USB\_MAXINTERFACES];

unsigned char \*extra;

int extralen;

};

**成员**

desc

设备的配置说明符。

string

iConfiguration字符串的缓存版本的指针，如果此配置存在。

intf\_assoc

此配置中任何接口关联说明符的列表

interface

指向usb\_interface结构的指针数组，每个数组对应一个配置中的接口。接口的数量存储在desc.bNumInterfaces中。只有在配置处于活动状态时，这些指针才有效。

intf\_cache

指向usb\_interface\_cache结构的指针数组，每个数组对应一个配置中的接口。这些结构在整个设备的生命周期内存在。

extra

指向包含与此配置关联的所有额外说明符（在第一个接口说明符之前）的缓冲区的指针。

extralen

额外说明符缓冲区的长度。

**说明**

USB设备可以具有多个配置，但一次只能活动一个配置。每个封装一个不同的操作环境;例如，双速设备将为全速和高速操作分别有单独的配置。可用配置的数量存储在设备说明符中，如bNumConfigurations。配置可以包含多个接口。每个接口对应于USB设备的不同功能，并且在配置处于活动状态时所有接口都可用。USB标准规定接口的编号应从0到desc.bNumInterfaces-1，但许多设备做这个编号时会有错误。此外，接口数组的顺序不能保证按数字顺序排序。使用usb\_ifnum\_to\_if()按照其编号查找接口条目。设备驱动程序不应尝试激活配置。安装哪个配置是基于可用功率、提供的功能和用户的愿望（通过用户空间工具表达）等考虑的策略决定。但是，驱动程序可以调用usb\_reset\_configuration()来重新初始化当前配置及其所有接口。

#### struct usb\_device

USB设备的内核表示

**定义**

struct usb\_device {

int devnum;

char devpath[16];

u32 route;

enum usb\_device\_state state;

enum usb\_device\_speed speed;

unsigned int rx\_lanes;

unsigned int tx\_lanes;

struct usb\_tt \*tt;

int ttport;

unsigned int toggle[2];

struct usb\_device \*parent;

struct usb\_bus \*bus;

struct usb\_host\_endpoint ep0;

struct device dev;

struct usb\_device\_descriptor descriptor;

struct usb\_host\_bos \*bos;

struct usb\_host\_config \*config;

struct usb\_host\_config \*actconfig;

struct usb\_host\_endpoint \*ep\_in[16];

struct usb\_host\_endpoint \*ep\_out[16];

char \*\*rawdescriptors;

unsigned short bus\_mA;

u8 portnum;

u8 level;

u8 devaddr;

unsigned can\_submit:1;

unsigned persist\_enabled:1;

unsigned reset\_in\_progress:1;

unsigned have\_langid:1;

unsigned authorized:1;

unsigned authenticated:1;

unsigned wusb:1;

unsigned lpm\_capable:1;

unsigned lpm\_devinit\_allow:1;

unsigned usb2\_hw\_lpm\_capable:1;

unsigned usb2\_hw\_lpm\_besl\_capable:1;

unsigned usb2\_hw\_lpm\_enabled:1;

unsigned usb2\_hw\_lpm\_allowed:1;

unsigned usb3\_lpm\_u1\_enabled:1;

unsigned usb3\_lpm\_u2\_enabled:1;

int string\_langid;

char \*product;

char \*manufacturer;

char \*serial;

struct list\_head filelist;

int maxchild;

u32 quirks;

atomic\_t urbnum;

unsigned long active\_duration;

#ifdef CONFIG\_PM;

unsigned long connect\_time;

unsigned do\_remote\_wakeup:1;

unsigned reset\_resume:1;

unsigned port\_is\_suspended:1;

#endif;

struct wusb\_dev \*wusb\_dev;

int slot\_id;

enum usb\_device\_removable removable;

struct usb2\_lpm\_parameters l1\_params;

struct usb3\_lpm\_parameters u1\_params;

struct usb3\_lpm\_parameters u2\_params;

unsigned lpm\_disable\_count;

u16 hub\_delay;

};

**成员**

devnum

设备号； USB总线上的地址

devpath

用于消息的设备ID字符串（例如，/port/…）

route

用于与xHCI配对的树拓扑十六进制字符串

state

设备状态配置、未连接等等。

speed

设备速度高速/全速/低速（或错误）

rx\_lanes

使用的rx通道数，USB 3.2增加了双通道支持

tx\_lanes

使用的tx通道数，USB 3.2增加了双通道支持

tt

关于低/全速Dev、高速Hub的事务转换器信息

ttport

那个tt hub上的设备端口

toggle

每个端点一个位，其中（[0] = IN，[1] = OUT）端点

parent

我们的集线器，除非我们是根

bus

我们所在的总线

ep0

端点0的数据（默认控制管道）

dev

通用设备接口

descriptor

USB设备说明符

bos

USB设备BOS说明符集

config

设备的所有配置

actconfig

活动配置

ep\_in

IN端点的数组

ep\_out

OUT端点的数组

rawdescriptors

每个配置的原始说明符

bus\_mA

总线上可用的电流

portnum

父级端口编号（起始值为1）

level

USB hub祖先的数量

devaddr

设备地址，XHCI由硬件分配，其他与devnum相同

can\_submit

可以提交URB

persist\_enabled

为此设备启用USB\_PERSIST

have\_langid

字符串langid是否有效

authorized

策略已经说过我们可以使用它；（用户空间）策略决定是否授权使用此设备。默认情况下，有线USB设备是授权的。除非我们从用户空间授权它们，否则WUSB设备将不可用。 FIXME-完成文档

authenticated

加密身份验证通过

wusb

设备是无线USB

lpm\_capable

设备支持LPM

lpm\_devinit\_allow

允许USB3设备发起的LPM，退出延迟在范围内

usb2\_hw\_lpm\_capable

设备可以执行USB2硬件LPM

usb2\_hw\_lpm\_besl\_capable

设备可以执行USB2硬件BESL LPM

usb2\_hw\_lpm\_enabled

启用USB2硬件LPM

usb2\_hw\_lpm\_allowed

用户空间允许启用USB 2.0 LPM

usb3\_lpm\_u1\_enabled

启用USB3硬件U1 LPM

usb3\_lpm\_u2\_enabled

启用USB3硬件U2 LPM

string\_langid

字符串的语言ID

product

如果存在，则为iProduct字符串（静态）

manufacturer

如果存在，则为iManufacturer字符串（静态）

serial

如果存在，则为iSerialNumber字符串（静态）

filelist

此设备打开的usbfs文件

maxchild

如果是集线器，则为端口数量

quirks

整个设备的怪癖

urbnum

为整个设备提交的URB数

active\_duration

设备不挂起的总时间

connect\_time

设备首次连接的时间

do\_remote\_wakeup

应启用远程唤醒

reset\_resume

需要复位而不是恢复

port\_is\_suspended

上游端口已挂起（L2或U3）

wusb\_dev

如果这是无线USB设备，则链接到该设备的WUSB特定数据。

slot\_id

由xHCI分配的插槽ID

removable

可以从该端口物理移除设备

l1\_params

USB2 L1 LPM状态的最佳效率服务延迟和L1超时。

u1\_params

USB3 U1 LPM状态和hub-initiated超时的退出延迟。

u2\_params

USB3 U2 LPM状态和hub-initiated超时的退出延迟。

lpm\_disable\_count

usb\_disable\_lpm()和usb\_enable\_lpm()使用的引用计数，用于跟踪需要为此USB设备禁用USB 3.0链接电源管理功能的功能数。这个计数应该只被这些函数操作，当带宽\_mutex被持有时。

hub\_delay

缓存值，包含parent->hub\_delay + wHubDelay + tTPTransmissionDelay（40ns）

**描述**

将用作SetIsochDelay请求的wValue。

**注释**

Usbcore驱动程序不应直接设置usbdev->state。而应该使用usb\_set\_device\_state()。

#### usb\_hub\_for\_each\_child

usb\_hub\_for\_each\_child（hdev、port1、child）

迭代虹膜上的所有子设备

**参数**

hdev

属于usb hub的USB设备

port1

与子设备相关联的端口号码

child

子设备指针

#### int usb\_interface\_claimed（struct usb\_interface \*iface）

返回true（数字不为零）接口是否被认领

**参数**

struct usb\_interface \*iface

检查的接口

**返回**

真（数字不为零）如果接口被认领，否则为假（数字为零）。

**注释**

调用者必须拥有驱动程序模型的USB总线读锁定。因此，驱动程序probe()条目不需要额外的锁定，但其他调用上下文可能需要显式声明该锁定。

#### int usb\_make\_path（struct usb\_device \*dev，char \*buf，size\_t size）

在usb树中返回稳定的设备路径

**参数**

struct usb\_device \*dev

正在构建其路径的设备

char \*buf

放置字符串的位置

size\_t size

“buf”有多大？

**返回**

字符串的长度（> 0）或如果大小太小则为负数。

**注释**

该标识符旨在是“稳定的”，反映硬件中的物理路径，例如主机控制器或USB集线器上的物理总线地址等。这使它保持相同，直到物理重新配置系统，通过重新连接USB设备树或移动USB主机控制器。添加和删除设备，包括主控制器驱动程序模块中的虚拟根集线器，不会更改这些路径标识符；重新启动或重新enum也不会更改。这些标识符比可变（“不稳定”）标识符更实用，例如总线号或设备地址。

除了连接USB 2.0根集线器上的设备之外，这些标识符还是可预测的。只要设备树没有更改，在给定的集线器端口插入任何USB设备始终会给它分配相同的路径。由于使用“伴侣”控制器，因此连接到USB 2.0根集线器（EHCI主机控制器）上的端口上的设备如果是高速，则会获得一个路径ID，如果是完整或低速，则会获得另一个路径ID。

#### USB\_DEVICE

USB\_DEVICE（vend，prod）

用于说明特定USB设备的宏

**参数**

vend

16位USB厂商ID

prod

16位USB产品ID

**说明**

此宏用于创建与特定设备匹配的struct usb\_device\_id。

#### USB\_DEVICE\_VER

USB\_DEVICE\_VER（vend，prod，lo，hi）

说明具有版本范围的特定USB设备

**参数**

vend

16位USB供应商ID

prod

16位USB产品ID

lo

bcdDevice\_lo值

hi

bcdDevice\_hi值

**说明**

此宏用于创建与特定设备及其版本范围匹配的struct usb\_device\_id。

#### USB\_DEVICE\_INTERFACE\_CLASS

USB\_DEVICE\_INTERFACE\_CLASS（vend，prod，cl）

说明具有特定接口类的USB设备

**参数**

vend

16位USB供应商ID

prod

16位USB产品ID

cl

bInterfaceClass值

**说明**

此宏用于创建与特定设备接口类匹配的struct usb\_device\_id。

#### USB\_DEVICE\_INTERFACE\_PROTOCOL

USB\_DEVICE\_INTERFACE\_PROTOCOL（vend，prod，pr）

说明具有特定接口协议的USB设备

**参数**

vend

16位USB供应商ID

prod

16位USB产品ID

pr

bInterfaceProtocol值

**说明**

此宏用于创建与特定设备接口协议匹配的struct usb\_device\_id。

#### USB\_DEVICE\_INTERFACE\_NUMBER

USB\_DEVICE\_INTERFACE\_NUMBER（vend，prod，num）

说明具有特定接口编号的USB设备

**参数**

vend

16位USB供应商ID

prod

16位USB产品ID

num

bInterfaceNumber值

**说明**

此宏用于创建与特定设备接口编号匹配的struct usb\_device\_id。

#### USB\_DEVICE\_INFO

USB\_DEVICE\_INFO（cl，sc，pr）

用于说明USB设备类的宏

**参数**

cl

bDeviceClass值

sc

bDeviceSubClass值

pr

bDeviceProtocol值

**说明**

此宏用于创建与特定设备类匹配的struct usb\_device\_id。

#### USB\_INTERFACE\_INFO

USB\_INTERFACE\_INFO（cl，sc，pr）

用于说明USB接口类的宏

**参数**

cl

bInterfaceClass值

sc

bInterfaceSubClass值

pr

bInterfaceProtocol值

**说明**

此宏用于创建与特定接口类匹配的struct usb\_device\_id。

#### USB\_DEVICE\_AND\_INTERFACE\_INFO

USB\_DEVICE\_AND\_INTERFACE\_INFO（vend，prod，cl，sc，pr）

说明具有USB接口类别的特定USB设备

**参数**

vend

16位USB供应商ID

prod

16位USB产品ID

cl

bInterfaceClass值

sc

bInterfaceSubClass值

pr

bInterfaceProtocol值

**说明**

此宏用于创建与具有特定接口类别的特定设备匹配的struct usb\_device\_id。当显式匹配具有供应商特定bDeviceClass值但符合标准的接口的设备时，这非常有用。

#### USB\_VENDOR\_AND\_INTERFACE\_INFO

USB\_VENDOR\_AND\_INTERFACE\_INFO（vend，cl，sc，pr）

说明具有USB接口类别的特定USB供应商

**参数**

vend

16位USB供应商ID

cl

bInterfaceClass值

sc

bInterfaceSubClass值

pr

bInterfaceProtocol值

**说明**

此宏用于创建与具有特定接口类匹配的特定供应商匹配的struct usb\_device\_id。当显式匹配具有供应商特定bDeviceClass值但符合标准的接口的设备时，这非常有用。

#### struct usbdrv\_wrap

驱动程序模型结构的包装器

**定义**

struct usbdrv\_wrap {

struct device\_driver driver;

int for\_devices;

};

**成员**

driver

驱动程序模型核心驱动程序结构。

for\_devices

非零为设备驱动程序，0为接口驱动程序。

#### struct usb\_driver

标识USB接口驱动程序给usbcore

**定义**

struct usb\_driver {

const char \*name;

int (\*probe) (struct usb\_interface \*intf, const struct usb\_device\_id \*id);

void (\*disconnect) (struct usb\_interface \*intf);

int (\*unlocked\_ioctl) (struct usb\_interface \*intf, unsigned int code, void \*buf);

int (\*suspend) (struct usb\_interface \*intf, pm\_message\_t message);

int (\*resume) (struct usb\_interface \*intf);

int (\*reset\_resume)(struct usb\_interface \*intf);

int (\*pre\_reset)(struct usb\_interface \*intf);

int (\*post\_reset)(struct usb\_interface \*intf);

const struct usb\_device\_id \*id\_table;

const struct attribute\_group \*\*dev\_groups;

struct usb\_dynids dynids;

struct usbdrv\_wrap drvwrap;

unsigned int no\_dynamic\_id:1;

unsigned int supports\_autosuspend:1;

unsigned int disable\_hub\_initiated\_lpm:1;

unsigned int soft\_unbind:1;

};

**成员**

name

驱动程序名称应该在USB驱动程序中是唯一的，并且通常应该与模块名称相同。

probe

用于查看驱动程序是否愿意管理设备上的特定接口。如果是，probe返回零，并使用usb\_set\_intfdata（）将驱动程序特定数据与接口关联。它还可以使用usb\_set\_interface（）指定适当的altsetting。如果不愿意管理接口，请返回-ENODEV。如果发生真正的IO错误，则返回适当的负errno值。

disconnect

当接口不再可访问时，通常是因为其设备已断开连接，或者正在卸载驱动程序模块时调用。

unlocked\_ioctl

用于希望通过“usbfs”文件系统与用户空间通信的驱动程序。这允许设备提供将信息暴露给用户空间的方法，无论它们在文件系统的哪个位置显示（或不显示）。

suspend

当系统将设备挂起时（从系统睡眠或运行时挂起上下文），将调用此函数。在系统睡眠上下文中，返回将被忽略，因此不要尝试在此情况下继续使用设备。相反，让resume或reset-resume例程从故障中恢复。

resume

当设备被系统恢复时调用。

reset\_resume

当挂起的设备已被重置而不是恢复时调用。

pre\_reset

当usb\_reset\_device（）将设备即将重置时调用。此例程不能返回，直到驱动程序对该设备没有活动的URB，并且在post\_reset方法被调用之前不能再提交URB。

post\_reset

在设备重置后，由usb\_reset\_device（）调用

id\_table

USB驱动程序使用ID表支持热插拔。将其导出为MODULE\_DEVICE\_TABLE（usb，...）。如果不设置此项，则您的驱动程序的探测函数永远不会被调用。

dev\_groups

附加到设备的属性，当设备绑定到驱动程序时将创建这些属性。

dynids

内部用于保存此驱动程序的动态添加设备ID的列表。

drvwrap

驱动程序模型核心结构包装器。

no\_dynamic\_id

如果设置为1，则USB核心将通过防止创建sysfs文件来防止为此驱动程序添加动态ID。

supports\_autosuspend

如果设置为0，则USB核心将不允许将接口绑定到此驱动程序进行自动挂起。

disable\_hub\_initiated\_lpm

如果设置为1，则USB核心将不允许集线器在空闲超时时启动低功率链路状态转换。仍允许设备启动的USB 3.0链路PM。

soft\_unbind

如果设置为1，则USB核心将在调用驱动程序的断开方法之前不会杀死URB并禁用端点。

**说明**

USB接口驱动程序必须提供名称，probe（）和disconnect（）方法以及id\_table。其他驱动程序字段是可选的。

id\_table在热插拔中使用。它包含一组说明符，每个条目都可以关联专用数据。该表由用户和内核模式热插拔支持使用。

probe（）和disconnect（）方法在可以睡眠的情况下调用，但它们应该避免滥用特权。大多数连接到设备的工作应在设备打开时完成，并在最后关闭时撤消。断开连接代码需要解决与open（）和close（）方法有关的并发问题，以及强制所有挂起的I / O请求完成（通过必要的取消链接和阻止直到取消链接完成）。

#### struct usb\_device\_driver

标识USB设备驱动程序为usbcore

**定义**

struct usb\_device\_driver {

const char \*name;

int (\*probe) (struct usb\_device \*udev);

void (\*disconnect) (struct usb\_device \*udev);

int (\*suspend) (struct usb\_device \*udev, pm\_message\_t message);

int (\*resume) (struct usb\_device \*udev, pm\_message\_t message);

struct usbdrv\_wrap drvwrap;

unsigned int supports\_autosuspend:1;

};

**成员**

name

驱动程序名称应在USB驱动程序中唯一，并且通常应与模块名称相同。

probe

调用以查看驱动程序是否愿意管理特定设备。如果是，则probe返回零并使用dev\_set\_drvdata（）将驱动程序特定数据关联到设备。如果不愿意管理设备，则返回负errno值。

disconnect

当设备不再可访问时通常调用，通常是因为已断开连接或正在卸载驱动程序的模块。

suspend

当系统将要挂起设备时调用。

resume

当设备被系统恢复时调用。

drvwrap

驱动程序模型核心结构包装器。

supports\_autosuspend

如果设置为0，则USB核心将不允许将设备绑定到此驱动程序进行自动挂起。

**说明**

USB驱动程序必须提供上面列出的所有字段，除了drvwrap，match和id\_table。

#### struct usb\_class\_driver

标识想要使用USB主要编号的USB驱动程序

**定义**

struct usb\_class\_driver {

char \*name;

char \*(\*devnode)(struct device \*dev, umode\_t \*mode);

const struct file\_operations \*fops;

int minor\_base;

};

**成员**

name

此驱动程序的USB类设备名称。将出现在sysfs中。

devnode

回调函数，提供要创建的可能设备节点的命名提示。

fops

指向此驱动程序的struct file\_operations的指针。

minor\_base

此驱动程序的次要范围的开始。

**说明**

该结构用于usb\_register\_dev（）和usb\_deregister\_dev（）函数，以整合它们使用的多个参数。

#### module\_usb\_driver

module\_usb\_driver（\_\_usb\_driver）

注册USB驱动程序的帮助器宏

**参数**

\_\_usb\_driver

usb\_driver结构

**说明**

用于在模块init / exit中不执行任何特殊操作的USB驱动程序的帮助器宏。这将消除大量样板文件。每个模块只能使用此宏一次，并且调用它将取代module\_init（）和module\_exit（）

#### struct urb

USB请求块

**定义**

struct urb {

struct list\_head urb\_list;

struct list\_head anchor\_list;

struct usb\_anchor \*anchor;

struct usb\_device \*dev;

struct usb\_host\_endpoint \*ep;

unsigned int pipe;

unsigned int stream\_id;

int status;

unsigned int transfer\_flags;

void \*transfer\_buffer;

dma\_addr\_t transfer\_dma;

struct scatterlist \*sg;

int num\_mapped\_sgs;

int num\_sgs;

u32 transfer\_buffer\_length;

u32 actual\_length;

unsigned char \*setup\_packet;

dma\_addr\_t setup\_dma;

int start\_frame;

int number\_of\_packets;

int interval;

int error\_count;

void \*context;

usb\_complete\_t complete;

struct usb\_iso\_packet\_descriptor iso\_frame\_desc[0];

};

**成员**

urb\_list

用于当前 URB 拥有者使用。

anchor\_list

属于锚点列表的成员

anchor

将 URB 锚定到公共系泊。

dev

标识要执行请求的 USB 设备。

ep

指向端点数据结构。最终将替换管道。

pipe

包含端点编号、方向、类型等。使用八个可用的宏创建这些值；usb\_{snd,rcv}TYPEpipe(dev,endpoint)，其中 TYPE 是 “ctrl” (控制)、“bulk”、“int” (中断) 或 “iso”(等时) 。例如usb\_sndbulkpipe() 或 usb\_rcvintpipe()。端点编号从零到十五。注意，“in”端点二是与“out”端点二不同的端点(和管道)。当前配置控制任何给定端点的存在、类型和最大包大小。

stream\_id

用于批量流的端点流 ID

status

在非等时完成函数中读取这个变量可以获取特定请求的状态。ISO 请求仅用它来告诉 URB 是否已被解除链接; 每个帧的详细状态在 iso\_frame-desc 的字段中。

transfer\_flags

可以使用多种标志来影响如何处理 URB 的提交、解除链接或操作。不同类型的 URB 可以使用不同的标志。

transfer\_buffer

这标识要执行 I/O 请求的缓冲区，除非设置了 URB\_NO\_TRANSFER\_DMA\_MAP，否则不要在 transfer\_buffer 中留下垃圾(即使是这样，也不要在 transfer\_buffer 中留下垃圾)。这个缓冲区必须适合进行 DMA；使用 kmalloc() 或等效函数进行分配。对于传输到“in”端点，这个缓冲区的内容将被修改。这个缓冲区用于控制传输的数据阶段。

transfer\_dma

当 transfer\_flags 包括 URB\_NO\_TRANSFER\_DMA\_MAP 时，设备驱动程序表示它提供了这个 DMA 地址，主机控制器驱动程序应优先使用它，而不使用 transfer\_buffer。

sg

分散收集缓冲区列表，如果没有在 ‘struct usb\_bus’ 中设置 no\_sg\_constraint，则列表中每个元素的缓冲区大小必须是端点的最大包大小的整数倍

num\_mapped\_sgs

映射的 sg 条目数(内部)

num\_sgs

sg 列表中的条目数

transfer\_buffer\_length

transfer\_buffer 大小，传输可能会根据当前端点的最大包大小分成多个块，这个大小是端点的函数，它编码在 pipe 中。当长度为零时，不使用 transfer\_buffer 或 transfer\_dma。

actual\_length

在非等时完成函数中读取这个变量可以告诉你实际传输了多少字节(transfer\_buffer\_length 中的字节数)。通常与请求的字节数相同，除非报告了错误或执行了短读取。可以使用 URB\_SHORT\_NOT\_OK 传输标志，使这样的短读取被报告为错误。

setup\_packet

仅用于控制传输，这指向八字节的设置数据。控制传输总是首先向设备发送这些数据。然后，如果需要，读取或写入 transfer\_buffer。

setup\_dma

设置数据包的DMA指针。调用者不能使用这个字段；setup\_packet 必须指向一个有效的缓冲区。

start\_frame

返回等时传输的初始帧。

number\_of\_packets

列出 ISO 传输缓冲区的数量。

interval

指定中断或等时传输的轮询间隔。对于全速和低速设备，单位为帧(毫秒)，对于高速和超级速度设备，单位为微帧(1/8 毫秒)。

error\_count

返回报告错误的 ISO 传输数。

context

用于完成函数。这通常指向请求特定的驱动程序上下文。

complete

完成处理程序。这个 URB 作为参数传递给完成函数。完成函数可以自己处理 URB，包括重新提交或释放它。

iso\_frame\_desc

用于提供 ISO 传输缓冲区的数组，并为每个缓冲区收集传输状态。

**说明**

这个结构标识 USB 传输请求。必须使用 usb\_alloc\_urb() 调用分配 URB，并使用 usb\_free\_urb() 调用释放。可以使用各种 usb\_fill\_\*\_urb() 函数进行初始化。使用 usb\_submit\_urb() 提交 URB，使用 usb\_unlink\_urb() 或 usb\_kill\_urb() 取消挂起请求。

数据传输缓冲区:

通常驱动程序提供使用 kmalloc() 分配的 I/O 缓冲区或其他从通用页面池中获取的缓冲区。它由 transfer\_buffer 提供(控制请求也使用 setup\_packet)，主机控制器驱动程序为每个传输的缓冲区执行 DMA 映射(和解映射)。这些映射操作在一些平台上可能是昂贵的(例如使用 DMA 弹跳缓冲区或与 IOMMU 通信)，尽管在通货流通的 x86 和 ppc 硬件上它们是便宜的。

或者，驱动程序可以传递 URB\_NO\_TRANSFER\_DMA\_MAP 传输标志，告诉主机控制器驱动程序对于 transfer\_buffer 不需要这样的映射，因为设备驱动程序是 DMA 感知的。例如，设备驱动程序可能使用 usb\_alloc\_coherent() 分配 DMA 缓冲区，或者调用 usb\_buffer\_map()。当提供这个传输标志时，主机控制器驱动程序将尝试使用在 transfer\_dma 字段中找到的 DMA 地址，而不是自己确定 DMA 地址。

请注意，如果控制器不支持DMA（由hcd\_uses\_dma()指示），或与根集线器通信，则仍必须设置transfer\_buffer。如果在这种控制器和设备之间进行传输，则必须创建一个跳跃缓冲区或以错误退出。如果无法设置transfer\_buffer（在高性能内存中），且控制器支持DMA，则将其分配NULL以使usbmon不使用该值。 setup\_packet必须始终设置，因此不能将其定位在高性能内存中。

初始化:

提供的所有URB都必须初始化dev，pipe，transfer\_flags（可能为零）和完整字段。所有URB也必须初始化transfer\_buffer和transfer\_buffer\_length。它们可以提供URB\_SHORT\_NOT\_OK传输标志，表示短读取将被视为错误；该标志对于写请求无效。

大容量URB可以使用URB\_ZERO\_PACKET传输标志，表示大容量OUT传输应始终以短数据包终止，即使这意味着添加一个额外的零长度数据包。

控制URB必须在setup\_packet字段中提供有效的指针。与transfer\_buffer不同，setup\_packet不能事先映射为DMA。

中断URB必须提供一个间隔，表示在多长时间内（以毫秒或高速设备的125微秒单位为单位）轮询传输。提交URB后，间隔字段反映了传输实际上是如何安排的。轮询间隔可能比请求的更频繁。例如，一些控制器的最大间隔为32毫秒，而其他控制器支持长达1024毫秒的间隔。等时URB也有传输间隔。（请注意，对于等时端点以及高速中断端点，传输间隔在端点说明符中的编码是对数的。设备驱动程序必须自行将该值转换为线性单位。）

如果等时端点队列尚未运行，则主机控制器将安排一个新的URB，在带宽利用允许的情况下立即启动。如果队列正在运行，则将安排一个新的URB，在前一个URB结束后的第一个传输插槽中启动，如果该插槽尚未过期。如果时隙已过期（当IRQ传递被延迟很长时间时可能会发生），则调度行为取决于URB\_ISO\_ASAP标志。如果标志未清除，则URB将被安排在过期的时隙中启动，这意味着有些数据包将不会传输；如果标志被设置，则URB将在第一个未过期的插槽中安排，打破队列的同步。完成URB后，start\_frame字段将设置为调度传输的（微）帧号。帧计数器值的范围是特定于控制器的，可以从256低至65536帧。

等时URB具有不同的数据传输模型，部分原因是由于服务质量只是“尽力而为”。调用者提供专门分配的URB，在末尾有number\_of\_packets个iso\_frame\_desc结构。每个这样的数据包都是单独的ISO传输。等时URB通常排队，由驱动程序提交以安排至少双缓冲区，然后在完成处理程序中明确重新提交，以使数据（例如音频或视频）以主机控制器调度程序可以支持的尽可能恒定的速率流动。

完成回调

完成回调在in\_interrupt()中进行，并且完成处理程序应该做的第一件事情之一是检查状态字段。状态字段适用于所有URB。它用于报告未链接的URB和所有非ISO传输的状态。在URB返回到完成处理程序之前，不应该检查它。

上下文字段通常用于将URB链接回相关的驱动程序或请求状态。

当完成回调被调用用于非等时性URB时，actual\_length字段告诉有多少字节被传输。即使URB以错误或未连接终止，这个字段也会被更新。

ISO传输状态在iso\_frame\_desc数组的status和actual\_length字段中报告，错误数量在error\_count中报告。等时性传输的完成回调通常会重新提交URB，以确保恒定的传输速率。

请注意，即使字段标记为“public”，驱动程序也不应在URB由hcd拥有（即从usb\_submit\_urb()调用到进入完成例程）时触摸它们。

#### **void** usb\_fill\_control\_urb(struct urb \*urb, struct usb\_device \*dev, unsigned int pipe, unsigned char\*setup\_packet, void \*transfer\_buffer, int buffer\_length, usb\_complete\_t complete\_fn, void \*context)

初始化控制URB

**参数**

struct urb \*urb

指向urb需初始化的指针

struct usb\_device \*dev

指向此urb所需的struct usb\_device的指针。

unsigned int pipe

端点pipe

unsigned char \*setup\_packet

指向setup\_packet缓冲区的指针

void \*transfer\_buffer

指向传输缓冲区的指针

int buffer\_length

传输缓冲区的长度

usb\_complete\_t complete\_fn

指向usb\_complete\_t函数的指针

void \*context

将urb上下文设置为什么。

**说明**

使用所需的正确信息初始化控制urb，并将其提交给设备。

#### void usb\_fill\_bulk\_urb(struct urb \*urb, struct usb\_device \*dev, unsigned int pipe, void \*transfer\_buffer, int buffer\_length, usb\_complete\_t complete\_fn, void \*context)

帮助初始化批量urb的宏

**参数**

struct urb \*urb

指向urb需初始化的指针

struct usb\_device \*dev

指向此urb所需的struct usb\_device的指针。

unsigned int pipe

端点pipe

void \*transfer\_buffer

指向传输缓冲区的指针

int buffer\_length

传输缓冲区的长度

usb\_complete\_t complete\_fn

指向usb\_complete\_t函数的指针

void \*context

将urb上下文设置为什么。

**说明**

使用所需的正确信息初始化批量urb，并将其提交给设备。

#### void usb\_fill\_int\_urb(struct urb \*urb, struct usb\_device \*dev, unsigned int pipe, void \*transfer\_buffer, int buffer\_length, usb\_complete\_t complete\_fn, void \*context, int interval)

帮助初始化中断urb的宏

**参数**

struct urb \*urb

指向urb需初始化的指针

struct usb\_device \*dev

指向此urb所需的struct usb\_device的指针。

unsigned int pipe

端点pipe

void \*transfer\_buffer

指向传输缓冲区的指针

int buffer\_length

传输缓冲区的长度

usb\_complete\_t complete\_fn

指向usb\_complete\_t函数的指针

void \*context

将urb上下文设置为什么。

int interval

将urb间隔设置为什么，编码方式与端点说明符的bInterval值相同。

**说明**

使用所需的正确信息初始化中断urb，并将其提交给设备。

请注意，高速和超级速度(+)中断端点使用端点间隔的对数编码，并将轮询间隔表示为微帧(每毫秒八个)而不是帧(每毫秒一个)。无线USB也使用对数编码，但是以128us为单位指定它，而不是125us。对于无线USB设备，间隔通过主机控制器传递，而不是被转换为微帧单位。

#### int usb\_urb\_dir\_in(struct urb \*urb)

检查URB是否说明IN传输

**参数**

struct urb \*urb

要检查的URB

**返回**

如果urb说明IN传输(设备到主机)，则返回1，否则返回0。

#### int usb\_urb\_dir\_out(struct urb \*urb)

检查URB是否说明OUT传输

**参数**

struct urb \*urb

要检查的URB

**返回**

如果urb说明OUT传输(主机到设备)，则返回1，否则返回0。

#### struct usb\_sg\_request

支持散布/聚集I/O

**定义**

struct usb\_sg\_request {

int status;

size\_t bytes;

};

**成员**

status

0表示成功，否则为负数错误号

bytes

计数传输的字节数。

**说明**

使用usb\_sg\_init()初始化这些请求，然后将它们用作请求句柄传递给usb\_sg\_wait()或usb\_sg\_cancel()。请求对象的大多数成员不是为驱动程序访问而设计的。

只有在usb\_sg\_wait()返回后，状态和字节计数值才是有效的。如果状态为0，则字节计数与请求的总字节数相匹配。

在错误完成后，驱动程序可能需要在端点上清除暂停条件。

### USB核心API

USB API中有两种基本的I/O模型。最基本的一种是异步的:驱动程序以URB的形式提交请求，并且URB的完成回调处理下一步。所有USB传输类型都支持该模型，尽管控制URB有特殊情况(始终具有设置和状态阶段，但可能没有数据阶段)，等时URB允许大数据包并包括每个数据包的故障报告。在此基础上建立同步API支持，其中驱动程序调用分配一个或多个URB的例程，提交它们，并等待它们完成。单个缓冲区控制和批量传输(在某些驱动程序断开连接方案中使用不方便)，以及散列表基础的流式传输和中断传输都有同步包装器。

USB驱动程序需要提供可用于DMA的缓冲区，尽管它们不一定需要自己提供DMA映射。在分配DMA缓冲区时使用的API可以防止在某些系统上使用反弹缓冲区。在某些情况下，驱动程序可以依靠64位DMA来消除另一种反弹缓冲区。

#### void usb\_init\_urb(struct urb \*urb)

初始化urb，以便USB驱动程序可以使用它

**参数**

struct urb \*urb

要初始化的urb指针

如果使用usb\_alloc\_urb()创建urb，则不需要调用此函数。只有在自己分配struct urb的空间时才使用此函数。如果调用此函数，请在释放urb的内存时小心，确保USB核心不再使用它。

只在您真正理解自己在做什么时才使用此函数。

#### struct urb \*usb\_alloc\_urb(int iso\_packets, gfp\_t mem\_flags)

创建一个新的urb供USB驱动程序使用

**参数**

int iso\_packets

该urb的iso数据包数

gfp\_t mem\_flags

要分配的内存类型，请参阅kmalloc()以获取此类型的有效选项列表。

**说明**

为USB驱动程序创建urb，初始化一些内部结构，增加使用计数器并返回指向它的指针。

如果驱动程序要将此urb用于中断，控制或批量端点，请将'0'作为iso数据包数。

驱动程序必须在完成urb时调用usb\_free\_urb()。

**返回**

指向新urb的指针，如果没有可用的内存，则为NULL。

#### void usb\_free\_urb(struct urb \*urb)

在所有使用者完成后释放urb所使用的内存

**参数**

struct urb \*urb

指向要释放的urb的指针，可以为NULL

**说明**

当urb的用户完成时必须调用它。当urb的最后一个用户调用此函数时，urb的内存被释放。

**注意**

除非设置了URB\_FREE\_BUFFER传输标志，否则不会释放与urb相关联的传输缓冲区。

#### struct urb \*usb\_get\_urb(struct urb \*urb)

增加urb的参考计数

**参数**

struct urb \*urb

指向要修改的urb的指针，可以为NULL

**说明**

每当将urb从设备驱动程序传输到主机控制器驱动程序时，必须调用此函数。这允许适当的urbs参考计数。

**返回**

指向具有增加的参考计数的urb的指针。

#### void usb\_anchor\_urb(struct urb \*urb, struct usb\_anchor \*anchor)

在处理urb时钳住urb

**参数**

struct urb \*urb

要绑定的urb的指针

struct usb\_anchor \*anchor

指向锚点的指针

**说明**

这个方法可以让您访问不必追踪的URB

#### void usb\_unanchor\_urb(struct urb \*urb)

取消urb的关联

**参数**

struct urb \*urb

要锚定的urb的指针

**说明**

调用此来停止系统跟踪此URB。

#### int usb\_urb\_ep\_type\_check(const struct urb \*urb)

在给定的urb中进行端点的合理性检查

**参数**

const struct urb \*urb

要检查的urb

**说明**

这会对urb中的端点进行轻量级合理性检查。如果urb包含有效的端点，则返回0，否则返回负错误代码。

#### int usb\_submit\_urb(struct urb \*urb, gfp\_t mem\_flags)

为端点发出异步传输请求

**参数**

struct urb \*urb

说明请求urb的指针

gfp\_t mem\_flags

要分配的内存类型，请参阅kmalloc()以获取此类型的有效选项列表。

**说明**

这个方法提交一个传输请求，并将说明该请求的URB的控制转移到USB子系统。稍后，将通过调用完成处理程序异步地指示请求完成。完成的三种类型是成功，错误和取消链接（软件引起的故障，也称为“请求取消”）。

URB可以在中断上下文中提交。

在提交URB之前，调用方必须正确初始化URB。例如，usb\_fill\_bulk\_urb()和usb\_fill\_control\_urb()等方法可供使用，以确保大多数字段正确初始化，以进行特定类型的传输，尽管它们不会初始化任何传输标志。

如果提交成功，则URB的完成回调将在USB核心和主机控制器驱动程序（HCD）完成URB时仅调用一次。当调用完成函数时，将URB的控制返回给发出请求的设备驱动程序。然后，完成处理程序可以立即释放或重用该URB。

除周期性传输调度外，USB设备驱动程序不应访问由usbcore或HCD提供的URB字段，直到完全完成（complete（））它。这些例外涉及周期性传输调度。对于中断和等时URB，作为成功URB提交的一部分，urb->interval会被修改以反映实际使用的传输周期（通常是某些2的幂单位）。对于等时URB，urb->start\_frame被修改以反映URB的传输何时被调度开始。

并非所有的等时传输调度策略都适用，但是大多数主机控制器驱动程序应该容易处理从现在到10-200毫秒的ISO队列。驱动程序应该尝试保留至少一到两毫秒的数据在队列中；许多控制器要求添加时，新传输至少在未来1毫秒开始。如果驱动程序无法跟上并且队列变空，则新提交的行为由URB\_ISO\_ASAP标志管理。如果设置了标志或队列处于空闲状态，则URB始终分配给端点计划中的第一个可用（尚未过期）插槽。如果未设置标志并且队列正在活动，则URB始终分配给沿着端点之前URB结束后的时间表中的下一个插槽，即使该插槽已经过时。当将数据包分配到已过期的插槽时，该数据包不会传输，并且相应的usb\_iso\_packet\_descriptor的状态字段将返回-EXDEV。如果URB中的所有数据包都这样分配，则提交会失败并返回-EXDEV错误代码。

对于控制端点，通常使用同步的usb\_control\_msg()调用（在非中断上下文中）来代替此调用。通常使用方便的包装器，用于符合USB 2.0规范的请求。对于大容量端点，存在一个同步的usb\_bulk\_msg()调用。

**返回**

0成功提交，否则为负错误数

请求排队：

URB可以在之前的请求完成之前提交到端点，以最小化中断延迟和系统开销对数据吞吐量的影响。采用这种排队策略，端点队列永远不会为空。这对于连续的异步数据流是必需的，并且在某些种类的中断传输中也可能是必需的。这种排队还通过允许USB控制器在驱动程序软件完成了早期（成功的）请求的完成处理之前开始后续请求的工作，从而最大化了带宽利用率。

从Linux 2.6开始，所有USB端点传输队列都支持深度大于1。这以前是HCD特定的行为，ISO传输除外。非异步端点队列在故障清理后处于不活动状态（传输错误或取消）。

保留带宽传输：

周期性传输（中断或异步）是重复执行的，使用urb中指定的间隔。将第一个urb提交到端点会保留执行这些传输所需的带宽。如果USB子系统无法分配足够的带宽来执行周期性请求，则提交此类周期性请求应失败。

对于xHCI下的设备，带宽是在配置时间或选择备用设置时保留的。如果总线带宽不足，则配置/备用设置请求将失败。因此，xHCI下的设备定期端点的提交不应因带宽限制而失败。

设备驱动程序必须显式请求重复，必须确保某些URB始终在端点队列上（除了完成回调期间的短暂时间）。当不再排队urb时，端点的带宽保留将被取消。这意味着驱动程序可以使用其完成处理程序，通过重新初始化和重新提交刚刚完成的urb来确保它们保留所需的带宽，直到驱动程序不再需要该周期性带宽为止。

内存标志：

决定使用哪些mem\_flags的一般规则与kmalloc相同。有四种可能的值；GFP\_KERNEL、GFP\_NOFS、GFP\_NOIO和GFP\_ATOMIC。

GFP\_NOFS从未被使用过，因为尚未实施。

当您处于完成处理程序、中断、底半部、tasklet或计时器时，或当您持有自旋锁或rwlock时（不适用于信号量），或current->state! =TASK\_RUNNING时，使用GFP\_ATOMIC。

在存储设备的块IO路径和错误处理中使用GFP\_NOIO。

所有其他情况使用GFP\_KERNEL。

可以推断出一些更具体的mem\_flags规则，例如

1. 网络驱动程序的start\_xmit、timeout和receive方法必须使用GFP\_ATOMIC（使用自旋锁调用）；
2. SCSI驱动程序的queuecommand方法必须使用GFP\_ATOMIC（也需要使用自旋锁调用）
3. 如果您使用带有网络驱动程序的内核线程，则必须使用GFP\_NOIO，除非（b）或（c）适用；
4. 在完成down（）操作后，可以使用GFP\_KERNEL，除非（b）或（c）适用或您在存储驱动程序的块IO路径中；
5. USB探测和断开连接可以使用GFP\_KERNEL，除非（b）或（c）适用；并且
6. 更改运行中的存储或网络设备上的固件使用GFP\_NOIO，除非适用b）或c）。

#### int usb\_unlink\_urb(struct urb \*urb)

中止/取消端点的传输请求

**参数**

struct urb \*urb

指向说明先前提交的请求的urb的指针，可能为NULL

**说明**

此例程取消正在处理的请求。 URB每次提交只完成一次，并且每次提交只能取消一次。成功取消意味着urb的终止将被加速，并将调用完成处理程序，指示请求已被取消（而不是任何其他代码）的状态代码。

驱动程序不应在其断开连接方法返回后调用此例程或相关的例程，例如usb\_kill\_urb（）或usb\_unlink\_anchored\_urbs（）。断开功能应与驱动程序的I / O例程同步，以确保在返回之前完成所有与URB相关的活动。

此请求是异步的，但是HCD可能会在unlink期间调用->complete（）回调。因此，当驱动程序调用usb\_unlink\_urb（）时，它们不能持有可能被完成函数接管的任何锁。成功通过返回-EINPROGRESS表示，此时URB可能尚未返回给设备驱动程序。当它最终被调用时，完成函数将看到urb->status == -ECONNRESET。如果usb\_unlink\_urb（）返回其他任何值，则表示失败。即使完成处理程序尚未运行，'unlink'也将在urb当前未“链接”（即，从未提交，或已在之前的提交中取消，或硬件已经完成了它）时失败。

在这个例程运行时，不得取消分配 URB。特别是，当驱动程序调用此例程时，必须确保完成处理程序不能取消分配 URB。

**返回**

成功返回 -EINPROGRESS。失败时请参见上述说明以了解其他值。

解除链接和端点队列

[下面说明的行为和保证仅适用于物理 USB 设备的端点队列，而不适用于虚拟根集线器。]

主机控制器驱动程序（HCD）将所有特定端点的 URB 放在一个队列中。通常队列在控制器硬件处理每个请求时会推进。但是当 URB 以错误结束时，它的队列通常会停止（见下文），至少直到该 URB 的完成例程返回。保证已停止的队列直到所有未链接的 URB 完全退役并运行其完成例程才开始重新启动，即使这是在原始完成处理程序退回之后的某个时候。

当 URB 因为被取消链接而终止时，相同的行为和保证也适用。保证无论出现什么样的错误，包括 -ECONNRESET、-ENOENT 和 -EREMOTEIO，块和中断端点队列都会以相同的方式停止。控制端点队列的行为相同，只是对于 -EREMOTEIO 错误它们不能保证停止。对等时端点的队列处理方式不同，因为它们必须以固定速率推进。这样的队列在 URB 遇到错误或被取消链接时不会停止。取消链接的等时 URB 可能在分组流中留下间隙；未确定是否可以填补这样的间隙。

请注意，如果接收到短数据包而早期终止 URB，只有 URB\_SHORT\_NOT\_OK 标志被设置才会生成 -EREMOTEIO 错误。通过设置此标志，USB 设备驱动程序可以为大型或复杂的块传输构建深度队列，并在出现任何类型的中止传输时可靠地清理它们，方法是取消链接所有待处理 URB，并在出现第一个故障时。

当控制 URB 以 -EREMOTEIO 以外的错误终止时，很可能转移的状态阶段不会发生。

#### void usb\_kill\_urb（struct urb \*urb）

取消传输请求并等待传输完成

**参数**

struct urb \*urb

指向说明先前提交的请求的 URB 的指针，可能为 NULL

**说明**

此例程取消正在进行的请求。返回后保证所有完成处理程序都已完成，URB 将完全空闲并可供重用。这些功能使其成为在 disconnect() 回调函数或 close() 函数中停止 I/O 的理想方式。如果请求尚未完成或未取消链接，则完成处理程序将看到 urb->status == -ENOENT。

当此例程在运行时，重新提交 URB 的尝试将失败并显示错误 -EPERM。因此，即使 URB 的完成处理程序总是尝试重新提交，它也不会成功，而 URB 将变为空闲状态。

在此例程运行时，不得取消分配 URB。特别是，当驱动程序调用此例程时，必须确保完成处理程序不能取消分配 URB。

此例程不能在中断上下文（如底半部分或完成处理程序）中使用，也不能在保持自旋锁或在其他无法计划的情况下使用()。

驱动程序在其断开方法返回后不应调用此例程。

#### void usb\_poison\_urb（struct urb \*urb）

可靠地取消传输并防止进一步使用 URB

**参数**

struct urb \*urb

指向说明先前提交的请求的 URB 的指针，可能为 NULL

**说明**

此例程取消正在进行的请求。返回后保证所有完成处理程序都已完成，URB 将完全空闲并且无法重用。这些功能使其成为 disconnect() 回调函数中停止 I/O 的理想方式。如果请求尚未完成或未取消链接，则完成处理程序将看到 urb->status == -ENOENT。

此例程运行后，重新提交 URB 的尝试将失败并显示错误 -EPERM。因此，即使 URB 的完成处理程序总是尝试重新提交，它也不会成功，而 URB 将变为空闲状态。

在此例程运行时，不得取消分配 URB。特别是，当驱动程序调用此例程时，必须确保完成处理程序不能取消分配 URB。

此例程不能在中断上下文（如底半部分或完成处理程序）中使用，也不能在保持自旋锁或在其他无法计划的情况下使用()。

驱动程序在其断开方法返回后不应调用此例程。

#### void usb\_block\_urb（struct urb \*urb）

可靠地阻止URB的进一步使用

**参数**

struct urb \* urb

指向要阻止的URB的指针，可以为NULL

**说明**

运行该例程后，尝试重新提交URB将失败，并显示错误-EPERM。因此，即使URB的完成处理程序始终尝试重新提交，也将无法成功，并且URB将变为空闲状态。

在运行此例程时不能取消分配URB。特别是当驱动程序调用此例程时，必须确保完成处理程序无法取消分配URB。

#### void usb\_kill\_anchored\_urbs（struct usb\_anchor \* anchor）

杀死与锚点关联的所有URB

**参数**

struct usb\_anchor \* anchor

绑定请求的锚点

**说明**

这将杀死从队列后面开始的所有未完成的URB，并保证在此函数返回后，锚点不会发生完成回调。

驱动程序在返回其断开方法后不应调用此例程。

#### void usb\_poison\_anchored\_urbs（struct usb\_anchor \* anchor）

停止来自锚点的所有流量

**参数**

struct usb\_anchor \* anchor

绑定请求的锚点

**说明**

这允许从队列后面开始毒害所有未完成的URB。新添加的URB也将被毒害

驱动程序在返回其断开方法后不应调用此例程。

#### void usb\_unpoison\_anchored\_urbs（struct usb\_anchor \* anchor）

使锚点再次成功使用

**参数**

struct usb\_anchor \* anchor

绑定请求的锚点

**说明**

撤消usb\_poison\_anchored\_urbs的效果，锚点在返回后可以正常使用

#### void usb\_unlink\_anchored\_urbs（struct usb\_anchor \* anchor）

异步取消传输请求的请求

**参数**

struct usb\_anchor \* anchor

绑定请求的锚点

**说明**

这允许从队列后面开始取消链接所有未完成的URB。这个功能是异步的。只是触发了取消链接。在此函数返回后，它可能会发生。

驱动程序在返回其断开方法后不应调用此例程。

#### void usb\_anchor\_suspend\_wakeups（struct usb\_anchor \* anchor）

**参数**

struct usb\_anchor \* anchor

要中止唤醒的锚点

**说明**

调用此函数可防止从未锚定的urb唤醒任何usb\_wait\_anchor\_empty\_timeout waiters。这在hcd urb give-back路径中用于延迟唤醒，直到完成处理程序运行。

#### void usb\_anchor\_resume\_wakeups（struct usb\_anchor \* anchor）

**参数**

struct usb\_anchor \* anchor

要恢复唤醒的锚点

**说明**

允许usb\_wait\_anchor\_empty\_timeout waiters再次被唤醒，如果该锚点为空，则唤醒当前waiters。

#### int usb\_wait\_anchor\_empty\_timeout（struct usb\_anchor \* anchor，unsigned int timeout）

等待锚点不再使用

**参数**

struct usb\_anchor \* anchor

要变成未使用的锚点

unsigned int timeout

允许等待的时间（以毫秒为单位）

**说明**

如果要确保锚点的所有URB都已完成，请调用此函数

返回

如果锚点变为空闲，则非零。超时返回零。

#### struct urb \* usb\_get\_from\_anchor（struct usb\_anchor \* anchor）

获取锚点的最旧urb

**参数**

struct usb\_anchor \* anchor

要获取urb的锚点

**说明**

这将从锚点提取最旧的urb，取消锚定并返回它

返回

来自锚点的最旧urb，如果锚点没有URB与其关联，则为NULL。

#### void usb\_scuttle\_anchored\_urbs（struct usb\_anchor \* anchor）

取消链接所有锚点电缆

**参数**

struct usb\_anchor \* anchor

要取消锚定URB的锚点

**说明**

使用此功能可以摆脱锚点的所有URB

int usb\_anchor\_empty（struct usb\_anchor \* anchor）

锚定是否为空

**参数**

struct usb\_anchor \* anchor

要查询的锚定对象

**返回**

如果锚点没有与之关联的URB，则返回1。

#### int usb\_control\_msg（struct usb\_device \* dev，unsigned int pipe，\_\_u8 request，\_\_u8 requesttype，\_\_u16 value，\_\_u16 index，void \* data，\_\_u16 size，int timeout）

构建控件urb，发送并等待完成

**参数**

struct usb\_device \* dev

指向要发送消息的USB设备的指针

unsigned int pipe

要发送消息的端点“管道”

\_\_u8 request

USB消息请求值

\_\_u8 requesttype

USB消息请求类型值

\_\_u16 value

USB消息值

\_\_u16 index

USB消息索引值

void \* data

要发送的数据指针

\_\_u16 size

要发送的数据的字节数

int timeout

在超时之前等待消息完成的时间（如果为0，则等待永远）

**上下文**

！in\_interrupt（）

**说明**

此函数向指定的端点发送一个简单的控制消息，并等待该消息完成或超时。

不要从中断上下文中使用此函数。如果需要异步消息或需要在中断上下文中发送消息，请使用usb\_submit\_urb（）。如果驱动程序中的线程使用此调用，请确保您的disconnect（）方法可以等待它完成。由于您没有使用URB的句柄，因此无法取消请求。

**返回**

如果成功，传输的字节数。否则，是一个负错误号码。

#### int usb\_interrupt\_msg(struct usb\_device \*usb\_dev，unsigned int pipe，void \*data，int len，int \*actual\_length，int timeout)

构建一个中断urb，发送它并等待完成

**参数**

struct usb\_device \*usb\_dev

指向要发送消息的USB设备的指针

unsigned int pipe

要发送消息的端点“管道”

void \*data

要发送的数据的指针

int len

要发送的数据的字节数

int \*actual\_length

将传输的实际长度（字节）放置到的位置的指针

int timeout

在超时之前等待消息完成的时间（如果为0，则永远等待）

**上下文**

！在中断（）中

**说明**

此函数向指定的端点发送简单的中断消息，并等待消息完成或超时。

不要在中断上下文中使用此函数。如果您需要异步消息或需要在中断上下文中发送消息，请使用usb\_submit\_urb（）。如果驱动程序中的线程使用此调用，请确保您的disconnect（）方法可以等待其完成。由于您没有URB的句柄，因此无法取消请求。

**返回**

如果成功，为0。否则，是一个负错误号码。实际传输的字节数将存储在actual\_length参数中。

#### int usb\_bulk\_msg(struct usb\_device \*usb\_dev，unsigned int pipe，void \*data，int len，int \*actual\_length，int timeout)

构建一个大量urb，发送它并等待完成

**参数**

struct usb\_device \*usb\_dev

指向要发送消息的USB设备的指针

unsigned int pipe

要发送消息的端点“管道”

void \*data

要发送的数据的指针

int len

要发送的数据的字节数

int \*actual\_length

将传输的实际长度（字节）放置到的位置的指针

int timeout

在超时之前等待消息完成的时间（如果为0，则永远等待）

**上下文**

！在中断（）中

**说明**

此函数向指定的端点发送简单的大容量消息，并等待消息完成或超时。

不要在中断上下文中使用此函数。如果您需要异步消息或需要在中断上下文中发送消息，请使用usb\_submit\_urb（）。如果驱动程序中的线程使用此调用，请确保您的disconnect（）方法可以等待其完成。由于您没有URB的句柄，因此无法取消请求。

因为没有usb\_interrupt\_msg（）和USBDEVFS\_INTERRUPT ioctl，所以用户被迫通过使用它来提交用于中断端点的URB来滥用这个例程。如果目标是中断端点，则我们将有权创建一个中断URB（使用默认间隔）。

**返回**

如果成功，为0。否则，是一个负错误号码。实际传输的字节数将存储在actual\_length参数中。

#### int usb\_sg\_init（struct usb\_sg\_request \*io，struct usb\_device \*dev，unsigned pipe，unsigned period，struct scatterlist \*sg，int nents，size\_t length，gfp\_t mem\_flags）

初始化基于分散列表的批量/中断I / O请求

**参数**

struct usb\_sg\_request \*io

正在初始化的请求块。直到usb\_sg\_wait（）返回，将此视为指向一个不透明内存块的指针，

struct usb\_device \*dev

将发送或接收数据的USB设备

unsigned pipe

用于传输数据的端点“管道”

unsigned period

中断端点的轮询速率，以帧或（对于高速端点）微帧为单位；对于大容量不起作用

struct scatterlist \*sg

分散列表条目

int nents

分散列表中的条目数

size\_t length

从分散列表发送多少字节，或零以发送在列表中标识的每个字节。

gfp\_t mem\_flags

影响此调用中内存分配的SLAB\_\*标志

**说明**

这将初始化分散/聚集请求，分配资源，如I / O映射和urb内存（除了可能由USB控制器驱动程序使用的内存）。

请求必须使用usb\_sg\_wait（）发出，该函数等待I / O完成（或被取消），然后清理usb\_sg\_init（）分配的所有资源。

请求可以使用usb\_sg\_cancel（）在调用usb\_sg\_wait（）之前或之后取消。

**返回**

成功返回零，否则返回一个负的errno值。

#### void usb\_sg\_wait（struct usb\_sg\_request \*io）

同步执行分散/聚集请求

**参数**

struct usb\_sg\_request \*io

请求块句柄，如使用usb\_sg\_init（）初始化。此调用返回时，某些字段变得可访问。

**上下文**

任务上下文，可能会睡眠。

**说明**

此函数阻塞，直到指定的I / O操作完成。它利用相关I / O请求的分组来获得良好的传输速度，通过排队请求。在更高的速度下，这样的排队可以显著提高USB吞吐量。

该函数有三种完成方式。

成功，其中io->status为零。传输的字节数与请求的相同。

错误，其中io->status为负errno值。错误发生前传输的字节数通常小于请求的，可以是非零。

取消，一种带有状态-ECONNRESET的错误，由usb\_sg\_cancel()发起。

当此函数返回时，通过usb\_sg\_init()或此调用分配的所有内存都已被释放。请求块参数仍然可以传递给usb\_sg\_cancel()，也可以被释放。也可以重新初始化然后重复使用。

数据传输速率

Bulk传输适用于全速或高速端点。最佳全速数据传输速率为每帧19个64字节的包，或每毫秒1216字节。最佳高速数据传输速率为每微帧13个512字节的包或每毫秒52 KB。

通过这个API使用中断传输的原因很可能是为了保留高速带宽，其中可以传输高达每毫秒24 KB。对于低速或全速中断端点而言，这种能力不太有用，它们最多允许每毫秒一个包，最大8或64字节（分别）。

对于位于xHCI主机控制器下的设备，不必调用此函数以保留带宽，因为当选择配置或接口备选设置时，会保留带宽。

#### void usb\_sg\_cancel(struct usb\_sg\_request \*io)

停止由usb\_sg\_wait()发出的散列/聚集I/O

**参数**

struct usb\_sg\_request \*io

请求块，使用usb\_sg\_init()进行初始化

**说明**

在由usb\_sg\_wait()启动请求后，这将停止请求。它也可以防止一个由usb\_sg\_init()初始化的请求启动，因此该调用只会释放分配给请求的资源。

#### int usb\_get\_descriptor(struct usb\_device \*dev, unsigned char type, unsigned char index, void \*buf, int size)

发出通用GET\_DESCRIPTOR请求

**参数**

struct usb\_device \*dev

正在检索说明符的设备

unsigned char type

说明符类型（USB\_DT\_\*）

unsigned char index

说明符编号

void \*buf

将说明符放置在哪里

int size

“buf”有多大？

**上下文**

!in\_interrupt()

**说明**

获取USB说明符。存在一些方便的函数以简化获取某些类型的说明符。对于USB\_DT\_STRING，请使用usb\_get\_string()或usb\_string()。设备（USB\_DT\_DEVICE）和配置说明符（USB\_DT\_CONFIG）属于设备结构。除了一些USB标准说明符外，一些设备还使用类特定或供应商特定说明符。

此调用是同步的，不能在中断上下文中使用。

**返回**

成功接收的字节数，或下层usb\_control\_msg()调用返回的状态代码。

#### int usb\_string(struct usb\_device \*dev, int index, char \*buf, size\_t size)

返回字符串说明符的UTF-8版本

**参数**

struct usb\_device \*dev

正在检索字符串说明符的设备

int index

说明符编号

char \*buf

将字符串放置在哪里

size\_t size

“buf”有多大？

**上下文**

!in\_interrupt ()

**说明**

将设备返回的UTF-16LE编码的字符串从usb\_get\_string\_descriptor()转换为可在大多数内核上下文中更易于使用的以空字符结尾的UTF-8编码字符串。请注意，此函数选择设备支持的第一种语言的字符串。

此调用是同步的，不能在中断上下文中使用。

**返回**

字符串长度（> = 0）或usb\_control\_msg状态（<0）。

#### int usb\_get\_status(struct usb\_device \*dev, int recip, int type, int target, void \*data)

发出GET\_STATUS调用

**参数**

struct usb\_device \*dev

正在检查状态的设备

int recip

USB\_RECIP\_\*;对于设备，接口或端点

int type

USB\_STATUS\_TYPE\_\*;对于标准或PTM状态类型

int target

零（对于设备），否则是接口或端点编号

void \*data

指向两个字节位图数据的指针

**上下文**

!in\_interrupt()

**说明**

返回设备，接口或端点状态。通常只关注查看设备是自供电的，还是已启用远程唤醒功能；或者是否批量或中断端点停止（“停滞”）。

这些状态位图中的比特是使用SET\_FEATURE请求设置并使用CLEAR\_FEATURE请求清除的。应使用usb\_clear\_halt()函数清除停滞（“停滞”）状态。

此调用是同步的，不能在中断上下文中使用。

返回成功并在\* data（以主机字节顺序）中返回状态值的0或状态代码（受底层usb\_control\_msg()调用的状态代码）。

#### int usb\_clear\_halt(struct usb\_device \*dev, int pipe)

告诉设备清除端点停滞/停滞条件

**参数**

struct usb\_device \*dev

停止的设备端点

int pipe

清除的端点“pipe”

**上下文**

!in\_interrupt ()

**说明**

这用于清除由URB完成状态报告的批量和中断端点的停滞条件。被停滞的端点有时被称为“停滞”。这样的端点无法传输或接收数据，直到清除停滞状态。在清除停滞条件之前，驱动程序应将这样的端点排队的任何URB正常取消链接，如USB 2.0规范的第5.7.5和5.8.5节中所述。

请注意，控制和等时端点不会停止，尽管控制端点会报告“协议停止”（用于不支持的请求），使用与报告真实停止相同的状态代码。

此调用是同步的，不可在中断上下文中使用。

**返回**

成功时为零，否则为底层usb\_control\_msg()调用返回的状态代码。

#### void usb\_reset\_endpoint(struct usb\_device \*dev, unsigned int epaddr)

重置端点状态。

**参数**

struct usb\_device \*dev

要重置其端点的设备

unsigned int epaddr

端点的地址。 输出端点号，输入端点号+USB\_DIR\_IN

**说明**

重置主机端的端点状态，例如切换位、序列号或当前窗口。

#### int usb\_set\_interface(struct usb\_device \*dev, int interface, int alternate)

使特定的备用设置成为当前设置

**参数**

struct usb\_device \*dev

正在更新其接口的设备

int interface

正在更新的接口

int alternate

正在选择的设置。

**上下文**

!in\_interrupt()

**说明**

这用于启用可能默认未启用的接口上的数据传输。并非所有设备都支持这种可配置性。只有绑定到接口的驱动程序才能更改其设置。

在任何给定的配置中，每个接口可以有几种备选设置。这些通常用于控制带宽消耗的级别。例如，高速中断端点的默认设置可能不会在每个微帧中发送超过64个字节，而每个微帧中最多可发送3KBytes的中断传输是合法的。此外，等时端点可能永远不会成为接口的默认设置的一部分。要访问此种带宽，必须使当前备用接口设置。

请注意，在Linux USB子系统中，与给定备用设置中的端点相关联的带宽不会在需要该带宽的URB提交之前保留。一些其他操作系统会在选择配置时提前分配带宽。

xHCI保留带宽并在usb\_hcd\_alloc\_bandwidth()中配置备用设置。如果它失败，原始接口altsetting可能会被禁用。驱动程序不能依赖任何特定的备用设置在失败后生效。

此调用是同步的，并且不能在中断上下文中使用。此外，驱动程序不能在计划在该接口的端点上的URB正在进行时更改altsettings；所有这样的URB都必须首先完成（可能通过取消链接来强制完成）。

**返回**

成功时为零，否则为底层usb\_control\_msg()调用返回的状态代码。

#### int usb\_reset\_configuration(struct usb\_device \*dev)

轻量级设备复位

**参数**

struct usb\_device \*dev

正在重置其配置的设备

**说明**

这会使用当前配置向设备发出标准的SET\_CONFIGURATION请求。其效果是重置设备中的大多数USB相关状态，包括接口altsetting（重置为零）、端点停止（清除）和端点状态（仅适用于块和中断端点）。其他usbcore状态保持不变，包括usb设备驱动程序与接口的绑定。

因为这会影响多个接口，因此避免在复合（多接口）设备上使用此功能。相反，每个接口的驱动程序可以在它所声明的接口上使用usb\_set\_interface()。但要小心；有些设备不支持SET\_INTERFACE请求，而其他设备不会重置所有接口状态（尤其是端点状态）。重置整个配置将影响其他驱动程序的接口。

调用者必须拥有设备锁定。

**返回**

成功时为零，否则为负的错误代码。

#### int usb\_driver\_set\_configuration(struct usb\_device \*udev, int config)

为驱动程序提供更改设备配置的方法

**参数**

struct usb\_device \*udev

正在更新其配置的设备

int config

正在选择的配置。

**上下文**

在进程上下文中，必须能够休眠

**说明**

不允许设备接口驱动程序更改设备配置。这是因为更改配置将销毁驱动程序绑定的接口并创建新接口；这就好像软盘驱动程序告诉计算机用磁带驱动器替换软盘驱动器！

尽管如此，在某些特殊情况下可能需要这样做。此例程通过使用工作线程提出更改配置请求，绕过了通常的限制。

**返回**

如果成功排队请求，则为0，否则为错误代码。呼叫者无法知道排队的请求最终是否会成功。

#### int cdc\_parse\_cdc\_header(struct usb\_cdc\_parsed\_header \* hdr，struct usb\_interface \* intf，u8 \* buffer，int buflen)

解析CDC设备中存在的额外标头

**参数**

struct usb\_cdc\_parsed\_header \*hdr

将解析结果放置的位置

struct usb\_interface \*intf

请求解析的接口

u8 \*buffer

要解析的额外标头的指针

int buflen

额外标头的长度

**说明**

对于绑定数据和控制的CDC设备，评估存在的额外标头并提供有关设备功能的详细信息。

**返回**

解析的说明符数量，如果头文件矛盾无法修复，则返回-EINVAL

#### int usb\_register\_dev（struct usb\_interface \* intf，struct usb\_class\_driver \* class\_driver）

注册USB设备，并请求次要号码

**参数**

struct usb\_interface \* intf

被注册的usb\_interface的指针

struct usb\_class\_driver \* class\_driver

此设备的usb\_class\_driver的指针

**说明**

所有使用USB主要号码的USB驱动程序都应调用此函数。如果启用了CONFIG\_USB\_DYNAMIC\_MINORS，则次要号码将动态分配，从可用号码列表中分配。如果未启用，则次要号码将基于class\_driver ->的下一个可用空闲次要号码。起始minor\_base。

此函数还在sysfs树中创建了一个USB类设备。

当驱动程序完成这个函数给出的次要号码时，必须调用usb\_deregister\_dev()。

**返回**

如果在尝试注册设备时发生错误，则返回-EINVAL，成功则返回0。

#### void usb\_deregister\_dev（struct usb\_interface \* intf，struct usb\_class\_driver \* class\_driver）

注销USB设备的动态小型号。

**参数**

struct usb\_interface \* intf

被注销的usb\_interface的指针

struct usb\_class\_driver \* class\_driver

此设备的usb\_class\_driver的指针

**说明**

与usb\_register\_dev（）配合使用。当USB驱动程序使用从调用usb\_register\_dev（）获取的次要号码（通常在从系统断开连接设备时）时，调用此函数。

此函数还从sysfs树中删除了usb类设备。

所有使用USB主要号码的驱动程序都应调用此函数。

#### int usb\_driver\_claim\_interface（struct usb\_driver \* driver，struct usb\_interface \* iface，void \* data）

将驱动程序绑定到接口

**参数**

struct usb\_driver \* driver

要绑定的驱动程序

struct usb\_interface \* iface

将要绑定的接口；必须在usb设备的活动配置中

void \* data

与该接口相关的驱动程序数据

**说明**

这是由需要在探测时声明设备上的多个接口的usb设备驱动程序使用的（音频和acm是当前的示例）。任何设备驱动程序都不应直接修改内部usb\_interface或usb\_device结构成员。

调用者必须拥有设备锁，因此驱动程序探测（）条目不需要额外的锁定，但其他调用上下文可能需要明确声明该锁。

**返回**

成功时返回0。

#### void usb\_driver\_release\_interface（struct usb\_driver \* driver，struct usb\_interface \* iface）

从接口解除绑定驱动程序

**参数**

struct usb\_driver \* driver

要取消绑定的驱动程序

struct usb\_interface \* iface

将要解除绑定的接口

**说明**

驱动程序可以使用此功能释放接口，而无需等待其断开连接（）方法被调用。在典型情况下，这也会导致驱动程序断开（）方法被调用。

此调用是同步的，不能在中断上下文中使用。调用者必须拥有设备锁，因此驱动程序断开（）条目不需要额外的锁定，但其他调用上下文可能需要明确声明该锁。

#### const struct usb\_device\_id \* usb\_match\_id（struct usb\_interface \* interface，const struct usb\_device\_id \* id）

找到与设备或接口匹配的第一个usb\_device\_id

**参数**

struct usb\_interface \* interface

感兴趣的接口

const struct usb\_device\_id \* id

由零条目终止的usb\_device\_id结构数组

**说明**

usb\_match\_id搜索usb\_device\_id数组并返回与设备或接口匹配的第一个usb\_device\_id或null。当将驱动程序绑定（或重新绑定）到接口时使用。大多数USB设备驱动程序将间接使用此函数，通过usb核心，但某些分层驱动程序框架直接使用它。这些设备表通过modutils使用MODULE\_DEVICE\_TABLE导出，以支持USB热插拔的驱动程序加载功能。

什么可以匹配?

“match\_flags”元素控制使用哪些成员的usb\_device\_id。如果设置了相应的位，则设备或接口说明符中的值必须与其相应的成员匹配，否则device\_id不匹配。

“driver\_info”通常仅由设备驱动程序使用，但如果提供了具有非零“driver\_info”字段的id，则可以创建通配符“匹配任何内容”的usb\_device\_id作为驱动程序的“modules.usbmap”条目。如果这样做，则USB设备驱动程序的probe（）例程应使用其他智能来决定是否绑定到指定的接口。

什么可以使良好的usb\_device\_id表

匹配算法非常简单，因此驱动程序选择中的智能必须来自智能驱动程序id记录。除非您有使用其他选择策略的充分理由，否则仅提供相关组中的匹配元素，并从具体到一般的顺序匹配说明符。如果可以，则使用为此目的提供的宏。

最特定的匹配说明符使用设备说明符数据。这些通常与特定产品的匹配一起使用；USB\_DEVICE宏允许您提供供应商和产品ID，并且还可以针对产品修订号范围进行匹配。这些广泛用于具有应用程序或供应商特定的bDeviceClass值的设备。

基于设备类/子类/协议规范的匹配略微更加通用；请使用USB\_DEVICE\_INFO宏或其类似项。它们适用于单功能设备，其中bDeviceClass未指定每个接口具有自己的类别。

基于接口类/子类/协议的匹配是最通用的；它们允许驱动程序绑定到多功能设备上的任何接口。请使用USB\_INTERFACE\_INFO宏或其类似项来匹配按界面分类的样式设备（如bInterfaceClass中记录的）。

请注意，由USB\_INTERFACE\_INFO创建的条目将不会匹配任何接口，如果设备类设置为供应商特定，则这是有意的。这是故意的；根据USB规范，这些设备的界面类/子类/协议的含义也是供应商特定的，因此对标准产品类别的匹配也无法工作。如果您真的想为这样的设备使用基于接口的匹配，请创建一个还指定供应商ID的匹配记录。 （不幸的是，没有用于创建此类记录的标准宏。）

在这些组内，记住并非所有组合都有意义。例如，不要指定产品版本范围而没有供应商和产品ID；或指定协议而没有关联的类和子类。

#### int usb\_register\_device\_driver（struct usb\_device\_driver \* new\_udriver，struct module \* owner）

注册USB设备（非接口）驱动程序

**参数**

struct usb\_device\_driver \* new\_udriver

设备驱动程序的USB操作

struct module \* owner

此驱动程序的模块所有者。

**说明**

向USB核心注册USB设备驱动程序。每当添加新驱动程序时，未连接设备的列表将被重新扫描，从而允许新驱动程序连接到任何已识别的设备。

**返回**

失败时为负错误代码，成功时为0。

#### void usb\_deregister\_device\_driver（struct usb\_device\_driver \* udriver）

注销USB设备（非接口）驱动程序

**参数**

struct usb\_device\_driver \* udriver

要注销的设备驱动程序的USB操作

**上下文**

必须能够睡眠

**说明**

从内部USB驱动程序列表中取消链接指定的驱动程序。

#### int usb\_register\_driver（struct usb\_driver \* new\_driver，struct module \* owner，const char \* mod\_name）

注册USB接口驱动程序

**参数**

struct usb\_driver \* new\_driver

接口驱动程序的USB操作

struct module \* owner

此驱动程序的模块所有者。

const char \* mod\_name

模块名称字符串

**说明**

向USB核心注册USB接口驱动程序。每当添加新驱动程序时，未连接接口的列表将被重新扫描，从而允许新驱动程序连接到任何已识别的接口。

**返回**

失败时为负错误代码，成功时为0。

**注意**

如果您想让驱动程序使用USB主编号，您必须调用usb\_register\_dev（）以启用该功能。此功能不再处理。

#### void usb\_deregister（struct usb\_driver \* driver）

注销USB接口驱动程序

**参数**

struct usb\_driver \* driver

要注销的接口驱动程序的USB操作

**上下文**

必须能够睡眠

**说明**

从内部USB驱动程序列表中取消链接指定的驱动程序。

**注意**

如果调用了usb\_register\_dev（），您仍然需要调用usb\_deregister\_dev（）来清理驱动程序分配的次要编号，此\*调用不再为您执行该操作。

#### void usb\_enable\_autosuspend（struct usb\_device \* udev）

允许USB设备自动挂起

**参数**

struct usb\_device \* udev

可以自动挂起的USB设备

**说明**

此例程允许udev自动挂起。自动挂起不会在经过autosuspend\_delay并且满足所有其他必要条件之前发生。

调用者必须持有udev的设备锁定。

#### void usb\_disable\_autosuspend（struct usb\_device \* udev）

防止USB设备自动挂起

**参数**

struct usb\_device \* udev

不得自动挂起的USB设备

**说明**

此例程会防止udev自动挂起，并在已自动挂起时唤醒它。

调用者必须持有udev的设备锁定。

#### void usb\_autopm\_put\_interface（struct usb\_interface \* intf）

减少USB接口的PM使用计数器

**参数**

struct usb\_interface \* intf

使用计数器应该减少的usb\_interface

**说明**

当接口驱动程序完成使用intf并希望允许它自动挂起时，应调用此例程。典型例子是字符设备驱动程序在其设备文件关闭时。

该例程会减少intf的使用计数器。当计数器达到0时，将尝试为intf的设备安排延迟自动挂起请求。尝试可能失败（请参见autosuspend\_check（））。

此例程仅可在进程上下文中运行。

#### void usb\_autopm\_put\_interface\_async（struct usb\_interface \* intf）

减少USB接口的PM使用计数器

**参数**

struct usb\_interface \* intf

使用计数器应该减少的usb\_interface

**说明**

此例程执行的操作与usb\_autopm\_put\_interface（）类似它减少intf的使用计数器，并在计数器<= 0时安排延迟自动挂起请求。不同之处在于它不执行任何同步;调用者应持有私有锁定并自行处理所有同步问题。

通常情况下，如果没有正在等待的 URB（USB 请求块），驱动程序会在 URB 的完成处理程序中调用此例程。

此例程可以在原子上下文中运行。

#### void usb\_autopm\_put\_interface\_no\_suspend（struct usb\_interface \*intf）

递减 USB 接口的 PM-使用计数器

**参数**

struct usb\_interface \*intf

应递减计数器的 USB 接口

**说明**

此例程递减 intf 的 usage 计数器，但不执行自动挂起。

此例程可以在原子上下文中运行。

#### int usb\_autopm\_get\_interface（struct usb\_interface \*intf）

增加 USB 接口的 PM-使用计数器

**参数**

struct usb\_interface \*intf

应递增计数器的 USB 接口

**说明**

当接口驱动程序想要使用 intf 并需要保证它没有被挂起时，应调用此例程。此外，该例程防止 intf 随后被自动挂起。（请注意，此不会防止源于 PM 核心的挂起事件。）此预防措施将持续到调用usb\_autopm\_put\_interface（）或解除绑定 intf。一个典型的例子是当字符设备驱动程序打开其设备文件时。

增加 intf 的 usage 计数器以防止后续自动挂起。但是，如果自动恢复失败，计数器将被重新递减。

此例程只能在进程上下文中运行。

**返回**

成功返回0。

#### int usb\_autopm\_get\_interface\_async（struct usb\_interface \*intf）

增加 USB 接口的 PM-使用计数器

**参数**

struct usb\_interface \*intf

应递增计数器的 USB 接口

**说明**

此例程与usb\_autopm\_get\_interface（）基本相同它增加intf的使用计数器，并在设备挂起时排队一个自动恢复请求。但它们之间的区别是它不执行任何同步（调用方应该持有私有锁并自行处理所有同步问题），并且它不直接自动恢复设备（它仅排队请求）。成功调用后，设备可能尚未恢复。

此例程可以在原子上下文中运行。

**返回**

成功返回0。否则，返回负误差代码。

#### void usb\_autopm\_get\_interface\_no\_resume（struct usb\_interface \*intf）

增加 USB 接口的 PM-使用计数器

**参数**

struct usb\_interface \*intf

应递增计数器的 USB 接口

**说明**

此例程增加 intf 的 usage 计数器，但不执行自动恢复。

此例程可以在原子上下文中运行。

#### int usb\_find\_common\_endpoints（struct usb\_host\_interface \*alt，struct usb\_endpoint\_descriptor \*\*bulk\_in，struct usb\_endpoint\_descriptor \*\*bulk\_out，struct usb\_endpoint\_descriptor \*\*int\_in，struct usb\_endpoint\_descriptor \*\*int\_out）

查找常见端点**说明**符

**参数**

struct usb\_host\_interface \*alt

要搜索的备用设置

struct usb\_endpoint\_descriptor \*\*bulk\_in

说明符指针的指针，或为 NULL

struct usb\_endpoint\_descriptor \*\*bulk\_out

说明符指针的指针，或为 NULL

struct usb\_endpoint\_descriptor \*\*int\_in

说明符指针的指针，或为 NULL

struct usb\_endpoint\_descriptor \*\*int\_out

说明符指针的指针，或为 NULL

**说明**

搜索备用设置的端点说明符，返回第一个批量输入，批量输出，中断输入和中断输出端点的指针（除非它们为 NULL）。

如果找不到请求的端点，则将相应的指针设置为 NULL。

**返回**

如果找到所有请求的说明符则返回零，否则返回-ENXIO。

#### int usb\_find\_common\_endpoints\_reverse（struct usb\_host\_interface \*alt，struct usb\_endpoint\_descriptor \*\*bulk\_in，struct usb\_endpoint\_descriptor \*\*bulk\_out，struct usb\_endpoint\_descriptor \*\*int\_in，struct usb\_endpoint\_descriptor \*\*int\_out）

查找常见端点说明符

**参数**

struct usb\_host\_interface \*alt

要搜索的备用设置

struct usb\_endpoint\_descriptor \*\*bulk\_in

说明符指针的指针，或为 NULL

struct usb\_endpoint\_descriptor \*\*bulk\_out

说明符指针的指针，或为 NULL

struct usb\_endpoint\_descriptor \*\*int\_in

说明符指针的指针，或为 NULL

struct usb\_endpoint\_descriptor \*\*int\_out

说明符指针的指针，或为 NULL

**说明**

搜索备用设置的端点说明符，返回最后一个批量输入，批量输出，中断输入和中断输出端点的指针（除非它们为 NULL）。

如果找不到请求的端点，则将相应的指针设置为 NULL。

**返回**

如果找到所有请求的说明符则返回零，否则返回-ENXIO。

#### struct usb\_host\_interface \*usb\_find\_alt\_setting（struct usb\_host\_config \*config，unsigned int iface\_num，unsigned int alt\_num）

给定配置，查找给定接口的替代设置。

**参数**

struct usb\_host\_config \*config

要搜索的配置（不一定是当前配置）。

unsigned int iface\_num

要搜索的接口号

unsigned int alt\_num

要搜索的替代接口设置编号。

**说明**

在配置的接口缓存中查找指定的 alt 设置。

**返回**

如果找到，则为替代设置；否则为 NULL。

#### struct usb\_interface \*usb\_ifnum\_to\_if（const struct usb\_device \*dev，unsigned ifnum）

使用给定的接口号获取接口对象。

**参数**

const struct usb\_device \*dev

考虑当前配置的设备

unsigned ifnum

所需接口

**说明**

这将遍历当前活动配置的设备说明符，以查找具有特定接口号的接口对象。

注意，配置说明符不需要顺序分配接口号，因此假设该说明符中的第一个接口对应于接口零是错误的。此例程可帮助设备驱动程序避免此类错误。但是，请确保您对此接口可用的所有备用设置执行正确操作。

除非您绑定到该设备的一个接口或锁定该设备，否则请勿调用此函数！

**返回**

找到 ifnum 作为接口号的接口的指针。否则返回 NULL。

#### struct usb\_host\_interface \*usb\_altnum\_to\_altsetting(const struct usb\_interface \*intf, unsigned int altnum)

获取给定备用设置号的 altsetting 结构。

**参数**

const struct usb\_interface \*intf

包含所需 altsetting 的接口

unsigned int altnum

所需备用设置编号

**说明**

这将搜索指定接口的 altsetting 数组以找到具有正确 bAlternateSetting 值的条目。

请注意，备用设置无需按数字顺序存储，因此假定数组中的第一个备用设置条目对应于备用设置零是不正确的。这个例程可帮助设备驱动程序避免这些错误。

除非您绑定到 intf 接口或锁定该设备，否则请勿调用此函数！

**返回**

返回 intf 的 altsetting 数组的 entry 指针，其中 altnum 是备用设置编号。如果找不到，则返回 NULL。

#### struct usb\_interface \*usb\_find\_interface(struct usb\_driver \*drv, int minor)

查找驱动程序和设备的 usb\_interface 指针

**参数**

struct usb\_driver \*drv

考虑其当前配置的驱动程序

int minor

所需设备的次要编号

**说明**

这将遍历总线设备列表并返回具有匹配次要和驱动程序的接口指针。请注意，这仅适用于共享 USB 主要编号的设备。

**返回**

具有匹配主要和次要编号的接口的指针。

#### int usb\_for\_each\_dev(void \*data, int (\*fn)(struct usb\_device\*, void\*))

遍历系统中的所有 USB 设备

**参数**

void \*data

将传递给回调函数的数据指针

int (\*fn)(struct usb\_device \*, void \*)

每个 USB 设备要调用的回调函数

**说明**

遍历所有 USB 设备并为每个调用 fn，向其传递数据。如果它返回除 0 以外的任何内容，则我们将提前中止迭代并返回该值。

#### struct usb\_device \*usb\_alloc\_dev(struct usb\_device \*parent, struct usb\_bus \*bus, unsigned port1)

USB 设备构造函数（仅限于 usbcore）

**参数**

struct usb\_device \*parent

连接设备的集线器；分配根集线器的 null

struct usb\_bus \*bus

用于访问设备的总线

unsigned port1

端口的基于 1 的索引；对于根集线器则忽略

**上下文**

任务上下文，可能会休眠。

**说明**

只有集线器驱动程序（包括主控制器的虚拟根集线器驱动程序）才能调用此函数。

在非睡眠上下文中可能不使用此调用。

**返回**

成功时，指向分配的 USB 设备的指针。失败时为 NULL。

#### struct usb\_device \*usb\_get\_dev(struct usb\_device \*dev)

增加对 USB 设备结构的引用计数

**参数**

struct usb\_device \*dev

正在引用的设备

**说明**

对设备的每个实时引用都应该进行引用计数。

USB 接口的驱动程序通常应该在它们绑定到接口时在其过程()方法中记录这样的引用，并通过调用usb\_put\_dev()来释放它们，在其disconnect()方法中。但是，如果驱动程序在其 disconnect() 方法返回后不访问 usb\_device 结构，则不需要进行 refcounting，因为USB核心保证 usb\_device 将在取消绑定其所有接口驱动程序之后才会被解除分配。

**返回**

计数器增加后的设备指针。

#### void usb\_put\_dev(struct usb\_device \*dev)

释放对 USB 设备结构的使用权

**参数**

struct usb\_device \*dev

已经断开连接的设备

**说明**

当设备的用户使用完毕时必须调用。当设备的最后一个用户调用此函数时，设备的内存将被释放。

#### struct usb\_interface \*usb\_get\_intf(struct usb\_interface \*intf)

增加对 USB 接口结构的引用计数

**参数**

struct usb\_interface \*intf

正在引用的接口

**说明**

每个实时引用的接口都必须进行引用计数。

USB 接口的驱动程序通常应该在它们绑定到接口时在其过程()方法中记录这样的引用，并通过调用usb\_put\_intf()在其disconnect()方法中释放它们。但是，如果驱动程序在其 disconnect() 方法返回后不访问 usb\_interface 结构，则不需要进行 refcounting，因为 USB 核心保证 usb\_interface 将在其驱动程序取消绑定后才被解除分配。

**返回**

计数器增加后的接口指针。

#### void usb\_put\_intf(struct usb\_interface \*intf)

释放对 USB 接口结构的使用权

**参数**

struct usb\_interface \*intf

已经减少的接口

**说明**

当接口的用户使用完毕时必须调用。当接口的最后一个用户调用此函数时，接口的内存将被释放。

#### int usb\_lock\_device\_for\_reset(struct usb\_device \*udev, const struct usb\_interface \*iface)

谨慎地获取USB设备struct 的锁定

**参数**

struct usb\_device \*udev

被锁定的设备

const struct usb\_interface \*iface

绑定到请求的驱动程序的界面（可选）

**说明**

试图获取设备锁定，但如果设备为NOTATTACHED或SUSPENDED，或如果iface被指定并且接口既不是BINDING也不是BOUND，则失败。该函数反复轮询而不是睡眠等待锁定。这是为了防止与断开连接的死锁；在某些驱动程序（例如usb-storage）中，disconnect（）或suspend（）方法将阻塞等待设备重置完成。

**返回**

失败为负错误代码，否则为0。

#### int usb\_get\_current\_frame\_number（struct usb\_device \*dev）

返回当前总线帧数

**参数**

struct usb\_device \*dev

询问总线的设备

**返回**

用于给定USB设备使用的USB主机控制器的当前帧数。这可以在调度等时请求中使用。

**注意**

不同类型的主机控制器具有不同的“调度地平线”。一种类型可能只支持将调度提前32帧，而其他类型则可以支持将调度提前到1024帧。

#### void \*usb\_alloc\_coherent（struct usb\_device \*dev、size\_t size、gfp\_t mem\_flags、dma\_addr\_t \*dma）

为URB\_NO\_xxx\_DMA\_MAP分配DMA一致性缓冲区

**参数**

struct usb\_device \*dev

将使用该缓冲区的设备

size\_t size

请求的缓冲区大小

gfp\_t mem\_flags

影响是否允许分配阻塞

dma\_addr\_t \*dma

用于返回缓冲区的DMA地址

**返回**

为Null（表示无法分配缓冲区）或CPU空间指针到缓冲区的返回。可以使用这样的cpu空间缓冲区来执行向指定设备的DMA。这些cpu空间缓冲区与DMA地址一起返回（通过提供的指针）。请注意这些缓冲区与URB\_NO\_xxx\_DMA\_MAP一起使用，以避免像使用“DMA跳跃缓冲区”，或在URB完成/重新提交期间使用IOMMU硬件等行为。在平台之间的实现因DMA如何在此设备上工作的详细信息而异。同样，在CPU缓存不是DMA一致的体系结构上使用这些缓冲区也消除了高速缓存线共享问题。在没有总线旁路缓存的系统中，这些缓冲区是无缓存的。

**说明**

不再使用该缓冲区时，请使用usb\_free\_coherent（）释放它。

#### void usb\_free\_coherent（struct usb\_device \*dev、size\_t size、void \*addr、dma\_addr\_t dma）

释放使用usb\_alloc\_coherent（）分配的内存

**参数**

struct usb\_device \*dev

使用缓冲区的设备

size\_t size

请求的缓冲区大小

void \*addr

缓冲区的CPU地址

dma\_addr\_t dma

缓冲区的DMA地址

**说明**

这将回收I/O缓冲区，使其可重复使用。必须使用usb\_alloc\_coherent（）分配内存，并且参数必须与那个分配请求中提供的参数匹配。

#### struct urb \*usb\_buffer\_map（struct urb \*urb）

为urb创建DMA映射

**参数**

struct urb \*urb

将映射transfer\_buffer/setup\_packet的urb

**说明**

如果操作成功，将URB\_NO\_TRANSFER\_DMA\_MAP添加到urb->transfer\_flags中。如果该设备通过非DMA控制器连接到此系统，则此操作始终成功。此调用通常用于作为大型定期传输目标重复使用的urb，使用usb\_buffer\_dmasync（）调用以同步内存和dma状态来实现。

使用usb\_buffer\_unmap（）撤消此调用的效果。

**返回**

为NULL（表示无法映射缓冲区）或urb。

#### void usb\_buffer\_dmasync（struct urb \*urb）

同步缓冲区的DMA和CPU视图

**参数**

struct urb \*urb

将同步transfer\_buffer/setup\_packet的urb

#### void usb\_buffer\_unmap（struct urb \*urb）

释放URB的DMA映射

**参数**

struct urb \*urb

将卸载transfer\_buffer的urb

**说明**

撤消usb\_buffer\_map（）的效果。

#### int usb\_buffer\_map\_sg（const struct usb\_device \*dev、int is\_in、struct scatterlist \*sg、int nents）

为端点创建scatterlist DMA映射

**参数**

const struct usb\_device \*dev

将scatterlist映射到其中的设备

int is\_in

映射传输方向

struct scatterlist \*sg

要映射的scatterlist

int nents

scatterlist中的条目数

**返回**

为负（表示无法映射缓冲区）或scatterlist中的DMA映射数组条目数。

**注意**

调用者负责将来自scatterlist的DMA地址放入URB传输缓冲区指针中，并为这些URB中的每个设置URB\_NO\_TRANSFER\_DMA\_MAP传输标志。最好使用usb\_sg\_\*（）调用，它们为您完成这个（和更多）。

通过排队URB而不是在开始下一个I/O之前等待每个URB完成，可以获得最高的I/O速率。这特别容易使用scatterlists完成。只需为返回的每个DMA映射条目分配并提交一个URB，在第一个错误或全部成功时停止。更好的做法是使用usb\_sg\_\*（）调用，它会为您完成这个（和更多）。因为使用IOMMU的一些硬件上，因为硬件会尝试将分配给单个DMA操作的缓冲区合并成单个DMA映射，所以通常使用此调用来翻译scatterlist请求，而不是使用usb\_buffer\_map（）。

使用usb\_buffer\_unmap\_sg()来撤消此调用的效果。

#### void usb\_buffer\_dmasync\_sg(const struct usb\_device \* dev, int is\_in, struct scatterlist \* sg, int n\_hw\_ents)

同步散列列表缓冲区的DMA和CPU视图

**参数**

const struct usb\_device \* dev

要映射散列列表的设备

int is\_in

映射传输方向

struct scatterlist \* sg

要同步的散列列表

int n\_hw\_ents

来自usb\_buffer\_map\_sg的正返回

**说明**

当您重新使用散列列表的数据缓冲区进行另一个USB请求时，请使用此选项。

#### void usb\_buffer\_unmap\_sg(const struct usb\_device \* dev, int is\_in, struct scatterlist \* sg, int n\_hw\_ents)

释放一个散列列表的DMA映射。

**参数**

const struct usb\_device \* dev

要映射散列列表的设备

int is\_in

映射传输方向

struct scatterlist \* sg

要解除映射的散列列表

int n\_hw\_ents

来自usb\_buffer\_map\_sg的正返回

**说明**

撤消usb\_buffer\_map\_sg()的效果。

#### int usb\_hub\_clear\_tt\_buffer(struct urb \*urb)

清除高速USB鼠标中的控制/批量TT状态

**参数**

struct urb \*urb

与失败或不完整的分裂事务相关联的URB

**说明**

高速HCD使用此方法告诉集线器驱动程序，某些控制或批量分割事务失败，需要清除事务转换器的内部状态。这通常是从中断上下文中检测到的（并报告的）。

在完全清除该状态之前，该集线器可能无法处理其他完整（或低速）事务。

**返回**

成功则返回0，否则为负错误代码。

#### void usb\_set\_device\_state(struct usb\_device \*udev, enum usb\_device\_state new\_state)

更改设备的当前状态（usbcore，hcds）

**参数**

struct usb\_device \*udev

应更改其状态的设备的指针

enum usb\_device\_state new\_state

要存储的新状态值

**说明**

udev->state不完全受设备锁保护。虽然大多数转换仅在持有锁定时进行，但状态随时可以更改为USB\_STATE\_NOTATTACHED。这是为了使设备尽可能快地标记为断开连接，而无需等待任何信号量的释放。因此，必须通过device\_state\_lock自旋锁来保护对任何设备状态的所有更改。

添加到设备树后，应使用此例程进行对其状态的所有更改。不应直接设置状态。

如果udev->state已经为USB\_STATE\_NOTATTACHED，则不进行更改。否则，设置udev->state为new\_state，如果new\_state为USB\_STATE\_NOTATTACHED，则还将设置udev的所有子项的状态为USB\_STATE\_NOTATTACHED。

#### void usb\_root\_hub\_lost\_power(struct usb\_device \*rhdev)

如果根集线器丢失Vbus电源，则由HCD呼叫

**参数**

struct usb\_device \*rhdev

根集线器的struct usb\_device

**说明**

USB主机控制器驱动程序在恢复其根集线器并且Vbus电源已中断或控制器已重置时调用此函数。该例程将rhdev标记为已失去电源。当集线器驱动程序恢复时，它会注意到并执行所有启用“USB-PERSIST”的子设备的电源会话恢复；其他设备将断开连接。

#### int usb\_reset\_device(struct usb\_device \*udev)

警告接口驱动程序并执行USB端口复位

**参数**

struct usb\_device \*udev

要复位的设备（不处于NOTATTACHED状态）

**说明**

警告绑定到已注册界面的所有驱动程序（使用其pre\_reset方法），执行端口重置，然后让驱动程序知道重置已结束（使用其post\_reset方法）。

如果接口当前正在被探查或断开连接，则假定其驱动程序知道如何处理重置。对于所有其他接口，如果驱动程序没有pre\_reset和post\_reset方法，则我们尝试解除其绑定，并在重置后重新绑定。

**返回**

与usb\_reset\_and\_verify\_device()相同。但是，如果重置已经在进行中（例如，如果驱动程序没有pre\_reset（）或post\_reset()回调，并且在正在进行的重置期间，在进行重置时，它的disconnect（）或probe()例程尝试执行第二个嵌套重置），则该例程返回-EINPROGRESS。

**注意**

呼叫者必须拥有设备锁。例如，可以在从驱动程序probe()例程下载新固件后使用此选项。对于可能不会在probe()期间发生的调用，驱动程序应使用usb\_lock\_device\_for\_reset()锁定设备。

如果接口当前正在被探查或断开连接，则假定其驱动程序知道如何处理重置。对于所有其他接口，如果驱动程序没有pre\_reset和post\_reset方法，则我们尝试解除其绑定，并在重置后重新绑定。

#### void usb\_queue\_reset\_device(struct usb\_interface \*iface)

从原子上下文重置USB设备

**参数**

struct usb\_interface \*iface

属于要重置的设备的USB接口

**说明**

此函数可用于从原子上下文中重置USB设备，在该上下文中，usb\_reset\_device()不起作用（因为它会阻塞）。

通过此方法进行重置，在功能上等同于调用usb\_reset\_device()，但是它被延迟到工作队列。这意味着，任何绑定到其他接口的驱动程序以及来自用户空间的usbfs用户可能会被取消绑定。

边际案例：

从同一设备的两个不同接口附加的两个不同驱动程序同时安排两个重置是可能的；根据附加到每个接口的驱动程序如何处理->pre\_reset()，第二个重置可能会发生或不发生。

如果重置被延迟太久，以至于接口从其驱动程序解除绑定，则重置将被跳过。

此功能可以在.probe()期间调用。它也可以在.disconnect()期间调用，但这样做是毫无意义的，因为重置不会发生。如果你真的想在.disconnect()期间重置设备，直接调用usb\_reset\_device()，但要注意嵌套取消绑定问题！

#### struct usb\_device \*usb\_hub\_find\_child(struct usb\_device \*hdev, int port1)

获取指定端口连接的子设备的指针。

**参数**

struct usb\_device \*hdev

属于usb集线器的USB设备

int port1

端口号，用于指示子设备连接到哪个端口。

**说明**

USB驱动程序调用此功能以获取集线器的子设备指针。

**返回**

如果输入参数无效，则为NULL，如果非NULL，则为子设备的usb\_device指针。

### 主控制器API

这些API仅供主控制器驱动程序使用，其中大多数实现标准注册接口，如XHCI、EHCI、OHCI或UHCI。UHCI是最早的接口之一，由英特尔设计，也用于VIA；它在硬件方面没有做太多的工作。OHCI是后来设计的，为了让硬件做更多的工作（更大的传输，跟踪协议状态等等）。EHCI是为USB 2.0设计的；它的设计具有与OHCI相似的特点（硬件做了很多工作）以及UHCI的一些部分（ISO支持的一些部分，TD列表处理）。XHCI是为USB 3.0设计的。它继续将功能支持转移到硬件中。

除了“三巨头”之外，还有其他主控制器，虽然大多数基于PCI的控制器（以及一些非基于PCI的控制器）使用其中一个接口。并非所有主控制器都使用DMA；有些使用PIO，还有一个模拟器和一个虚拟主控制器来通过网络管道传输USB。

对于所有这些控制器，驱动程序都可以使用相同的基本API。出于历史原因，它们分为两层struct usb\_bus是一个相当薄的层，它在2.2内核中变得可用，而struct usb\_hcd是一个更具特色的层，它允许HCD共享公共代码以缩小驱动程序大小并显着减少hcd特定的行为。

#### long usb\_calc\_bus\_time(int speed, int is\_input, int isoc, int bytecount)

近似周期事务时间（以纳秒为单位）

**参数**

int speed

从dev->speed; USB\_SPEED\_{LOW,FULL,HIGH}

int is\_input

如果交易将数据发送到主机，则为true

int isoc

对于等时性事务为真，对于中断性事务为假

int bytecount

事务中的字节数。

**返回**

周期性事务的近似总线时间（以纳秒为单位）。

**注意**

请参阅USB 2.0规范第5.11.3节；只有周期性传输需要在软件中进行调度，此函数仅用于此类调度。

#### int usb\_hcd\_link\_urb\_to\_ep(struct usb\_hcd \*hcd, struct urb \*urb)

将URB添加到其端点队列

**参数**

struct usb\_hcd \*hcd

urb提交到的主控制器

struct urb \*urb

正在提交的URB

**说明**

主控制器驱动程序应在其enqueue()方法中调用此例程。必须保留HCD的私有自旋锁，并禁用中断。在这里进行的操作对于URB提交以及端点关闭和usb\_kill\_urb是必需的。

**返回**

0表示无错误，否则为负错误代码（在这种情况下，enqueue()方法必须失败）。如果没有错误发生，但是enqueue()仍然失败，则必须在释放私有自旋锁并返回之前调用usb\_hcd\_unlink\_urb\_from\_ep()。

#### int usb\_hcd\_check\_unlink\_urb(struct usb\_hcd \*hcd, struct urb \*urb, int status)

检查是否可以取消链接URB

**参数**

struct usb\_hcd \*hcd

urb提交到的主控制器

struct urb \*urb

正在检查可链接性的URB

int status

如果取消链接成功，则在urb中存储错误代码

**说明**

主控制器驱动程序应在其dequeue()方法中调用此例程。必须保留HCD的私有自旋锁，并禁用中断。在这里进行的操作需要确保取消链接有效。

**返回**

0表示无错误，否则为负错误代码（在这种情况下，dequeue()方法必须失败）。可能的错误代码为

-EIDRMurb未提交或已完成。

可能尚未调用完成函数。

-EBUSYurb已被取消链接。

#### void usb\_hcd\_unlink\_urb\_from\_ep（struct usb\_hcd \* hcd，struct urb \* urb）

从其端点队列中删除URB

**参数**

struct usb\_hcd \* hcd

urb提交的主机控制器

struct urb \* urb

正在取消链接的URB

**说明**

主机控制器驱动程序应在调用usb\_hcd\_giveback\_urb（）之前调用此例程。必须保持HCD的私有自旋锁和禁用中断。在这里执行的操作对于完成URB是必需的。

#### void usb\_hcd\_giveback\_urb（struct usb\_hcd \* hcd，struct urb \* urb，int status）

将URB从HCD返回到设备驱动程序

**参数**

struct usb\_hcd \* hcd

返回URB的主机控制器

struct urb \* urb

正在返回到USB设备驱动程序的urb。

int status

URB的完成状态代码。

**上下文**

原子。完成回调在调用者的上下文中调用。对于HCD\_BH标志设置的HCD，完成回调在任务let上下文中调用（除了提交到根集线器的URB始终在调用者的上下文中完成）。

**说明**

使用其完成函数将URB从HCD移交给其USB设备驱动程序。 HCD已释放了所有每个urb资源（并且正在使用urb->hcpriv）。它还释放了所有HCD锁定；如果设备驱动程序释放，修改或重新提交此URB，它不会引起问题。

如果未链接urb，则通过urb->unlinked覆盖状态的值。在HCD没有检查它们的情况下检测到错误的短传输。

#### int usb\_alloc\_streams（struct usb\_interface \* interface，struct usb\_host\_endpoint \*\* eps，unsigned int num\_eps，unsigned int num\_streams，gfp\_t mem\_flags）

分配批量端点流ID。

**参数**

struct usb\_interface \* interface

包含所有端点的备选设置。

struct usb\_host\_endpoint \*\* eps

需要流的端点数组。

unsigned int num\_eps

数组中的端点数。

unsigned int num\_streams

要分配的流数量。

gfp\_t mem\_flags

hcd应使用的标志来分配内存。

**说明**

设置一组批量端点，使num\_streams流ID可用。驱动程序可以将多个传输排队到不同的流ID，这些传输可能以不同于它们排队的顺序完成。

**返回**

成功时，分配的流数量。失败时，为负错误代码。

#### int usb\_free\_streams（struct usb\_interface \* interface，struct usb\_host\_endpoint \*\* eps，unsigned int num\_eps，gfp\_t mem\_flags）

释放批量端点流ID。

**参数**

struct usb\_interface \* interface

包含所有端点的备选设置。

struct usb\_host\_endpoint \*\* eps

要从中删除流的端点数组。

unsigned int num\_eps

数组中的端点数。

gfp\_t mem\_flags

hcd应使用的标志来分配内存。

**说明**

将一组批量端点还原为不使用流ID。如果我们提供错误的参数或HCD有问题，则可能失败。

**返回**

成功为0。失败时，为负错误代码。

#### void usb\_hcd\_resume\_root\_hub（struct usb\_hcd \* hcd）

由HCD调用以恢复其根集线器

**参数**

struct usb\_hcd \* hcd

此根集线器的主机控制器

**说明**

当USB主机控制器被挂起（启用了远程唤醒功能）并收到远程唤醒请求时，USB主机控制器调用此函数。该例程提交一个工作队列请求以恢复根集线器（即再次管理其下游端口）。

#### int usb\_bus\_start\_enum（struct usb\_bus \* bus，unsigned port\_num）

立即开始enum（用于OTG）

**参数**

struct usb\_bus \* bus

总线（必须使用hcd框架）

unsigned port\_num

端口的基于1的编号；通常为bus->otg\_port

**上下文**

原子

**说明**

启动enum，带有立即复位，随后由hub\_wq标识和可能配置设备。这对于OTG控制器驱动程序是必需的，因为它有助于满足启动端口复位的HNP协议定时要求。

**返回**

如果成功，则为0。

#### irqreturn\_t usb\_hcd\_irq（int irq，void \* \_\_ hcd）

将IRQ连接到HCD框架（总线胶水）

**参数**

int irq

正在提高的IRQ

void \* \_\_ hcd

指向信号化其IRQ的HCD的指针

**说明**

如果控制器没有被HALTed，则调用驱动程序的irq处理程序。检查控制器现在是否已死亡。

**返回**

如果已处理IRQ，则为IRQ\_HANDLED。否则为IRQ\_NONE。

#### void usb\_hc\_died（struct usb\_hcd \* hcd）

报告主机控制器的异常关闭（总线胶水）

**参数**

struct usb\_hcd \* hcd

指向表示控制器的HCD的指针

**说明**

总线胶水调用此函数以报告仍可能有未决操作的USB主机控制器已死亡。 PCI胶水自动调用它，因此仅非PCI总线的胶水需要调用它。

仅使用主HCD调用此函数。

#### struct usb\_hcd \* usb\_create\_shared\_hcd（const struct hc\_driver \* driver，struct device \* dev，const char \* bus\_name，struct usb\_hcd \* primary\_hcd）

创建和初始化HCD结构

**参数**

const struct hc\_driver \* driver

将使用此hcd的HC驱动程序

struct 设备\*dev

存储在hcd->self.controller中的此HC的设备

const char\*bus\_name

存储在hcd->self.bus\_name中的值

struct usb\_hcd\*primary\_hcd

指向共享PCI设备的usb\_hcd结构的指针。仅为主HCD分配某些资源

**上下文**

!:C:funcin\_interrupt（）

**说明**

分配一个struct usb\_hcd，其末尾有HC驱动程序的私有数据的额外空间。初始化hcd结构的通用成员。

**返回**

成功时，返回创建并初始化的HCD结构的指针。失败时（例如如果内存不可用），返回NULL。

#### struct usb\_hcd \*usb\_create\_hcd(const struct hc\_driver \*driver, struct device \*dev, const char \*bus\_name)

创建和初始化HCD结构

**参数**

const struct hc\_driver \*driver

将使用此HCD的HC驱动程序

struct device \*dev

存储在hcd->self.controller中的此HC的设备

const char \*bus\_name

存储在hcd->self.bus\_name中的值

**上下文**

!:C:funcin\_interrupt（）

**说明**

分配一个struct usb\_hcd，其末尾有HC驱动程序的私有数据的额外空间。初始化hcd结构的通用成员。

**返回**

成功时，返回创建并初始化的HCD结构的指针。失败时（例如如果内存不可用），返回NULL。

#### int usb\_add\_hcd(struct usb\_hcd \*hcd, unsigned int irqnum, unsigned long irqflags)

完成通用HCD结构初始化并注册

**参数**

struct usb\_hcd \*hcd

要初始化的usb\_hcd结构

unsigned int irqnum

要分配的中断线

unsigned long irqflags

中断类型标志

**说明**

完成通用HCD初始化的剩余部分分配一致内存的缓冲区，注册总线，请求IRQ线路，并调用驱动程序的reset（）和start（）例程。

#### void usb\_remove\_hcd(struct usb\_hcd \*hcd)

用于通用HCD的关闭处理

**参数**

struct usb\_hcd \*hcd

要删除的usb\_hcd结构

**上下文**

!:C:funcin\_interrupt（）

**说明**

断开根集线器，然后撤消usb\_add\_hcd（）的影响，调用HCD的stop（）方法。

#### int usb\_hcd\_pci\_probe(struct pci\_dev \*dev, const struct hc\_driver \*driver)

初始化基于PCI的HCD

**参数**

struct pci\_dev \*dev

正在探测的USB主机控制器

const struct hc\_driver \*driver

USB HC驱动程序句柄

**上下文**

!:C:funcin\_interrupt（）

**说明**

为此USB主机控制器分配基本PCI资源，然后通过热插拔入口的driver\_data调用与其关联的HCD的start（）方法。

将此函数存储在HCD的struct pci\_driver中作为probe（）。

**返回**

成功时返回0。

#### void usb\_hcd\_pci\_remove(struct pci\_dev \*dev)

用于基于PCI的HCD的关闭处理

**参数**

struct pci\_dev \*dev

正在删除的USB主机控制器

**上下文**

任务上下文，可能休眠

**说明**

撤消usb\_hcd\_pci\_probe（）的影响，首先调用HCD的stop（）方法。它始终从线程上下文调用，通常为“rmmod”、“apmd”或类似的内容。

将此函数存储在HCD的struct pci\_driver中作为remove（）。

#### void usb\_hcd\_pci\_shutdown(struct pci\_dev \*dev)

关机主机控制器

**参数**

struct pci\_dev \*dev

正在关闭的USB主机控制器

#### int hcd\_buffer\_create(struct usb\_hcd \*hcd)

初始化缓冲池

**参数**

struct usb\_hcd \*hcd

要初始化其缓冲区池的总线

**上下文**

任务上下文，可能休眠

**说明**

作为使用dma内存分配器的主机控制器初始化的一部分调用此函数。它初始化一些dma一致性内存池，这些池将由使用该控制器的所有驱动程序共享。

使用完这些池后调用hcd\_buffer\_destroy（）以进行清理。

**返回**

成功时返回0。否则为负的errno值。

#### void hcd\_buffer\_destroy(struct usb\_hcd \*hcd)

释放缓冲池

**参数**

struct usb\_hcd \*hcd

要销毁其缓冲区池的总线

**上下文**

任务上下文，可能休眠

**说明**

此函数释放hcd\_buffer\_create（）创建的缓冲池。

### USB字符设备节点

本章介绍Linux字符设备节点。您可能希望避免为您的USB驱动程序编写新内核代码。用户模式设备驱动程序通常被打包为应用程序或库，并且可以通过某些编程库使用字符设备。这些库包括

1）用于C / C ++的libusb和

2）Java的jUSB。

可以在USB指南的“USB设备文件系统”部分看到有关其的一些旧信息。可以在http://www.linux-usb.org/找到USB指南的最新副本。

**注意**

它们曾经是通过usbfs实现的，但这不是sysfs调试接口的一部分。

此文档尤其是关于异步模式的部分尚不完整。从内核2.5.66开始，需要对代码和这个（新的）文档进行交叉审查。

#### “devtmpfs”中有哪些文件？

传统上，usbfs被安装在/dev/bus/usb/，其功能包括

/dev/bus/usb/BBB/DDD … 魔术文件显示每个设备的配置说明符，并支持一系列ioctl用于发起设备请求，包括向设备进行I/O。（纯粹供程序访问）

每个总线都有一个基于enum顺序的编号（BBB）。在每个总线中，每个设备都有一个类似的编号（DDD）。这些BBB/DDD路径不是“稳定”标识符；即使你总是将设备插入到相同的集线器端口，它们也会发生变化。甚至不要考虑将它们保存在应用程序配置文件中。对于想要使用它们的用户模式应用程序，可用的是稳定的标识符。HID和网络设备公开这些稳定的ID，例如您可以确保将正确的UPS关机以关闭其第二个服务器。请注意它目前并不公开这些ID。

#### /dev/bus/usb/BBB/DDD

使用这些文件有以下基本方法

可以读取它们，首先得到设备说明符（18字节），然后是当前配置的说明符。有关这些二进制数据格式的详细信息，请参见USB 2.0规范。您需要将大多数多字节数值从小端格式转换为本地主机字节顺序，尽管设备说明符中的一些字段（BCD编码字段和供应商和产品ID）将被自动交换字节顺序。请注意，配置说明符包括接口、altsettings、端点和可能额外的类说明符的说明符。

使用ioctl()请求执行USB操作，以发起端点I/O请求（同步或异步）或管理设备。这些请求需要CAP\_SYS\_RAWIO能力，以及文件系统访问权限。在这些设备文件中，一次只能进行一个ioctl请求。这意味着，如果您正在从一个线程同步读取一个端点，那么在读取完成之前将无法从另一个线程写入到不同的端点。这适用于半双工协议，但在其他情况下，您需要使用异步I/O请求。

每个连接的USB设备都有一个文件。BBB表示总线编号。DDD表示该总线上的设备地址。这些数字都是按顺序分配的，可以被重用，因此您不能依赖它们来稳定地访问设备。例如，当设备仍然连接时（可能是因为有人挤压了它们的电源供应、中心或USB电缆），设备可能在您第一次连接它时是002/027，而稍后可能是002/048。

可以将这些文件作为二进制数据读取。二进制数据由设备说明符和每个设备配置的说明符组成。内核将设备说明符中的多字节字段转换为主机字节顺序。配置说明符以总线字节序格式表示！配置说明符相距wTotalLength字节。如果设备返回的配置说明符数据少于wTotalLength所指示的，则缺少字节的文件中将有一个洞。这些信息也可以通过稍后说明的/sys/kernel/debug/usb/devices文件以文本形式显示。

这些文件也可以用于编写USB设备的用户级驱动程序。您可以以读/写方式打开/dev/bus/usb/BBB/DDD文件，并从中读取说明符以确保它是您所期望的设备，然后使用ioctl调用绑定到一个或多个接口。您将向该设备发出更多的ioctls，以使用控制、批量或其他类型的USB传输与其通信。IOCTL在<linux/usbdevice\_fs.h>文件中列出，本文撰写时源代码(linux/drivers/usb/core/devio.c)是访问这些文件的主要参考。

请注意，默认情况下，这些BBB/DDD文件仅可由root编写，因此只有root可以编写这样的用户模式驱动程序。您可以使用chmod有选择地授予其他用户读写权限。此外，像devmode=0666这样的usbfs挂载选项可能会有所帮助。

#### 用户模式驱动程序的生命周期

这样的驱动程序首先需要为它知道如何处理的设备找到一个设备文件。也许是因为/sbin/hotplug事件处理代理选择该驱动程序来处理新设备，或者可能是应用程序扫描了所有的/dev/bus/usb设备文件，并忽略了大多数设备。在任一情况下，它都应该从设备文件中读取所有说明符，并将其与它知道如何处理的说明符进行检查。它可能只拒绝除特定厂商和产品ID之外的所有内容，或需要更复杂的策略。

永远不要假设在系统上只有一个这样的设备！如果您的代码不能同时处理多个设备，请至少检测到是否有多个设备，并让您的用户选择要使用哪个设备。

一旦您的用户模式驱动程序知道要使用什么设备，它会以两种方式之一与其进行交互。简单的方式是仅进行控件请求；一些设备不需要比这更复杂的交互。（例如，使用供应商特定的控制请求进行某些初始化或配置任务，并使用内核驱动程序处理其余部分可能是一个示例）。

更可能的是，您需要一个更复杂的样式驱动程序使用非控制端点，读取或编写数据，并声称独占界面。批量传输最容易使用，但仅当其兄弟间断性传输与低速设备一起使用时才起作用。中断和等时传输都提供服务保证，因为它们的带宽是预留的。这些“周期性”传输在usbfs中使用起来很麻烦，除非您使用异步调用。但是，中断传输也可以以同步的“单次”样式使用。

您的用户模式驱动程序不应该担心在设备断开连接时清除请求状态，尽管一旦开始看到ENODEV错误，它应该立即关闭其打开的文件说明符。

#### ioctl（）请求

#include <linux/usb.h>

#include <linux/usbdevice\_fs.h>

#include <asm/byteorder.h>

USB 2.0 规范“第 9 章”中的标准 USB 设备模型请求自动包含在标<linux/usb/ch9.h> 头中。

除非另有说明，此处描述的 ioctl 请求将更新应用它们的 usbfs 文件的修改时间（除非它们失败）。返回零表示成功；否则，将返回一个标准的 USB 错误代码（这些代码记录在 [USB 错误代码](https://www.kernel.org/doc/html/v4.19/driver-api/usb/error-codes.html" \l "usb-error-codes)中）。

这些文件中的每一个都复用对多个 I/O 流的访问，每个端点一个。每个设备都有一个控制端点（端点零），它支持有限的 RPC 样式 RPC 访问。设备由 hub\_wq（在内核中）配置，设置设备范围的配置，影响功耗和基本功能等。端点是 USB接口的一部分，它可能有altsettings 影响诸如哪些端点可用之类的事情。许多设备只有一个配置和接口，因此它们的驱动程序将忽略配置和 altsettings。

#### 管理/状态请求

许多 usbfs 请求并不直接处理设备 I/O。它们主要与设备管理和状态有关。这些都是同步请求。

1）USBDEVFS\_CLAIMINTERFACE

这用于强制 usbfs 声明一个特定的接口，该接口以前没有被 usbfs 或任何其他内核驱动程序声明过。ioctl 参数是一个包含接口编号的整数（来自描述符的 bInterfaceNumber）。

请注意，如果您的驱动程序在尝试使用其端点之一之前未声明接口，并且没有其他驱动程序绑定到它，则 usbfs 会自动声明该接口。

此声明将通过 RELEASEINTERFACE ioctl 或通过关闭文件描述符来释放。此请求未更新文件修改时间。

2）USBDEVFS\_CONNECTINFO

表示设备是否为低速。ioctl 参数指向这样的结构：

struct usbdevfs\_connectinfo {

unsigned int devnum;

unsigned char slow;

};

此请求未更新文件修改时间。

您无法判断“不慢”的设备是高速连接（480 MBit/秒）还是全速连接（12 MBit/秒）。你应该已经知道 devnum 值，它是设备文件名的 DDD 值。

3）USBDEVFS\_GETDRIVER

返回绑定到给定接口（字符串）的内核驱动程序的名称。参数是指向这个结构体的指针，修改为：

struct usbdevfs\_getdriver {

unsigned int interface;

char driver[USBDEVFS\_MAXDRIVERNAME + 1];

};

此请求未更新文件修改时间。

4）USBDEVFS\_IOCTL

将来自用户空间的请求传递给内核驱动程序，该驱动程序在其注册的结构 usb\_driver中具有 ioctl 条目：

struct usbdevfs\_ioctl {

int ifno;

int ioctl\_code;

void \*data;

};

/\* user mode call looks like this.

\* 'request' becomes the driver->ioctl() 'code' parameter.

\* the size of 'param' is encoded in 'request', and that data

\* is copied to or from the driver->ioctl() 'buf' parameter.

\*/

static int

usbdev\_ioctl (int fd, int ifno, unsigned request, void \*param)

{

struct usbdevfs\_ioctl wrapper;

wrapper.ifno = ifno;

wrapper.ioctl\_code = request;

wrapper.data = param;

return ioctl (fd, USBDEVFS\_IOCTL, &wrapper);

}

此请求未更新文件修改时间。

此请求允许内核驱动程序通过文件系统操作与用户模式代码通信，即使它们不创建字符或块特殊设备。它也被用来做一些事情，比如询问设备应该使用什么设备特殊文件。两个预定义的 ioctl 用于断开和重新连接内核驱动程序，以便用户模式代码可以完全管理设备的绑定和配置。

5）USBDEVFS\_RELEASEINTERFACE

这用于释放声明 usbfs 在文件描述符关闭之前隐式地或由于 USBDEVFS\_CLAIMINTERFACE 调用而对接口进行的声明。ioctl 参数是一个包含接口编号的整数（来自描述符的 bInterfaceNumber）；此请求未更新文件修改时间。

**警告**

没有进行安全检查来确保提出申请的任务是释放它的任务。这意味着用户模式驱动程序可能会干扰其他用户模式驱动程序。

6）USBDEVFS\_RESETEP

将端点（块或中断）的数据切换值重置为 DATA0。IOCTL 参数是一个整数端点号（1 到 15，如在端点说明符中标识的那样），如果设备的端点向主机发送数据，则添加 USB\_DIR\_IN。

**警告**

应避免使用此请求。它可能会被移除。使用它通常意味着设备和驱动程序将失去切换同步。如果您真的失去了同步，您可能需要完全与设备握手，使用诸如 CLEAR\_HALT 或 SET\_INTERFACE 的请求。

7）USBDEVFS\_DROP\_PRIVILEGES

这是用于放弃在 USBFS 文件说明符上进行 certain 操作的能力，这些操作被认为是有特权的。这包括声明任意接口，重置设备上的当前被其他用户声明的接口，并发出 USBDEVFS\_IOCTL 调用。IOCTL 参数是 32 位接口掩码，用于指示用户允许在此文件说明符上声明哪个接口。您可以多次发出此 IOCTL，以缩小所述掩码。

#### 同步 I/O 支持

同步请求涉及内核阻塞，直到用户模式请求完成，或者通过报告错误来完成。在大多数情况下，这是使用 USBFS 的最简单方法，尽管如上所述，它会阻止对多个端点进行 I/O。

1）USBDEVFS\_BULK

向设备发出块读取或写入请求。IOCTL 参数是指向此结构的指针:

struct usbdevfs\_bulktransfer {

unsigned int ep;

unsigned int len;

unsigned int timeout; /\* in milliseconds \*/

void \*data;

};

ep 值标识一个块端点号（1 到 15，如在端点说明符中标识的那样），当引用从设备发送数据到主机的端点时，使用 USB\_DIR\_IN 进行屏蔽。数据缓冲区的长度由 len 标识；最近的内核支持请求高达大约 128KB 的数据。 FIXME 说明如何返回读取长度，以及如何处理短读取……

2）USBDEVFS\_CLEAR\_HALT

清除端点停止（stall）并重置端点切换。这仅对块或中断端点有意义。IOCTL 参数是一个整数端点号（1 到 15，如在端点说明符中标识的那样），当引用从设备发送数据到主机的端点时，使用 USB\_DIR\_IN 进行屏蔽。

在块或中断端点上使用它们已经停滞，返回 -EPIPE 状态给数据传输请求。不要直接发出控制请求，因为这可能会使主机对数据切换的记录无效。

3）USBDEVFS\_CONTROL

向设备发出控制请求。IOCTL 参数指向类似于以下结构的结构:

struct usbdevfs\_ctrltransfer {

\_\_u8 bRequestType;

\_\_u8 bRequest;

\_\_u16 wValue;

\_\_u16 wIndex;

\_\_u16 wLength;

\_\_u32 timeout; /\* in milliseconds \*/

void \*data;

};

此结构的前八个字节是要发送到设备的 SET UP 数据包的内容；有关详细信息，请参见 USB 2.0 规范。 bRequestType 值是通过组合 USB\_TYPE\_\* 值、USB\_DIR\_\* 值和 USB\_RECIP\_\* 值（来自 linux/usb.h）组成的。如果 wLength 不为零，则它说明了数据缓冲区的长度，该长度将被写入到设备（USB\_DIR\_OUT）或从设备读取（USB\_DIR\_IN）。

在撰写本文时，您不能将超过 4 KB 的数据传输到设备或从设备传输；usbfs 有一定的限制，并且一些主机控制器驱动程序也有限制。（这通常不是问题。）此外，没有办法说不能从设备获得短读取是不可以的。

4）USBDEVFS\_RESET

执行 USB 级设备复位。IOCTL 参数被忽略。复位后，这将重新绑定所有设备接口。此请求不会影响文件的修改时间。

**警告**

在一些 USB 核心的错误得到修复之前，应避免使用此调用，因为它不完全同步设备、接口和驱动程序（不仅仅是 usbfs）状态。

5）USBDEVFS\_SETINTERFACE

设置界面的备用设置。IOCTL 参数是指向此结构的指针:

struct usbdevfs\_setinterface {

unsigned int interface;

unsigned int altsetting;

};

此结构成员来自适用于当前配置的某些界面说明符。界面编号是 bInterfaceNumber 值，备用设置编号是 bAlternateSetting 值。（这将重置界面中的每个端点。）此请求不会影响文件的修改时间。

6）USBDEVFS\_SETCONFIGURATION

发出用于设备的usb\_set\_configuration()调用。参数是一个整数，保存着配置号码（说明符中的bConfigurationValue）。这个请求不会更新文件的修改时间。

警告

在某些usbcore bug得到修复之前避免使用此调用，因为它没有完全同步设备、界面和驱动程序（不仅仅是usbfs）的状态。

#### 异步I/O支持

如上所述，有些情况下，从用户模式代码发起并行操作可能是非常重要的。这对于定期传输（中断和等时传输）尤其重要，但也可用于其他类型的USB请求。在这种情况下，异步请求是至关重要的。与提交一个请求并使内核阻塞直到它完成不同的是，阻塞是单独进行的。

这些请求被封装为类似于内核设备驱动程序使用的URB结构板。它标识出端点类型（USBDEVFS\_URB\_TYPE\_\*）、端点（数字，根据情况与USB\_DIR\_IN屏蔽），缓冲区和长度以及一个用户“上下文”值，用于唯一标识每个请求（通常是指向每个请求数据的指针）。标志可以修改请求（不像内核驱动程序支持的那么多）。

每个请求都可以指定实时信号号码（在SIGRTMIN和SIGRTMAX之间，包括两者）来请求在请求完成时发送信号。

当usbfs返回这些URB时，状态值会更新，缓冲区可能已被修改。除了等时传输，实际长度都更新为传输了多少字节；如果设置了USBDEVFS\_URB\_DISABLE\_SPD标志（“短数据包不行”），如果读取的字节数小于请求的字节数，则会获得一份错误报告:

struct usbdevfs\_iso\_packet\_desc {

unsigned int length;

unsigned int actual\_length;

unsigned int status;

};

struct usbdevfs\_urb {

unsigned char type;

unsigned char endpoint;

int status;

unsigned int flags;

void \*buffer;

int buffer\_length;

int actual\_length;

int start\_frame;

int number\_of\_packets;

int error\_count;

unsigned int signr;

void \*usercontext;

struct usbdevfs\_iso\_packet\_desc iso\_frame\_desc[];

};

对于这些异步请求，文件修改时间反映出请求的初始化时间。这与它们与同步请求的使用形成对比，后者反映出请求完成的时间。

USBDEVFS\_DISCARDURB

待定 文件修改时间不会因此请求而更新。

USBDEVFS\_DISCSIGNAL

待定 文件修改时间不会因此请求而更新。

USBDEVFS\_REAPURB

待定 文件修改时间不会因此请求而更新。

USBDEVFS\_REAPURBNDELAY

待定 文件修改时间不会因此请求而更新。

USBDEVFS\_SUBMITURB

待定

### USB设备

现在可以通过debugfs导出USB设备

/sys/kernel/debug/usb/devices … 显示内核中已知的每个USB设备及其配置说明符的文本文件。您还可以对它进行poll()操作以了解新设备。

#### /sys/kernel/debug/usb/devices

这个文件对于用户模式下的状态查看工具非常方便，它可以扫描文本格式并忽略大部分内容。有关设备特定文件的更详细的设备状态（包括类和厂商状态）可用。有关此文件的当前格式信息，请参见下文。

这个文件与poll()系统调用结合使用，还可用于检测设备何时添加或删除:

int fd;

struct pollfd pfd;

fd = open("/sys/kernel/debug/usb/devices", O\_RDONLY);

pfd = { fd, POLLIN, 0 };

for (;;) {

/\* The first time through, this call will return immediately. \*/

poll(&pfd, 1, -1);

/\* To see what's changed, compare the file's previous and current

contents or scan the filesystem. (Scanning is more precise.) \*/

}

请注意，这种行为旨在用于信息和调试目的。初始化设备或启动用户模式帮助程序更适合使用诸如udev或HAL等程序。

在此文件中，每个设备的输出有多行ASCII输出。

我故意把它设为ASCII而不是二进制，以便某人可以无需辅助程序获取有用的数据。但是，使用辅助程序，每个T:行的前4列中的数字（拓扑信息Lev、Prnt、Port、Cnt）可用于构建USB拓扑图。

每行都有一个字母标记表示该行的ID:

T = Topology (etc.)

B = Bandwidth (applies only to USB host controllers, which are

virtualized as root hubs)

D = Device descriptor info.

P = Product ID info. (from Device descriptor, but they won't fit

together on one line)

S = String descriptors.

C = Configuration descriptor info. (\* = active configuration)

I = Interface descriptor info.

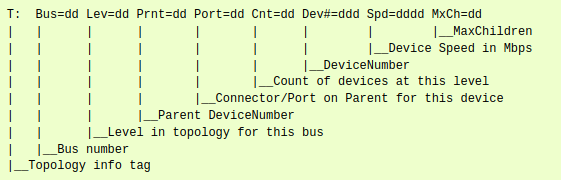
E = Endpoint descriptor info.

**/sys/kernel/debug/usb/devices输出格式**

解释

d = 十进制数字（可能有前导空格或0）x = 十六进制数字（可能有前导空格或0）s = 字符串

1）拓扑信息

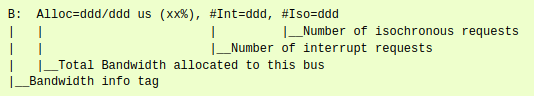


速度可能是:

|  |  |
| --- | --- |
| 1.5 | Mbit/s for low speed USB |
| 12 | Mbit/s for full speed USB |
| 480 | Mbit/s for high speed USB (added for USB 2.0); also used for Wireless USB, which has no fixed speed |
| 5000 | Mbit/s for SuperSpeed USB (added for USB 3.0) |

由于过去的原因已不可考，端口号始终比实际少1。例如，插入第4个端口的设备将显示为Port=03。

2）带宽信息



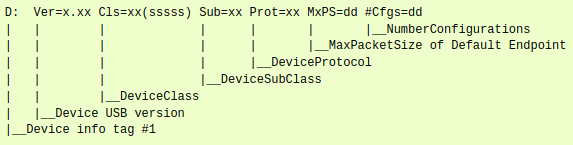
带宽分配是一个框架（毫秒）使用量的近似值。它仅反映了周期性传输，这是保留带宽的唯一传输。控制和块传输使用所有其他带宽，包括未用于传输的保留带宽（如短数据包）。

百分比表示这些传输所安排的“保留”带宽的比例。对于低速或全速总线（宽松地说，“USB 1.1”），总线带宽的90%被保留。对于高速总线（宽松地说，“USB 2.0”）80%被保留。

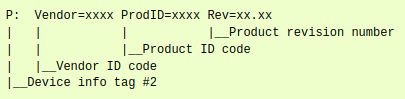
3）设备说明符信息和产品ID信息



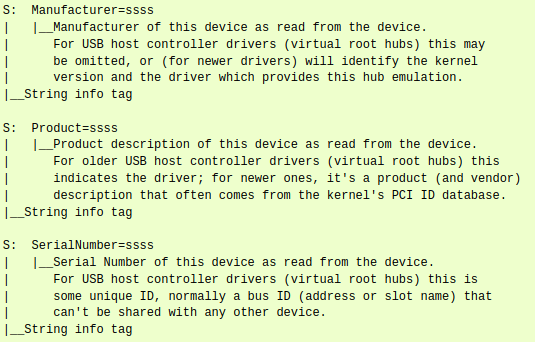
其中:



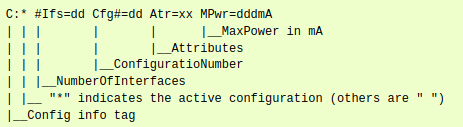
其中:



4）字符串描述符信息



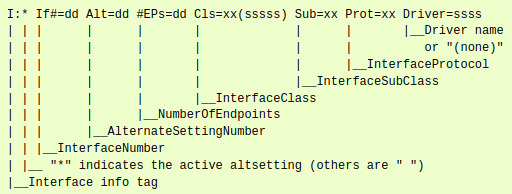
5）配置描述符信息



USB设备可能有多个配置，每个配置的行为都有所不同。例如，总线供电的配置可能比自供电的配置功能要少得多。一次只能激活一个设备配置；大多数设备仅有一个配置。

每个配置由一个或多个接口组成。每个接口提供独特的“功能”，通常与不同的USB设备驱动程序绑定。一个常见的例子是带有音频接口用于回放和一个HID接口用于与软件音量控制使用的USB扬声器。

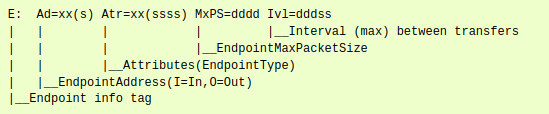
6）接口说明符信息（每个配置可以有多个）



给定接口可能有一个或多个“备用”设置。例如，默认设置可能不使用超过少量的周期带宽。要使用总线带宽的大部分，驱动程序必须选择非默认的altsetting。

在同一时间只能激活接口的一个设置，也只能绑定一个驱动程序到接口。大多数设备每个接口仅有一个备用设置。

7）端点说明符信息（每个接口可以有多个）



对于所有周期性（中断或等时）端点，间隔值为非零值。对于高速端点，传输间隔可以用微秒而不是毫秒来测量。

对于高速周期性端点，EndpointMaxPacketSize反映每个微帧数据传输的大小。对于“高带宽”端点，那可以反映两个或三个数据包（每个端点每125μs最多3K字节）。

在Linux-USB栈中，周期性带宽预留使用URB提供的传输间隔和大小，它们可以小于在端点说明符中找到的那些。

**使用示例：**

例如，如果用户或脚本只对拓扑信息感兴趣，则可以使用类似grep ^T/sys/kernel/debug/usb/devices的命令，只用于顶点线。可以使用类似grep -i ^[tdp]: /sys/kernel/debug/usb/devices的命令来列出仅以方括号中的字符开头的行，其中有效字符是TDPCIE。使用稍微更能的脚本，它可以显示任何选定的行（例如，仅T，D和P行）并更改其输出格式。（procusb Perl脚本是这个想法的开始。它将仅列出来自/sys/kernel/debug/usb/devices的所选行[从TBDPSCIE中选择]或“所有”行。）

拓扑线可以用于生成系统根集线器USB设备的图形/图示。 （有关如何执行此操作的更多信息，请参见下文。）

接口线可用于确定每个设备使用的驱动程序以及哪个altsetting激活。

配置行可用于列出系统USB设备使用的最大功率（以毫安为单位）。例如，grep /sys/kernel/debug/usb/devices。

以下是一个示例，来自具有UHCI根集线器、连接到根集线器的外部集线器以及连接到外部集线器的鼠标和串行转换器的系统。

T: Bus=00 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#= 1 Spd=12 MxCh= 2

B: Alloc= 28/900 us ( 3%), #Int= 2, #Iso= 0

D: Ver= 1.00 Cls=09(hub ) Sub=00 Prot=00 MxPS= 8 #Cfgs= 1

P: Vendor=0000 ProdID=0000 Rev= 0.00

S: Product=USB UHCI Root Hub

S: SerialNumber=dce0

C:\* #Ifs= 1 Cfg#= 1 Atr=40 MxPwr= 0mA

I: If#= 0 Alt= 0 #EPs= 1 Cls=09(hub ) Sub=00 Prot=00 Driver=hub

E: Ad=81(I) Atr=03(Int.) MxPS= 8 Ivl=255ms

T: Bus=00 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#= 2 Spd=12 MxCh= 4

D: Ver= 1.00 Cls=09(hub ) Sub=00 Prot=00 MxPS= 8 #Cfgs= 1

P: Vendor=0451 ProdID=1446 Rev= 1.00

C:\* #Ifs= 1 Cfg#= 1 Atr=e0 MxPwr=100mA

I: If#= 0 Alt= 0 #EPs= 1 Cls=09(hub ) Sub=00 Prot=00 Driver=hub

E: Ad=81(I) Atr=03(Int.) MxPS= 1 Ivl=255ms

T: Bus=00 Lev=02 Prnt=02 Port=00 Cnt=01 Dev#= 3 Spd=1.5 MxCh= 0

D: Ver= 1.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS= 8 #Cfgs= 1

P: Vendor=04b4 ProdID=0001 Rev= 0.00

C:\* #Ifs= 1 Cfg#= 1 Atr=80 MxPwr=100mA

I: If#= 0 Alt= 0 #EPs= 1 Cls=03(HID ) Sub=01 Prot=02 Driver=mouse

E: Ad=81(I) Atr=03(Int.) MxPS= 3 Ivl= 10ms

T: Bus=00 Lev=02 Prnt=02 Port=02 Cnt=02 Dev#= 4 Spd=12 MxCh= 0

D: Ver= 1.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS= 8 #Cfgs= 1

P: Vendor=0565 ProdID=0001 Rev= 1.08

S: Manufacturer=Peracom Networks, Inc.

S: Product=Peracom USB to Serial Converter

C:\* #Ifs= 1 Cfg#= 1 Atr=a0 MxPwr=100mA

I: If#= 0 Alt= 0 #EPs= 3 Cls=00(>ifc ) Sub=00 Prot=00 Driver=serial

E: Ad=81(I) Atr=02(Bulk) MxPS= 64 Ivl= 16ms

E: Ad=01(O) Atr=02(Bulk) MxPS= 16 Ivl= 16ms

E: Ad=82(I) Atr=03(Int.) MxPS= 8 Ivl= 8ms

从中筛选出 Ti 行和 I 行，例如使用 procusb ti，然后我们有

T: Bus=00 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#= 1 Spd=12 MxCh= 2

T: Bus=00 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#= 2 Spd=12 MxCh= 4

I: If#= 0 Alt= 0 #EPs= 1 Cls=09(hub ) Sub=00 Prot=00 Driver=hub

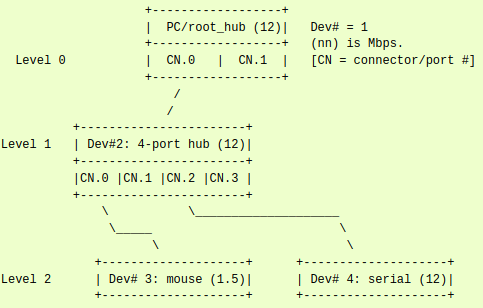
T: Bus=00 Lev=02 Prnt=02 Port=00 Cnt=01 Dev#= 3 Spd=1.5 MxCh= 0

I: If#= 0 Alt= 0 #EPs= 1 Cls=03(HID ) Sub=01 Prot=02 Driver=mouse

T: Bus=00 Lev=02 Prnt=02 Port=02 Cnt=02 Dev#= 4 Spd=12 MxCh= 0

I: If#= 0 Alt= 0 #EPs= 3 Cls=00(>ifc ) Sub=00 Prot=00 Driver=serial

物理上看起来像这样（或可以转换为这样）:



或者，以更树状的结构显示（没有连接的端口 [连接器] 可以省略）:



## 适用于Linux的USB Gadget API

### 介绍

本文介绍了一种 Linux-USB“Gadget”内核模式 API，用于嵌入 Linux 的外设和其他 USB 设备中。它提供了 API 结构的概述，并展示了如何将其适用于系统开发项目。这是 Linux 上首个采用这种 API 来解决多个重要问题的例子，包括：

1）支持 USB 2.0，适用于可以同时传输多达几十兆字节数据的高速设备。

2）与仅具有两个固定功能的设备一样，处理具有数十个端点的设备。Gadget 驱动程序可以编写得易于移植到新的硬件上。

3）足够灵活，以公开更复杂的 USB 设备功能，例如多个配置、多个接口、复合设备和备用接口设置。

4）与 Linux-USB host 端的更新一起提供 USB “On-The-Go” (OTG) 支持。

5）共享数据结构和 API 模型与 Linux-USB host 端 API。这有助于 OTG 支持，并且考虑到更对称的框架（其中两侧驱动程序使用相同的 I/O 模型）。

6）极简主义，因此更容易支持新的设备控制器硬件。I/O 处理不意味着对内存或 CPU 资源的大量需求。

大多数 Linux 开发人员将无法使用此 API，因为他们在个人计算机、工作站或服务器中拥有 USB host 硬件。使用嵌入式系统的 Linux 用户更有可能拥有 USB 外围硬件。为了区分在这种硬件中运行的驱动程序和更熟悉的 Linux “USB 设备驱动程序”（它们是真实 USB 设备的主机端代理），“USB gadget 驱动程序”使用不同的术语。在 USB 协议交互中，设备驱动程序是主控方（或“客户端驱动程序”），Gadget 驱动程序是从属方（或“功能驱动程序”）。

Gadget API 类似于主机端 Linux-USB API，它们都使用请求对象队列来打包 I/O 缓冲区，这些请求对象可以被提交或取消。它们共享标准 USB 第 9 章消息、结构和常量的公共定义。此外，两个 API 都将驱动程序绑定和解绑设备。这两个 API 在细节上有所不同，因为主机端的当前 URB 框架暴露了一些不适用于 gadget API 的实现细节和假设。虽然控制传输和配置管理的模型必然不同（一侧是硬件中立的主控方，另一侧是硬件感知的从属方），但在这里使用的端点 I/O API 也应该可用于开销减少的 host 端 API。

### Gadget驱动的结构

在一个USB外围设备内部运行的系统通常至少有三个内核层来处理USB协议处理，还可能在用户空间代码中有附加的层。中间层使用Gadget API与最低层交互（直接处理硬件的层）。

在Linux中，从下到上，这些层为

#### USB控制器驱动程序

这是最低的软件层，它是唯一与硬件通讯的层，通过寄存器，FIFO，DMA，IRQ等方式。 <linux/usb/gadget.h> API抽象外围控制器端点硬件。该硬件通过端点对象公开，其接收IN/OUT缓冲区流，并通过与Gadget驱动程序交互的回调公开。由于普通的USB设备只有一个上游端口，它们只有一个这样的驱动程序。控制器驱动程序可以支持任意数量的不同Gadget驱动程序，但一次只能使用一个。

此类控制器硬件的示例包括基于PCI的NetChip 2280 USB 2.0高速控制器，SA-11x0或PXA-25x UDC（在许多PDAs中找到）以及各种其他产品。

#### Gadget驱动程序

该驱动程序的下边界实现了与控制器驱动程序相互作用的硬件中立的USB功能。由于这种硬件的能力和限制各不相同，并且用于空间有限的嵌入式环境中，因此Gadget驱动程序通常在编译时配置为与一个特定控制器支持的端点一起工作。使用条件编译，这种驱动程序可以移植到多个不同的控制器上。（最近的内核极大地简化了自动为许多面向大量的驱动程序自动配置端点的工作。） Gadget驱动程序的职责包括：

1）处理设置请求（ep0协议响应），可能包括类特定功能

2）返回配置和字符串说明符

3）（重新）设置配置和界面备用设置，包括启用和配置端点

4）处理生命周期事件，例如管理与硬件的绑定，USB挂起/恢复，远程唤醒以及从USB主机断开连接。

5）管理所有当前启用端点上的IN和OUT传输

这些驱动程序可以是专有代码的模块，虽然这种方法在Linux社区中受到反对。

#### 上层

大多数Gadget驱动程序都有上边界，链接到Linux中的某些驱动程序或框架。通过该边界流动由Gadget驱动程序通过USB协议传输产生和/或消耗的数据。例如：

用户模式代码，使用通用（gadgetfs）或/ dev中的应用程序特定文件

网络子系统（用于网络设备，如CDC以太网模型Gadget驱动程序）

数据捕获驱动程序，例如video4Linux或扫描仪驱动程序；或测试和测量硬件。

输入子系统（用于HID设备）

声音子系统（用于音频设备）

文件系统（用于PTP设备）

块IO子系统（用于USB存储设备）

…等等

#### 其他层

可能存在其他层。这些可能包括内核层，例如网络协议堆栈，以及构建在标准POSIX系统调用API（例如open（），close（），read（）和write（））上的用户模式应用程序。在较新的系统上，POSIX异步I / O调用可能是一种选择。这样的用户模式代码不一定受GNU通用公共许可证（GPL）的约束。

OTG能力系统还需要包括标准的Linux-USB主机端堆栈，其中包括usbcore，一个或多个主机控制器驱动程序（HCDs），USB设备驱动程序以支持OTG“有针对性的外围设备列表”等。还将有OTG控制器驱动程序，后只能间接地对Gadget和设备驱动程序开发人员可见。这有助于主机和设备端USB控制器实现两个新的OTG协议（HNP和SRP）。在USB挂起处理期间，角色切换（从主机到外围设备，反之亦然）使用HNP，而SRP可以看作是一种更省电的设备唤醒协议。

随着时间的推移，可重用的实用程序正在发展，以帮助使一些Gadget驱动程序任务更简单。例如，从说明符向量构建配置说明符现在已经自动化，并且许多驱动程序现在使用自动配置来选择硬件端点并初始化其说明符。特别感兴趣的潜在示例是实现由HID，网络，存储或音频类提出的标准USB-IF协议的代码。一些开发人员对KDB或KGDB钩子感兴趣，以允许远程调试目标硬件。大多数此类USB协议代码不需要是特定于硬件的，就像X11，HTTP或NFS等网络协议一样。这样的Gadget端接口驱动程序最终应该合并，以实现复合设备。

### 内核模式Gadget API

通过struct usb\_gadget\_driver申明Gadget驱动程序，该申明负责struct usb\_gadget的大多数enum部分。对于struct usb\_gadget的一个set\_configuration的响应，通常涉及启用由该Gadget暴露的一个或多个struct usb\_ep对象，并提交一个或多个struct usb\_request缓冲区以传输数据。了解这四种数据类型及其操作，您就会理解这个API的工作方式。

**注意**

除了“第9章”数据类型外，大部分重要的数据类型和功能在此处被说明。

然而，您所阅读的内容可能省略了一些相关信息。例如，端点自动配置就是这样的一种信息。要完全理解 API，您必须阅读头文件并使用示例源代码（例如“Gadget Zero”的源代码）。

实现一些基本驱动器功能的 API 部分是特定于正在使用的 Linux 内核版本的。2.6 及更高版本的内核包括一个驱动程序模型框架，该框架在早期内核上没有模拟。因此，该设备 API 的这些部分并不是完全可移植的。（它们在 2.4 内核上有实现，但是采用了不同的方式。）驱动程序模型状态是此 API 的另一部分，它被内核记录工具所忽略。

核心 API 并没有公开所有可能的硬件功能，只公开了最广泛可用的功能。有一些重要的硬件特性，例如设备到设备 DMA（在内存缓冲区中没有临时存储）将使用硬件特定的 API 进行添加。

此 API 允许驱动程序使用条件编译来处理不同硬件的端点功能，但并不要求这样做。硬件通常具有任意限制，涉及传输类型、寻址、包大小、缓冲和可用性。通常，这些差异只与处理设备配置和管理的“端点零”逻辑有关。此 API 支持通过端点的命名约定来有限制地运行时检测功能。许多驱动程序将能够至少部分地自动配置自己。特别地，驱动程序初始化部分通常具有自动配置逻辑，它会扫描硬件端点列表以查找与驱动程序要求匹配的端点（依靠这些约定），以消除某些常见的条件编译原因。

与 Linux-USB 主机端 API 类似，此 API 公开了 USB 消息的“大块”特性I/O 请求以一个或多个“包”为单位，包边界对驱动程序可见。与 RS-232 串行协议相比，USB 更类似于同步协议，如 HDLC（每帧 N 字节、多点寻址，主机为主站，设备为次要站），而不是异步协议（tty 样式每帧 8 个数据位，无校验位，一个停止位）。因此，例如，当控制器驱动程序实现的协议不需要包边界（和“短包”）时，控制器驱动程序不会将两个单字节写入缓冲为一个两字节的 USB IN 包中，尽管 Gadget 驱动程序可能会这样做。

#### 驱动程序生命周期

Gadget 驱动程序向硬件发出端点 I/O 请求，而无需了解硬件的许多细节，但是驱动程序设置/配置代码需要处理一些差异。使用此 API 如下

1. 为特定设备端 usb 控制器硬件注册驱动程序，例如 PCI 上的 net2280（USB 2.0）、Linux PDA 中发现的 sa11x0 或 pxa25x 等。此时，该设备在逻辑上处于 USB ch9 初始状态（附加状态），没有使用权限（因为它还不支持enum）。任何主机都不应查看该设备，因为它尚未激活主机用于检测设备的数据线拉起功能，即使 VBUS 电源可用。
2. 注册一个实现一些更高级设备功能的 gadget 驱动程序。然后，该驱动程序将绑定到一个 usb\_gadget，该驱动程序会在检测到 VBUS 之后的某个时间激活数据线拉起。
3. 硬件驱动程序现在可以开始enum了。它处理的步骤包括接受 USB 电源和 set\_address 请求。其他步骤由 gadget 驱动程序处理。如果主机开始enum之前卸载了 gadget 驱动程序模块，则跳过步骤 7 之前的步骤。
4. gadget 驱动程序的 setup() 调用基于总线接口硬件提供的内容和正在实现的功能返回 USB 说明符。这可能涉及备选设置或配置，除非硬件防止进行此类操作。对于 OTG 设备，每个配置说明符都包括一个 OTG 说明符。
5. 当 USB 主机发出 set\_configuration 调用时，gadget 驱动程序处理enum的最后一步。它启用该配置中使用的所有端点，并将所有接口设置为其默认设置。这涉及到使用硬件的端点列表，按照其说明符启用每个端点。它也可能涉及使用 usb\_gadget\_vbus\_draw，以允许从该配置允许的那样从 VBUS 中提取更多电力。对于 OTG 设备，设置配置还可能涉及通过用户界面报告 HNP 能力。
6. 进行实际工作并执行数据传输，可能涉及更改接口设置或切换到新配置，直到设备与主机断开连接。对每个端点队列任意数量的传输请求。它可能会在断开连接之前被暂停和恢复多次。在断开连接时，驱动程序回到步骤 3（上述）。
7. 当设备驱动模块被卸载时，驱动程序的unbind()回调被发出。这允许控制器驱动程序被卸载。

驱动程序通常被安排成仅通过加载设备驱动程序模块（或将其静态链接到Linux内核）允许外围设备被enum，但是一些驱动程序会延迟enum，直到某个更高级别的组件（例如用户模式守护程序）启用它。请注意，在这个最低层级别上，没有关于如何实现ep0配置逻辑的策略，除了它应该遵守USB规范。这些问题属于gadget驱动程序的领域，包括了解某些USB控制器强加的实现限制或理解可能会通过集成可重用组件构建复合设备所发生的事情。

请注意，上述生命周期对于OTG设备可能会略有不同。除了在每个配置中提供附加的OTG说明符之外，只有与HNP相关的差异特别明显于驱动程序代码中。它们涉及在SET\_CONFIGURATION请求期间报告要求，以及在某些挂起回调期间调用HNP的选项。此外，SRP稍微改变了usb\_gadget\_wakeup的语义。

#### USB 2.0章节9类型和常量

Gadget驱动程序依赖于在Linux 2.6+内核中标准定义的linux/usb/ch9.h头文件中定义的常见USB结构和常量。这些是主机端驱动程序（和usbcore）使用的相同类型和常量。

**核心对象和方法**

这些在<linux/usb/gadget.h>中声明，并由gadget驱动程序用于与USB外围控制器驱动程序交互。

**struct usb\_request**

说明一个I/O请求

1）定义

struct usb\_request {

void \*buf;

unsigned length;

dma\_addr\_t dma;

struct scatterlist \*sg;

unsigned num\_sgs;

unsigned num\_mapped\_sgs;

unsigned stream\_id:16;

unsigned no\_interrupt:1;

unsigned zero:1;

unsigned short\_not\_ok:1;

unsigned dma\_mapped:1;

void (\*complete)(struct usb\_ep \*ep, struct usb\_request \*req);

void \*context;

struct list\_head list;

int status;

unsigned actual;

};

2）成员

buf

用于数据的缓冲区。始终提供这个；一些控制器只使用 PIO，或者不对某些端点使用 DMA。

length

该数据的长度

dma

'buf' 对应的 DMA 地址。如果你不设置这个字段，而 usb 控制器需要一个，它负责映射和取消映射缓冲区。

sg

支持 SG 的控制器的散点列表。

num\_sgs

SG条目数

num\_mapped\_sgs

映射到 DMA（内部）的 SG 条目数

stream\_id

stream id，当使用USB3.0 bulk streams时

no\_interrupt

如果为真，则提示不需要完成 irq。有时对于直接由 DMA 控制器处理的深度请求队列很有帮助。

zero

如果为真，则在写入数据时，根据需要添加零长度数据包，使最后一个数据包变“短”；

short\_not\_ok

读取数据时，将短数据包视为错误（队列停止前进直到清理）。

dma\_mapped

指示请求是否已映射到 DMA（内部）

complete

请求完成时调用的函数，因此可以重新使用此请求及其缓冲区。该函数将始终在禁用中断的情况下被调用，并且它不能休眠。读取以短数据包或缓冲区填满时终止，以先到者为准。当写入终止时，一些数据字节通常仍在传输中（通常在硬件 fifo 中）。错误（对于读取或写入）会阻止队列前进，直到完成函数返回，因此任何因错误而无效的传输可能首先被出队。

context

供完成回调使用

list

供小工具驱动程序使用。

status

报告完成代码、零或负错误号。通常，故障会阻止传输队列前进，直到完成回调返回。代码“-ESHUTDOWN”表示由于设备断开连接或驱动程序禁用端点而导致完成。

actual

报告传输到/从缓冲区传输的字节数。对于读取（OUT 传输），这可能小于请求的长度。如果设置了 short\_not\_ok 标志，即使状态指示成功完成，短读取也会被视为错误。请注意，对于写入（IN 传输），当请求被报告为完成时，一些数据字节可能仍驻留在设备端 FIFO 中。

3）说明

它们是通过它们使用的端点进行分配/释放的。硬件驱动程序可以向它返回的内存中添加额外的每个请求数据，这通常避免了单独的内存分配（潜在的故障），稍后可以在请求队列中排队时使用。

请求标志影响请求处理，例如是否写入零长度数据包（“零”标志），是否应将短读取视为错误（阻止请求队列前进，“short\_not\_ok”标志），或者提示无需中断（“no\_interrupt”标志，用于深请求队列）。大容量端点可以使用任何大小的缓冲区，也可以用于中断传输。仅中断端点可以具有更少的功能。

4）注意事项

这与主机端的‘struct urb’类似，但更为简洁且提供了更多的预分配功能。

**struct usb\_ep\_caps**

端点功能说明

1）定义

struct usb\_ep\_caps {

unsigned type\_control:1;

unsigned type\_iso:1;

unsigned type\_bulk:1;

unsigned type\_int:1;

unsigned dir\_in:1;

unsigned dir\_out:1;

};

2）成员

type\_control

端点支持控制类型（保留用于ep0）。

type\_iso

端点支持等时传输。

type\_bulk

端点支持大量传输。

type\_int

端点支持中断传输。

dir\_in

端点支持输入方向。

dir\_out

端点支持输出方向。

**struct usb\_ep**

USB端点的设备端表示

1）定义

struct usb\_ep {

void \*driver\_data;

const char \*name;

const struct usb\_ep\_ops \*ops;

struct list\_head ep\_list;

struct usb\_ep\_caps caps;

bool claimed;

bool enabled;

unsigned maxpacket:16;

unsigned maxpacket\_limit:16;

unsigned max\_streams:16;

unsigned mult:2;

unsigned maxburst:5;

u8 address;

const struct usb\_endpoint\_descriptor \*desc;

const struct usb\_ss\_ep\_comp\_descriptor \*comp\_desc;

};

2）成员

driver\_data

供功能驱动程序使用。

name

端点的标识符，如“ep-a”或“ep9in-bulk”。

ops

用于访问硬件特定操作的函数指针。

ep\_list

设备所支持的所有端点均存储在该设备的ep\_list中。

caps

说明端点支持类型和方向的结构。

claimed

如果该端点已被功能声明，则值为true。

enabled

当前端点的启用/禁用状态。

maxpacket

在该端点上使用的最大分组大小。根据用于配置该端点的端点说明符，初始值有时可能会降低（如果硬件允许）。

maxpacket\_limit

该端点可以处理的最大分组大小值。当端点被初始化时，由UDC驱动程序设置一次，注意不要与maxpacket混淆。

max\_streams

该端点支持的最大流数（0-16，实际数字为2^n）。

mult

SS Isoc EPs的‘mult’值。

maxburst

该端点支持的最大突发数（用于USB3）。

address

用于标识端点，以查找与连接速度相匹配的说明符。

desc

端点说明符。在启用端点之前设置该指针，并在禁用端点之前保持有效。

comp\_desc

在支持SuperSpeed的情况下，这是用于配置端点的端点配套说明符。

3）说明

总线控制器驱动程序列出了设备的所有通用端点，其中控制端点（gadget->ep0）不在该列表中，仅在驱动程序setup()回调中响应时访问。

**struct usb\_gadget**

表示USB从属设备

1）定义

struct usb\_gadget {

struct work\_struct work;

struct usb\_udc \*udc;

const struct usb\_gadget\_ops \*ops;

struct usb\_ep \*ep0;

struct list\_head ep\_list;

enum usb\_device\_speed speed;

enum usb\_device\_speed max\_speed;

enum usb\_device\_state state;

const char \*name;

struct device dev;

unsigned isoch\_delay;

unsigned out\_epnum;

unsigned in\_epnum;

unsigned mA;

struct usb\_otg\_caps \*otg\_caps;

unsigned sg\_supported:1;

unsigned is\_otg:1;

unsigned is\_a\_peripheral:1;

unsigned b\_hnp\_enable:1;

unsigned a\_hnp\_support:1;

unsigned a\_alt\_hnp\_support:1;

unsigned hnp\_polling\_support:1;

unsigned host\_request\_flag:1;

unsigned quirk\_ep\_out\_aligned\_size:1;

unsigned quirk\_altset\_not\_supp:1;

unsigned quirk\_stall\_not\_supp:1;

unsigned quirk\_zlp\_not\_supp:1;

unsigned quirk\_avoids\_skb\_reserve:1;

unsigned is\_selfpowered:1;

unsigned deactivated:1;

unsigned connected:1;

unsigned lpm\_capable:1;

};

2）成员

work

（内部使用）用于sysfs\_notify()的工作队列

udc

指向该gadget的struct usb\_udc指针

ops

用于访问硬件特定操作的函数指针。

ep0

端点零，用于读取或写入驱动程序setup()请求的响应。

ep\_list

设备支持的其他端点列表。

speed

当前连接到USB主机的速度。

max\_speed

UDC能够处理的最高速度。UDC必须支持此速度和所有较慢的速度。

state

当前状态（已连接、挂起、已配置等）

name

控制器硬件类型的标识符。用于诊断和有时的配置。

dev

该抽象设备的驱动程序模型状态。

isoch\_delay

来自Set Isoch Delay请求的值。仅在SS / SSP上有效。

out\_epnum

最后一个使用的输出端点编号

in\_epnum

最后一个使用的输入端点编号

mA

上次设置的mA值

otg\_caps

该gadget的OTG功能。

sg\_supported

如果我们可以处理分散聚集，则为true

is\_otg

如果USB设备端口使用Mini-AB插孔，那么手机驱动程序必须提供USB OTG说明符，则为true。

is\_a\_peripheral

除非为is\_otg，否则USB电缆的“A”端位于Mini-AB插孔中，并使用HNP转换角色，以便“A”设备当前充当A-Periphera，而不是A-Host。

b\_hnp\_enable

OTG设备功能标志，指示A-Host启用了HNP支持。

a\_hnp\_support

OTG设备功能标志，指示A-Host支持此端口的HNP。

a\_alt\_hnp\_support

OTG设备功能标志，指示A-Host仅在不同的根端口上支持HNP。

hnp\_polling\_support

OTG设备功能标志，指示外围模式下的OTG设备是否支持HNP轮询。

host\_request\_flag

OTG 设备特征标志，指示 A-Peripheral 或 B-Peripheral 是否想要扮演主机角色。

quirk\_ep\_out\_aligned\_size

epout 需要将缓冲区大小对齐到 MaxPacketSize。

quirk\_altset\_not\_supp

UDC 控制器不支持 alt 设置。

quirk\_stall\_not\_supp

UDC 控制器不支持停止。

quirk\_zlp\_not\_supp

UDC 控制器不支持 ZLP。

quirk\_avoids\_skb\_reserve

udc/platform 希望在 u\_ether.c 中避免使用 skb\_reserve() 以提高性能。

is\_selfpowered

如果 gadget 是自供电的。

deactivated

如果 gadget 被停用，则为 True - 在停用状态下，无法连接它。

connected

如果 gadget 已连接，则为 True。

lpm\_capable

如果 gadget 的最大速度是 FULL 或 HIGH，则该标志表示它支持 LPM，如 LPM ECN 和勘误所述。

3）说明

Gadgets 具有大多数可移植的“gadget driver”，实现设备功能，处理所有 usb 配置和接口。Gadget 驱动程序通过 ops 向硬件特定代码间接交流。这使得 gadget 驱动程序免于处理硬件细节，并通过通用 i/o 队列打包硬件端点。"usb\_gadget" 和 "usb\_ep" 接口提供了与硬件的隔离。

除了驱动程序数据外，此结构中的所有字段对 gadget 驱动程序来说都是只读的。该驱动程序数据是在 2.6（及更高版本）内核中的“驱动程序模型”基础设施的一部分，对于较早的系统，则分组在一个类似的结构中，无需知道内核的其他部分。

在 USB\_REQ\_SET\_CONFIGURATION 对应的 setup() 调用之前，以及在驱动程序 suspend() 调用之前，会更新三个 OTG 设备特征标志的值。它们仅在 is\_otg，以及设备作为 B-Peripheral（因此 is\_a\_peripheral 为 false）时才有效。

**size\_t usb\_ep\_align(struct usb\_ep \*ep, size\_t len)**

返回已将 len 对齐到 ep 的最大包长度。

1）参数

struct usb\_ep \*ep

用于对齐 len 的端点

size\_t len

对齐到 ep 的最大包长度的缓冲区大小的长度

2）说明

此 helper 用于将缓冲区大小对齐到 ep 的最大包长度。

**size\_t usb\_ep\_align\_maybe(struct usb\_gadget \*g, struct usb\_ep \*ep, size\_t len)**

如果 gadget 需要 quirk\_ep\_out\_aligned\_size，则返回已将 len 对齐到 ep 的最大包长度，否则返回 len。

1）参数

struct usb\_gadget \*g

检查 quirk 的控制器

struct usb\_ep \*ep

用于对齐 len 的端点

size\_t len

对齐到 ep 的最大包长度的缓冲区大小的长度

2）说明

这个 helper 用于在任何情况下都需要检查并可能将缓冲区大小对齐到 ep 的最大包长度。

**int gadget\_is\_altset\_supported(struct usb\_gadget \*g)**

如果硬件支持 altsettings，则返回 true

1）参数

struct usb\_gadget \*g

检查 quirk 的控制器

**int gadget\_is\_stall\_supported(struct usb\_gadget \*g)**

如果硬件支持停止，则返回 true

1）参数

struct usb\_gadget \*g

检查 quirk 的控制器

**int gadget\_is\_zlp\_supported(struct usb\_gadget \*g)**

如果硬件支持 zlp，则返回 true

1）参数

struct usb\_gadget \*g

检查 quirk 的控制器

**int gadget\_avoids\_skb\_reserve(struct usb\_gadget \*g)**

如果硬件要想避免使用 skb\_reserve 来提高性能，则返回 true。

参数

struct usb\_gadget \*g

检查 quirk 的控制器

**int gadget\_is\_dualspeed(struct usb\_gadget \*g)**

如果硬件处理高速数据，则返回 true

参数

struct usb\_gadget \*g

可能支持高速和全速的控制器

**int gadget\_is\_superspeed(struct usb\_gadget \*g)**

如果硬件处理超速数据，则返回 true

参数

struct usb\_gadget \*g

可能支持超速的控制器

**int gadget\_is\_superspeed\_plus(struct usb\_gadget \*g)**

如果硬件处理超速加，则返回 true

参数

struct usb\_gadget \*g

可能支持超速加的控制器

**int gadget\_is\_otg(struct usb\_gadget \*g)**

如果硬件已准备好进行 OTG，则返回 true

1）参数

struct usb\_gadget \*g

可能有 Mini-AB 连接器的控制器

2）说明

这是运行时测试，因为带有 USB-OTG 堆栈的内核有时会在只有 Mini-B（或 Mini-A）连接器的板上运行。

**struct usb\_gadget\_driver**

USB gadget 设备的驱动程序

1）定义

struct usb\_gadget\_driver {

char \*function;

enum usb\_device\_speed max\_speed;

int (\*bind)(struct usb\_gadget \*gadget, struct usb\_gadget\_driver \*driver);

void (\*unbind)(struct usb\_gadget \*);

int (\*setup)(struct usb\_gadget \*, const struct usb\_ctrlrequest \*);

void (\*disconnect)(struct usb\_gadget \*);

void (\*suspend)(struct usb\_gadget \*);

void (\*resume)(struct usb\_gadget \*);

void (\*reset)(struct usb\_gadget \*);

struct device\_driver driver;

char \*udc\_name;

struct list\_head pending;

unsigned match\_existing\_only:1;

};

2）成员

function

说明 gadget 功能的字符串

max\_speed

驱动程序处理的最高速度。

bind

驱动程序的 bind 回调

unbind

从 gadget 卸载驱动程序时调用，通常是来自 rmmod 的断开连接报告后调用。在允许睡眠的上下文中调用。

setup

对于硬件级驱动程序未处理的 ep0 控制请求进行调用，包括说明符和配置管理在内的大多数调用必须由 gadget 驱动程序处理。setup 数据的 16 位成员以 USB 字节顺序呈现。在中断时被调用；这可能不会睡觉。驱动程序将响应排队到 ep0，或返回负数以停止。

disconnect

设备断开连接时调用。

suspend

设备挂起时调用。

resume

设备恢复时调用。

reset

设备复位时调用。

driver

驱动程序模型中的设备驱动程序

udc\_name

UDC 名称

pending

等待队列头

match\_existing\_only

仅匹配现有项的标志（用于 UDC 驱动程序）

1. 说明

设备被禁用，直到小工具驱动程序成功请求枚举（并满足“第 9 章”要求）然后做一些有用的工作。bind()`s, which means the driver will handle :c:func:`setup()

如果 gadget->is\_otg 为真，gadget 驱动程序必须在枚举期间提供 OTG 描述符，否则调用失败bind()。在这种情况下，没有 USB 流量可能会流动，直到两者都bind()没有调用而返回usb\_gadget\_disconnect()，并且 USB 主机堆栈已初始化。

驱动程序使用特定于硬件的知识来配置 USB 硬件。端点寻址只是 ep0 实现从setup()调用返回的描述符中的几个硬件特征之一。

除了 ep0 实现之外，大多数驱动程序代码不需要更改即可在不同的 usb 控制器上运行。它将使用由 ep0 实现设置的端点。

USB 控制器驱动程序处理一些标准的 USB 请求。这些包括设备、接口和端点的 set\_address 和功能标志（get\_status、set\_feature 和 clear\_feature 请求）。

因此，驱动程序的setup()回调必须始终实现所有 get\_descriptor 请求，至少返回一个设备描述符和一个配置描述符。驱动程序必须确保端点描述符匹配任何硬件约束。一些硬件还约束其他描述符。（pxa250 仅允许配置 1、2 或 3）。

驱动程序的setup()回调还必须实现 set\_configuration，还应该实现 set\_interface、get\_configuration 和 get\_interface。设置配置（或接口）是端点应该被激活或（config 0）关闭的地方。

（请注意，仅支持默认控制端点。除了 ep0 之外，主机和设备通常都不支持控制流量。）

大多数设备会忽略 USB 挂起/恢复操作，因此不会提供这些回调。但是，当主持人不再指导这些活动时，有些人可能需要更改模式。例如，本地控件（按钮、转盘等）可能需要重新启用，因为（远程）主机不能再这样做；或者可能会清除错误状态，以使设备无论是否保持电源都具有相同的行为。

**int usb\_gadget\_probe\_driver(struct [usb\_gadget\_driver](https://www.kernel.org/doc/html/v4.19/driver-api/usb/gadget.html" \l "c.usb_gadget_driver" \o "usb_gadget_driver) \* driver)**

探测小工具驱动程序

1）参数

struct usb\_gadget\_driver \* driver

正在注册的司机

2）语境

可以睡觉

3）描述

在您的小工具驱动程序的模块初始化函数中调用它，以告知底层 usb 控制器驱动程序有关您的驱动程序的信息。在此注册调用返回之前，将调用 bind() 函数以将其绑定到小工具。预计bind()函数将在 init 部分中。

**int usb\_gadget\_unregister\_driver（struct usb\_gadget\_driver \*driver）**

取消注册小工具驱动程序

1）参数

struct usb\_gadget\_driver \*driver

正在取消注册的驱动程序

2）上下文

可以休眠

3）说明

在您的小工具驱动程序的模块清理函数中调用此函数，告诉底层USB控制器，您的驱动程序正在消失。如果控制器连接到USB主机，则首先它会执行disconnect（）。在此过程最终返回之前，也会请求驱动程序进行解绑（unbind（））并清理任何设备状态。预计unbind（）函数将在退出部分中，因此在某些内核中可能无法链接。

**struct usb\_string**

封装C字符串及其USB ID

1）定义

struct usb\_string {

u8 id;

const char \*s;

};

2）成员

id

此字符串的（非零）ID

s

字符串，使用UTF-8编码

3）说明

如果您正在使用usb\_gadget\_get\_string（），请使用此函数包装字符串及其ID。

**struct usb\_gadget\_strings**

给定语言中的一组USB字符串

1）定义

struct usb\_gadget\_strings {

u16 language;

struct usb\_string \*strings;

};

2）成员

language

标识字符串的语言（en-us为0x0409）

strings

带有其ID的字符串数组

3）说明

如果您正在使用usb\_gadget\_get\_string（），请使用此函数包装给定语言的所有字符串。

**void usb\_free\_descriptors(struct usb\_descriptor\_header \*\*v)**

返回的自由描述符usb\_copy\_descriptors()

参数

struct usb\_descriptor\_header \*\*v

描述符向量

#### 可选工具

核心API足以编写USB Gadget驱动程序，但提供了一些可选工具以简化常见任务。这些实用程序包括端点自动配置。

**int usb\_gadget\_get\_string(const struct usb\_gadget\_strings \*table，int id，u8 \*buf)**

填充字符串说明符

1）参数

const struct usb\_gadget\_strings \*table

使用UTF-8编码的c字符串

int id

字符串ID，来自getStringDescriptor的wValue的低字节

u8 \*buf

至少256字节，必须为16位对齐

2）说明

查找与ID匹配的UTF-8字符串，并将其转换为utf16-le中的字符串说明符。返回说明符的长度（始终为偶数）或负errno。如果您的驱动程序需要多种语言的字符串，您可能会在ep0字符串说明符逻辑中“switch(wIndex){...}”使用此例程，使用完后选择要使用的UTF-8字符串集。请注意，US-ASCII是UTF-8的严格子集；任何具有第八位设置的字符串字节将是多字节的UTF-8字符，而不是ISO-8859 / 1字符（也广泛用于C字符串）。

**int usb\_descriptor\_fillbuf(void \*buf，unsigned buflen，const struct usb\_descriptor\_header \*\*src)**

使用说明符填充缓冲区

1）参数

void \*buf

要填充的缓冲区

unsigned buflen

buf的大小

const struct usb\_descriptor\_header \*\*src

以空指针结尾的说明符指针数组。

2）说明

将说明符复制到缓冲区中，如果不能全部复制它们，则返回长度或否定的错误代码。在为作为配置组合设备配置的一部分所使用的一组接口组装说明符或在其他情况下需要组合一组说明符时非常有用。

**int usb\_gadget\_config\_buf(const struct usb\_config\_descriptor \*config，void \*buf，unsigned length，const struct usb\_descriptor\_header \*\*desc)**

构建完整的配置说明符

1）参数

const struct usb\_config\_descriptor \*config

说明符的头，包括特征，如功率要求和接口数。

void \*buf

用于结果配置说明符的缓冲区。

unsigned length

缓冲区的长度。如果不足以容纳整个配置说明符，则会返回错误代码。

const struct usb\_descriptor\_header \*\*desc

指向定义此设备配置中的所有功能（接口、端点等）的说明符的指针的空指针向量。

2）说明

将说明符复制到响应缓冲区中，构建该配置的说明符。返回缓冲区长度或否定状态代码。config.wTotalLength字段设置为与结果的长度匹配，但必须由调用程序设置其他说明符字段（包括功率使用和接口计数）。

Gadget驱动程序可以在响应USB\_REQ\_GET\_DESCRIPTOR时使用它来构建配置说明符。如果需要USB\_DT\_OTHER\_SPEED\_CONFIG，则它们将需要修补结果的bDescriptorType值。

**struct usb\_descriptor\_header \*\*usb\_copy\_descriptors(struct usb\_descriptor\_header \*\*src)**

复制USB说明符向量

1）参数

struct usb\_descriptor\_header \*\*src

要复制的以空指针结尾的向量

2）上下文

可能会休眠的初始化代码

3）说明

这会复制USB说明符向量。它的主要用途是支持usb\_function对象，每个对象可以有多个副本，每个副本需要不同的说明符。函数可以具有静态说明符表，这些说明符表用作模板，并根据给定函数实例所需的标识符（对于接口、字符串、端点等）进行自定义。

#### 组合设备框架

核心API足以编写组合USB设备（在给定配置中具有多个功能）的驱动程序，以及多配置设备（也具有多个功能，但不一定共享给定配置）的驱动程序。但是，有一个可选的框架可使重用和组合功能更容易。

使用此框架的设备提供一个struct usb\_composite\_driver，它又提供一个或多个struct usb\_configuration实例。每个这样的配置包括至少一个结构usb\_function，它封装了一个用户可见的角色，例如“网络链接”或“大容量存储设备”。还可能存在管理功能，例如“设备固件升级”。

**struct usb\_os\_desc\_ext\_prop**

说明一个“扩展属性”

1）定义

struct usb\_os\_desc\_ext\_prop {

struct list\_head entry;

u8 type;

int name\_len;

char \*name;

int data\_len;

char \*data;

struct config\_item item;

};

2）成员

entry

用于保留扩展属性列表

type

扩展属性类型

name\_len

扩展属性Unicode名称长度，包括终止符'0'

name

扩展属性名称

data\_len

扩展属性blob的长度（用于unicode存储双倍len）

data

扩展属性blob

item

在configfs中表示此扩展属性

**struct usb\_os\_desc**

说明与一个接口相关的OS说明符

1）定义

struct usb\_os\_desc {

char \*ext\_compat\_id;

struct list\_head ext\_prop;

int ext\_prop\_len;

int ext\_prop\_count;

struct mutex \*opts\_mutex;

struct config\_group group;

struct module \*owner;

};

2）成员

ext\_compat\_id

由“兼容ID”和“子兼容ID”的16个字节组成

ext\_prop

扩展属性列表

ext\_prop\_len

扩展属性块的总长度

ext\_prop\_count

扩展属性的数量

opts\_mutex

用于保护usb\_function\_instance配置数据的可选互斥锁

group

表示与configfs中接口相关的操作系统说明符

owner

与这个操作系统说明符相关的模块

**struct usb\_os\_desc\_table**

说明与usb\_function一个接口相关的操作系统说明符

1）定义

struct usb\_os\_desc\_table {

int if\_id;

struct usb\_os\_desc \*os\_desc;

};

2）成员

if\_id

接口ID

os\_desc

接口的“扩展兼容性ID”和“扩展属性”

3）说明

每个接口最多只有一个“扩展兼容性ID”，以及一些“扩展属性”。

**struct usb\_function**

说明配置中的一个功能

1）定义

struct usb\_function {

const char \*name;

struct usb\_gadget\_strings \*\*strings;

struct usb\_descriptor\_header \*\*fs\_descriptors;

struct usb\_descriptor\_header \*\*hs\_descriptors;

struct usb\_descriptor\_header \*\*ss\_descriptors;

struct usb\_descriptor\_header \*\*ssp\_descriptors;

struct usb\_configuration \*config;

struct usb\_os\_desc\_table \*os\_desc\_table;

unsigned os\_desc\_n;

int (\*bind)(struct usb\_configuration \*, struct usb\_function \*);

void (\*unbind)(struct usb\_configuration \*, struct usb\_function \*);

void (\*free\_func)(struct usb\_function \*f);

struct module \*mod;

int (\*set\_alt)(struct usb\_function \*, unsigned interface, unsigned alt);

int (\*get\_alt)(struct usb\_function \*, unsigned interface);

void (\*disable)(struct usb\_function \*);

int (\*setup)(struct usb\_function \*, const struct usb\_ctrlrequest \*);

bool (\*req\_match)(struct usb\_function \*,const struct usb\_ctrlrequest \*, bool config0);

void (\*suspend)(struct usb\_function \*);

void (\*resume)(struct usb\_function \*);

int (\*get\_status)(struct usb\_function \*);

int (\*func\_suspend)(struct usb\_function \*, u8 suspend\_opt);

};

2）成员

name

用于诊断，标识该功能。

strings

字符串表，由bind()分配的标识符和控制请求中提供的语言ID进行标识键入。

fs\_descriptors

完整（或低速）说明符表，使用bind()分配的接口和字符串标识符。如果该指针为空，则该功能将不可在全速（或低速）使用。

hs\_descriptors

高速说明符表，使用bind()分配的接口和字符串标识符。如果该指针为空，则该功能将不可在高速使用。

ss\_descriptors

超级速度说明符表，使用bind()分配的接口和字符串标识符。如果在初始化后该指针为空，则该功能将不可在超级速度下使用。

ssp\_descriptors

超级速度加说明符表，使用bind()分配的接口和字符串标识符。如果该指针在初始化后为空，则该功能将不可在超级速度加下使用。

config

在调用usb\_add\_function()时分配；这是此功能所关联的配置。

os\_desc\_table

（接口ID，操作系统说明符）对的表。该功能可以公开多个接口。如果接口是IAD的成员，则仅在表格中具有IAD的第一个接口的条目。

os\_desc\_n

os\_desc\_table中的条目数

bind

在gadget可以注册之前，所有的功能（包括interface或class说明符中使用的字符串和接口标识符；端点；I/O缓冲区等）都应该绑定到可用资源中。

unbind

将bind反转；当取消注册添加此功能的驱动程序时，会作为副作用调用。

free\_func

释放struct usb\_function。

mod

（内部）指向创建此结构的模块。

set\_alt

（必需）重新配置altsettings；功能驱动程序可以在此时初始化usb\_ep.driver数据（当使用时）。请注意，将接口设置为当前的altsetting将重置接口状态，并且所有接口都有一个禁用状态。

get\_alt

返回活动的altsetting。如果没有提供此内容，则仅支持altsetting零。

disable

（必需）表示应禁用该功能。原因包括主机重置或重新配置gadget，以及断开连接。

setup

用于特定于接口的控制请求。

req\_match

测试此功能是否可以处理给定的类请求。

suspend

当主机停止发送USB流量时通知功能。

resume

当主机重新启动USB流量时通知功能。

get\_status

作为回复GetStatus()请求的函数状态，当接收方为Interface时。

func\_suspend

在收到SetFeature(FUNCTION\_SUSPEND)时调用的回调

3）说明

单个USB功能使用一个或多个接口，并且在大多数情况下应支持全速和高速操作。通过usb\_add\_function()与一个配置相关联的每个功能都会导致调用bind()以便在设置gadget驱动程序时分配资源。这些资源包括端点，应使用usb\_ep\_autoconfig()进行分配。

为了支持双速操作，功能驱动程序提供了高速和全速操作的说明符。除了不涉及批量端点的罕见情况外，在每个速度下需要不同的端点说明符。

功能驱动程序选择其自己的策略来管理实例数据。最简单的策略只是声明它“静态”，这意味着该功能只能被激活一次。如果要在给定速度的超过一个配置中公开该功能，则需要支持多个usb\_function结构（每个配置一个）。

更复杂的策略可能会将usb\_function结构封装在驱动程序特定的实例结构中，以允许多次激活。多个激活示例可能是支持在同一配置中具有两个或多个不同实例的CDC ACM功能，为USB主机提供多个独立的逻辑数据链路。

**struct usb\_configuration**

表示一个 gadget 配置name

struct usb\_configuration {

const char \*label;

struct usb\_gadget\_strings \*\*strings;

const struct usb\_descriptor\_header \*\*descriptors;

void (\*unbind)(struct usb\_configuration \*);

int (\*setup)(struct usb\_configuration \*, const struct usb\_ctrlrequest \*);

u8 bConfigurationValue;

u8 iConfiguration;

u8 bmAttributes;

u16 MaxPower;

struct usb\_composite\_dev \*cdev;

};

2）成员

label

用于诊断，说明配置。

strings

字符串表，由 bind() 分配的标识符和控制请求中提供的语言 ID 键控。

descriptors

位于所有函数说明符之前的说明符表。例如，OTG 和供应商特定说明符等。

unbind()

反向绑定；作为注销添加此配置的驱动程序的副作用调用。

setup()

用于委派未由标准设备结构处理或针对特定接口的控制请求。

bConfigurationValue

复制到配置说明符。

iConfiguration

复制到配置说明符。

bmAttributes

复制到配置说明符。

MaxPower

功耗，单位为毫安。考虑总线速度后，用于计算配置说明符中的 bMaxPower。

cdev

由 usb\_add\_config() 分配给 bind() 前；这是与此配置相关联的设备。

3）介绍

配置是围绕功能驱动程序构建的小工具驱动程序的组成部分。简单的 USB gadgets 仅需要一个功能和一个配置，并通过始终提供相同的功能处理双速硬件。稍微复杂一些的 gadgets 可能具有在给定速度下的多个单功能配置；或者具有仅在一个速度下工作的配置。

组合设备定义为包括多个功能的配置。

usb\_configuration 的生命周期包括分配、初始化上述字段和调用 usb\_add\_config() 来设置内部数据并将其绑定到特定设备。然后使用配置的 bind() 方法来初始化所有功能，然后对它们调用 usb\_add\_function()。

这些功能通常彼此独立，但这并非强制性的。CDC WMC 设备是功能经常依赖于其他功能的示例，有些功能是其他功能的附属物。这种相互依赖可以以任何方式进行管理，只要当组合驱动程序从其 bind() 例程返回时，所有说明符都完成即可。

**struct usb\_composite\_driver**

将配置分组为小工具

1）定义

struct usb\_composite\_driver {

const char \*name;

const struct usb\_device\_descriptor \*dev;

struct usb\_gadget\_strings \*\*strings;

enum usb\_device\_speed max\_speed;

unsigned needs\_serial:1;

int (\*bind)(struct usb\_composite\_dev \*cdev);

int (\*unbind)(struct usb\_composite\_dev \*);

void (\*disconnect)(struct usb\_composite\_dev \*);

void (\*suspend)(struct usb\_composite\_dev \*);

void (\*resume)(struct usb\_composite\_dev \*);

struct usb\_gadget\_driver gadget\_driver;

};

2）成员

name

用于诊断，标识驱动程序。

dev

设备的模板说明符，包括默认设备标识符。

strings

由 bind 分配的标识符和控制请求中提供的语言 ID 键控的字符串表。注意第一项是预定义的。可以使用的第一项是 USB\_GADGET\_FIRST\_AVAIL\_IDX。

max\_speed

驱动程序支持的最高速度。

needs\_serial

如果小工具需要用户空间提供序列号，则设置为 1。如果未提供序列号，将打印警告。

bind()

（必填）用于分配整个设备共享的资源，例如字符串 ID，并使用 usb\_add\_config() 添加其配置。此操作可能通过返回负 errno 值而失败；在成功初始化时应返回零。

unbind()

反向绑定；作为注销此驱动程序时的副作用调用。

disconnect()

可选的驱动程序断开方法

suspend()

在功能通知后，通知主机停止发送 USB 流量时使用

resume()

在功能通知前，通知主机重新启动 USB 流量时使用。

gadget\_driver

控制此驱动程序的小工具驱动程序

3）介绍

设备默认报告自供电操作。依靠总线供电的设备应在它们的 bind 方法中报告这一点。

在从 bind 返回之前，可以覆盖模板说明符中的各个字段。这些字段包括正常情况下用于绑定适当的主机侧驱动程序的 idVendor、idProduct 和 bcdDevice 值，以及用于提供用户有意义的设备标识符的三个字符串（iManufacturer、iProduct 和 iSerialNumber）。 （字符串只有在 dev 和字符串中定义它们时才定义。）还报告正确的 ep0 最大包大小，由底层控制器驱动定义。

**module\_usb\_composite\_driver**

module\_usb\_composite\_driver (\_\_usb\_composite\_driver)

用于注册 USB gadget 组合驱动程序的辅助宏

1）参数

\_\_usb\_composite\_driver

usb\_composite\_driver struct

2）介绍

用于 USB gadget 组合驱动程序的辅助宏，它在模块 init/exit 中不执行任何特殊操作。这消除了许多样板文件。每个模块只能使用此宏一次，并且调用它将替换 module\_init() 和 module\_exit()。

**struct usb\_composite\_dev**

表示一个组合 USB gadget

1）定义

struct usb\_composite\_dev {

struct usb\_gadget \*gadget;

struct usb\_request \*req;

struct usb\_request \*os\_desc\_req;

struct usb\_configuration \*config;

u8 qw\_sign[OS\_STRING\_QW\_SIGN\_LEN];

u8 b\_vendor\_code;

struct usb\_configuration \*os\_desc\_config;

unsigned int use\_os\_string:1;

unsigned int setup\_pending:1;

unsigned int os\_desc\_pending:1;

};

2）成员

gadget

只读，抽象了 gadget 的 usb 外围控制器

req

用于控制响应；缓冲区是预分配的

os\_desc\_req

用于操作系统说明符响应；缓冲区是预分配的

config

当前活动的配置

qw\_sign

OS字符串的qwSignature部分

b\_vendor\_code

OS字符串的bMS\_VendorCode部分

os\_desc\_config

要与操作系统说明符一起使用的配置

use\_os\_string

默认为false，感兴趣的小工具可以设置它

setup\_pending

当设置请求排队但未完成时为true

os\_desc\_pending

当os\_desc请求排队但未完成时为true

3）说明

在调用相关设备驱动程序的bind（）之前，将分配并初始化其中一个设备。

未解决的问题：似乎某些WUSB设备将需要通过将正常（有线）小工具与无线小工具组合来构建。该小工具框架的这个版本应该尝试确保这样做不会太糟糕。

处理无线USB设备的一种想法涉及：

第二个小工具在此处，发现机制TBD，但可能需要单独的“注册/注销WUSB小工具”调用；

更新usb\_gadget以包括标志“它是无线的”，“它是有线的”，以及（可能在包装结构中）波段组和物理信息；

假设一个wireless\_ep包装一个usb\_ep，并报告最大突发和最大序列等无线特定参数；

特定于无线链路的配置；

了解无线配置并支持附加功能实例的功能驱动程序；

支持协会设置（如CBAF）的函数，不一定需要无线适配器；

可以创建一个或多个无线配置，包括适当的协会设置支持的组合设备设置；

更多，待定。

**int config\_ep\_by\_speed（struct usb\_gadget \*g，struct usb\_function \*f，struct usb\_ep \* \_ep）**

根据小工具速度配置给定的端点。

1）参数

struct usb\_gadget \* g

指向小工具的指针

struct usb\_function \* f

USB功能

struct usb\_ep \*\_ep

要配置的端点

2）返回

错误代码，成功返回0

此函数根据小工具速度选择正确的说明符并将其保存在端点说明符字段中。如果端点已经分配了说明符，则将其覆盖为当前相应的说明符。端点maxpacket字段根据选择的说明符进行更新。

3）注意

提供的函数应保存所有支持速度的说明符

**int usb\_add\_function（struct usb\_configuration \* config，struct usb\_function \* function）**

将函数添加到配置中

1）参数

struct usb\_configuration \* config

配置

struct usb\_function \* function

正在添加的函数

2）上下文

单线程的小工具设置期间

3）说明

初始化后，每个配置必须添加一个或多个函数。添加函数涉及调用其bind（）方法以分配资源，例如接口和字符串标识符和端点。

该函数返回函数的bind（）的值，其成功为零，否则为负的errno值。

**int usb\_function\_deactivate（struct usb\_function \* function）**

防止函数和小工具enum

1）参数

struct usb\_function \* function

尚未准备好响应的功能

2）说明

通过阻止数据线拉上来激活阻止设备驱动程序响应主机enum的操作。这通常在bind（）处理期间调用，以从初始的“准备响应”状态更改，或者当所需资源可用时。

例如，充当与用户空间守护程序的对通过驱动程序可以在该守护程序（例如OBEX，MTP或打印服务器）准备好处理主机请求之前阻止enum。

并非所有系统都支持对其USB外设数据拉起的软件控制。

成功返回零，否则为负errno。

**int usb\_function\_activate（struct usb\_function \* function）**

允许函数和工具enum

1）参数

struct usb\_function \* function

在其上调用usb\_function\_activate（）的函数

2）说明

反转usb\_function\_deactivate（）的影响。如果没有更多的功能延迟其激活，则设备驱动程序将响应主机enum程序。

成功返回零，否则为负errno。

**int usb\_interface\_id（struct usb\_configuration \* config，struct usb\_function \* function）**

分配未使用的接口ID

1）参数

struct usb\_configuration \* config

与接口相关联的配置

struct usb\_function \* function

处理接口的功能

2）上下文

单线程的小工具设置期间

3）说明

从usb\_function.bind（）回调中调用usb\_interface\_id（）以分配新的接口ID。然后，函数驱动程序将在接口、组合、CDC union和其他说明符中存储该ID。它还将处理针对该接口的任何控件请求，特别是通过set\_alt（）更改其altsetting。还可能存在类特定或供应商特定的请求要处理。

所有接口标识符都应使用此程序分配，以确保例如不同的函数不会错误地将不同的含义分配给相同的标识符。请注意，由于接口标识符是特定于配置的，因此在多个配置中（或在给定配置中多次使用）使用的函数需要相关说明符的多个版本。

返回分配的接口ID；若无法再分配接口ID，则返回-ENODEV。

**int usb\_add\_config(struct usb\_composite\_dev \*cdev, struct usb\_configuration \*config, int (\*bind)(struct usb\_configuration\*))**

向设备添加配置。

1）参数

struct usb\_composite\_dev \*cdev

封装了USB gadget。

struct usb\_configuration \*config

配置，带有已分配的bConfigurationValue。

int (\*bind)(struct usb\_configuration \*)

配置的绑定函数。

2）上下文

在合成设备设置期间是单线程的。

3）说明

复合绑定（bind）程序的主要任务之一是使用此程序添加其支持的每个配置。

此函数返回配置的bind()的值，对于成功为零，否则为负的错误值。绑定配置会分配全局资源，包括字符串ID，以及每个配置资源，例如接口ID和端点。

**int usb\_string\_id(struct usb\_composite\_dev \*c**dev)

分配未使用的字符串ID。

1）参数

struct usb\_composite\_dev \*cdev

正在分配其字符串说明符ID的设备。

2）上下文

在合成设备设置期间是单线程的。

3）说明

从绑定（bind）回调调用usb\_string\_id()来分配字符串ID。功能、配置或设备的驱动程序将在适当的说明符和字符串表中存储该ID。

所有字符串标识符都应使用此usb\_string\_id()，usb\_string\_ids\_tab()或usb\_string\_ids\_n()例程进行分配，以确保例如不同函数不会错误地分配不同含义的相同标识符。

**int usb\_string\_ids\_tab(struct usb\_composite\_dev \*cdev, struct usb\_string \*str)**

分批分配未使用的字符串ID。

1）参数

struct usb\_composite\_dev \*cdev

正在分配其字符串说明符ID的设备。

struct usb\_string \*str

一个usb\_string对象数组，用于指定编号。

2）上下文

在合成设备设置期间是单线程的。

3）说明

从绑定（bind）回调调用usb\_string\_ids()来分配字符串ID。功能、配置或设备的驱动程序将在其他语言的适当说明符和字符串表中将ID从字符串表复制到。

所有字符串标识符都应使用此usb\_string\_ids\_tab()，usb\_string\_id()或usb\_string\_ids\_n()例程进行分配，以确保例如不同函数不会错误地分配不同含义的相同标识符。

**struct usb\_string \*usb\_gstrings\_attach(struct usb\_composite\_dev \*cdev, struct usb\_gadget\_strings \*\*sp, unsigned n\_strings)**

将设备的字符串说明符和id关联起来。

1）参数

struct usb\_composite\_dev \*cdev

正在分配和附加字符串说明符id的设备。

struct usb\_gadget\_strings \*\*sp

要附加的usb\_gadget\_strings数组。

unsigned n\_strings

每个usb\_strings数组条目（sp [] -> strings）中的条目数。

2）说明

此函数将创建usb\_gadget\_strings和usb\_string的深层副本，并将其附加到cdev。实际的字符串（usb\_string.s）将不被复制，但只会做出引用。struct usb\_gadget\_strings数组可以包含多种语言，并应以空终止。每个struct usb\_gadget\_strings的->language指针必须包含相同数量的条目。例如sp [0]是en-US，sp [1]是es-ES。预计es-ES的第一个usb\_string条目包含en-US的第一个usb\_string条目的翻译。因此，这两个条目将分配相同的id。

**int usb\_string\_ids\_n(struct usb\_composite\_dev \*c, unsigned n)**

分批分配未使用的字符串ID。

1）参数

struct usb\_composite\_dev \*c

正在分配其字符串说明符ID的设备。

unsigned n

要分配的字符串ID数。

2）上下文

在合成设备设置期间是单线程的。

3）说明

返回第一个请求的ID。此ID和接下来的n-1个ID现在是有效的ID。至少，只要n是非零的，因为如果是，则返回最后一个请求的ID，此时非常有用。

从绑定（bind）回调调用usb\_string\_ids\_n()来分配字符串ID。功能、配置或设备的驱动程序将在适当的说明符和字符串表中存储该ID。

所有字符串标识符都应使用此usb\_string\_id()，usb\_string\_ids\_tab()或usb\_string\_ids\_n()例程进行分配，以确保例如不同函数不会错误地分配不同含义的相同标识符。

**int usb\_composite\_probe(struct usb\_composite\_driver \*driver)**

注册复合驱动程序。

1）参数

struct usb\_composite\_driver \*driver

要注册的驱动程序。

2）上下文

在合成设备设置期间是单线程的。

3）说明

此函数用于使用组合驱动程序框架注册驱动程序。返回为零或负的errno值。这些值通常来自驱动程序的bind方法，该方法完成设置驱动程序以匹配硬件的所有工作。

在成功返回后，该设备已准备好响应主机的请求，除非其组件在绑定期间调用了usb\_gadget\_disconnect()。这通常是为了等待一些用户空间参与。

**void usb\_composite\_unregister(struct usb\_composite\_driver \*driver)**

注销组合驱动程序。

1）参数

struct usb\_composite\_driver \*driver

要注销的驱动程序。

2）说明

此函数用于使用组合驱动程序框架注销驱动程序。

**void usb\_composite\_setup\_continue(struct usb\_composite\_dev \*cdev)**

继续控制传输

1）参数

struct usb\_composite\_dev \*cdev

等待控制传输的复合设备

2）说明

USB功能驱动程序必须调用此函数以继续控制传输的数据/状态阶段，以防它请求延迟数据/状态阶段。 USB功能的设置处理程序（例如set\_alt（））可以通过返回USB\_GADGET\_DELAYED\_STATUS来请求复合框架延迟设置请求的数据/状态阶段。

#### 复合设备功能

在编写本文时，一些当前的小工具驱动程序已转换为此框架。近期计划包括将所有这些功能转换为此框架，除了gadgetfs以外。

### 外围控制器驱动程序

支持此API的第一个硬件是NetChip 2280控制器，它支持USB2.0高速并基于PCI。这是net2280驱动程序模块。该驱动程序支持Linux内核版本2.4和2.6；请联系NetChip Technologies了解开发板和产品信息。

在小工具框架中工作的其他硬件包括英特尔的PXA 25x和IXP42x系列处理器（pxa2xx\_udc），东芝TC86c001“Goku-S”（goku\_udc），Renesas SH7705/7727（sh\_udc），MediaQ 11xx（mq11xx\_udc），Hynix HMS30C7202（h7202\_udc），国家9303/4（n9604\_udc），德州仪器的OMAP（omap\_udc），Sharp LH7A40x（lh7a40x\_udc）等。其中大多数是全速控制器。

在编写本文时，有人正在小工具驱动程序的驱动程序框架上工作，计划使其中的许多人普遍可用。

有一个部分的USB模拟器，dummy\_hcd驱动程序是可用的。它可以像net2280，pxa25x或sa11x0一样，以可用的端点和设备速度为基础；它模拟控件、批量和在某种程度上打断转移。这让你在普通PC上开发一些小工具驱动程序的部分，而无需任何特殊硬件，可能需要使用诸如在用户模式Linux上运行的GDB等工具来辅助开发。至少有一人表达了兴趣，以适应这种方法，并将其连接到微控制器的模拟器上。这样的模拟器可以帮助调试运行时硬件对软件开发不友好或尚不可用的子系统。

预计会在时间内开发和贡献对其他控制器的支持，随着该驱动程序框架的发展。

### 小工具驱动程序

除了小工具零（主要用于测试和开发与USB控制器硬件驱动程序），还存在其他小工具驱动程序。

有一个以太网小工具驱动程序，它实现了最有用的通信设备类（CDC）模型之一。甚至有一些有关电缆调制解调器互操作性的标准规定使用这个以太网模型作为两个强制性选择之一。使用此代码的小工具像USB主机一样，看起来像是一个以太网适配器。它提供对网络的访问，其中小工具的CPU是一个主机，它可以轻松地桥接、路由或防火墙访问其他网络。由于某些硬件无法完全实现CDC以太网要求，因此该驱动程序还实现了CDC以太网的“仅良好部分”子集。 （该子集不会将自己声明为CDC以太网，以避免创建问题。）

由Pengutronix和Auerswald GmbH贡献了对Microsoft的RNDIS协议的支持。它与CDC Ethernet很像，但它在更少的USB硬件上运行（但不及CDC子集）。然而，它的主要优点是能够直接连接到Microsoft的最新版本，使用Microsoft捆绑和支持的驱动程序，使与Windows的网络变得更加简单。

还支持使用gadgetfs的用户模式小工具驱动程序。这提供了一个用户模式API，将每个端点呈现为单个文件说明符。使用普通的read（）和read（）调用完成I/O。熟悉的工具，如GDB和pthread，可用于开发和调试用户模式驱动程序，因此一旦有了强大的控制器驱动程序，许多应用程序就不需要新的内核模式软件。可用Linux 2.6异步I/O（AIO）支持，以便用户模式软件可以流数据而只略微增加开销。

有一个USB大容量存储类驱动程序，它为与MS-Windows和MacOS等系统的互操作性提供了不同的解决方案。该大容量存储驱动程序使用文件或块设备作为驱动器的后备存储器，就像环形驱动器一样。 USB主机使用大容量存储类规范的BBB、CB或CBI版本，使用透明SCSI命令从后备存储器访问数据。

有一个“串行线”驱动程序，可用于通过USB进行TTY样式操作。该驱动程序的最新版本支持类似于CDC ACM的操作，如USB调制解调器等，并且在大多数硬件上它可以轻松地与MS-Windows互操作。该驱动程序的一个有趣用途是在引导固件（如BIOS）中使用该模型，有时可以在非常小的系统中使用该模型而不需要实际的串行线。

预计随着驱动程序框架的发展，其他类型的小工具将被开发和贡献。

### USB移动 (OTG)

Linux 2.6上的USB OTG支持最初由得克萨斯仪器公司为OMAP 16xx和17xx系列处理器开发。其他OTG系统应以类似的方式工作，但硬件级别的详细信息可能会有很大的差异。

系统需要专门的硬件支持来实现OTG，特别是包括一个特殊的Mini-AB插孔和相关的收发器，以支持双重角色操作它们可以作为主机使用标准的Linux-USB主机端驱动程序堆栈，也可以作为外围设备使用此小工具框架。为此，系统软件依赖于对这些编程接口的小修改，以及新的内部组件（此处称为“OTG控制器”），影响连接到OTG端口的驱动程序堆栈。在每个角色中，系统可以重用现有的硬件中立驱动程序池，分层在控制器驱动程序接口（usb\_bus或usb\_gadget）的顶部。这样的驱动程序最多需要进行小修改，添加支持OTG的大多数调用也可以使非OTG产品受益。

1. 小工具驱动程序测试is\_otg标志，并用它来确定是否在它们的每个配置中包括OTG说明符。
2. 小工具驱动程序可能需要更改以支持两个新的OTG协议，公开在新的小工具属性中，例如b\_hnp\_enable标志。 HNP支持应通过用户界面（两个LED足以）报告，并在某些情况下在主机挂起外围设备时触发。 SRP支持可以像远程唤醒一样由用户发起，可能通过按下相同的按钮实现。
3. 在主机端，USB设备驱动程序需要在适当的时候教会触发HNP，使用usb\_suspend\_device（）。这也有利于节省电池电力，即使对于非OTG配置也很有用。
4. 同样在主机端，驱动程序必须支持OTG“定向外围设备列表”。这只是一个白名单，用于拒绝不受给定Linux OTG主机支持的外围设备。此白名单是产品特定的；每个产品都必须修改otg\_whitelist.h以符合其互操作性规范。

非OTG Linux主机（例如PC和工作站）通常有一些解决方案来添加驱动程序，以便最终可以支持未被识别的外围设备。这种方法对于可能永远不会升级其固件的消费品来说是不合理的，并且通常不太现实，希望像传统的PC /工作站/服务器支持模型那样工作。例如，在产品发货后发现驱动程序错误通常无法修复设备固件，因此不能正常工作。

在那些硬件中立的usb\_bus和usb\_gadget驱动程序接口之下需要做出其他更改；这里不会详细讨论。这些影响每个USB主机或外围控制器的硬件特定代码，以及HCD的初始化方式（因为OTG只能在单个端口上激活）。它们还涉及可能称为OTG控制器驱动程序的内容，该驱动程序管理OTG收发器和OTG状态机逻辑以及OTG端口的大部分根集线器行为。OTG控制器驱动程序需要根据相关设备角色激活和停用USB控制器。一些相关的更改需要在usbcore内部进行，以便它可以识别支持OTG的设备并适当地响应HNP或SRP协议。

## USB锚点

### 什么是锚点？

USB驱动程序需要支持一些回调，要求驱动程序停止与接口的所有IO。为此，驱动程序必须跟踪它已提交的URB，以知道它们全部完成，或者为它们调用usb\_kill\_urb。锚点是一个数据结构，负责跟踪URB并提供处理多个URB的方法。

### 分配和初始化

没有API来分配锚点。它只是声明为struct usb\_anchor。必须调用init\_usb\_anchor（）来初始化数据结构。

### 重新分配

一旦没有与其关联的URB，就可以使用普通的内存管理操作来释放锚点。

### 将URB与锚点关联和解除关联

通过显式调用usb\_anchor\_urb（）将URB与锚点相关联。该关联保持到URB完成（成功）完成。因此，解除关联是自动的。提供了一个函数，用于强制完成（杀死）与锚点关联的所有URB。此外，可以使用usb\_unanchor\_urb（）解除关联。

### 处理多个URB的操作

#### usb\_kill\_anchored\_urbs（）

此功能杀死与锚点相关联的所有URB。URB按照它们被提交的倒序调用。这样就不会重新排序数据。

#### usb\_unlink\_anchored\_urbs（）

该函数解链与锚点关联的所有 URB。 URBs 的处理顺序是它们提交的相反时间顺序。 这类似于 usb\_kill\_anchored\_urbs（），但它不会休眠。 因此，当调用返回时，不保证 URBs 已解链。 它们可能会稍后解链，但将在有限时间内解链。

#### usb\_scuttle\_anchored\_urbs（）

卸载锚点上的所有 URBs。

#### usb\_wait\_anchor\_empty\_timeout（）

该函数等待与锚点关联的所有 URBs 完成或超时，以先到达者为准。它的返回将告诉您是否达到了超时时间。

#### usb\_anchor\_empty（）

如果没有 URBs 与锚点关联，则返回 true。 锁定由调用者负责。

#### usb\_get\_from\_anchor（）

返回锚点的最早的锚定 URB。 URB 将解链并带有一个引用返回。由于您可能将 URB 混合到一个锚点的几个目的地中，因此不能保证返回的是按时间顺序最先提交的 URB。

## USB 批量流

### 背景

USB 3.0 规范中添加了批量端点流。 流允许设备驱动程序超载批量端点，以便可以同时排队多个传输。

流在通用串行总线 3.0 规范的第 4.4.6.4 和 8.12.1.4 节中定义，网址为 https://www.usb.org/developers/docs/ 使用流排队多个 SCSI 命令的 USB 附加 SCSI 协议可以在 T10 网站上找到 （https://t10.org/）。

### 设备端影响

一旦将缓冲区排队到流环中，设备将通过另一个端点上的带外机制被通知数据准备就绪。 然后设备告诉主机要启动哪个“流”。 主机也可以在设备请求之外启动流传输，但是设备可以拒绝该传输。 设备可以随时在流之间切换。

### 驱动程序的影响

int usb\_alloc\_streams(struct usb\_interface \*interface,

struct usb\_host\_endpoint \*\*eps, unsigned int num\_eps,

unsigned int num\_streams, gfp\_t mem\_flags);

设备驱动程序将调用此 API 请求主机控制器驱动程序分配内存，以便驱动程序可以使用达到 num\_streams 个流 ID。他们必须传递需要设置类似流 ID 的 usb\_host\_endpoints 数组。这是为了确保 UASP 驱动程序能够使用相同的流 ID 用于双向命令序列中使用的批量 IN 和 OUT 端点。

返回是错误条件（如果一个端点不支持流，或者 xHCI驱动程序的内存已用尽），或者主机控制器分配给此端点的流数。 xHCI 主机控制器硬件声明可以支持多少流 ID，每个超级速度设备上的批量端点都会说明可以处理多少个流 ID。 因此，驱动程序应该能够处理被分配的流 ID 数少于请求的情况。

如果您有 URB 排队在任何传输中，请勿调用此函数。请勿调用此函数请求少于两个流。

驱动程序将只允许针对相同端点调用一次此 API，而不调用 usb\_free\_streams（）。这是 xHCI 主机控制器驱动程序的简化，并且未来可能会发生改变。

### 选择要使用的新流 ID

流 ID 0 已保留，不应用于与设备通信。如果 usb\_alloc\_streams（）返回为 N，则可以使用流 1 到 N。要将 URB 排队到特定流，请设置urb->stream\_id 值。 如果端点不支持流，则会返回错误。

请注意，如果 xHCI 驱动程序支持辅助流 ID，则必须添加新的 API 以选择下一个流 ID。

### 整理

如果驱动程序希望停止使用流与设备通信，则应调用:

void usb\_free\_streams(struct usb\_interface \*interface,

struct usb\_host\_endpoint \*\*eps, unsigned int num\_eps,

gfp\_t mem\_flags);

当驱动程序释放接口时，所有流 ID 将被释放，以确保不支持流的驱动程序将能够使用端点。

## USB 核心回调

### USB 核心将执行哪些回调？

Usbcore 将通过在驱动程序结构中定义的回调以及驱动程序提交的 URB 的完成处理程序调用驱动程序。仅前者在本文档的范围内。这两种类型的回调是完全独立的。关于完成回调的信息可以在 USB 请求块（URB）中找到。

驱动程序结构中定义的回调包括

1）热插拔回调

@探针

调用以查看驱动程序是否愿意管理设备上的特定接口。

@断开连接

当接口不再可用时调用，通常是因为其设备已被（或正在）断开连接，或者驱动程序模块正在被卸载。

2）通过usbfs的奇怪后门

@ioctl

用于希望通过“usbfs”文件系统与用户空间进行通信的驱动程序。这使得设备能够提供将信息暴露给用户空间的方式，无论它们是否在文件系统中出现或不出现。

3）电源管理（PM）回调

@暂停

当设备即将进入暂停状态时调用。

@恢复

当设备正在恢复时调用。

@reset\_resume

当挂起的设备已经重置而不是被恢复时调用。

4）设备级操作

@重置前

当设备即将被重置时调用。

@重置后

设备重置后调用

ioctl接口（2）仅应在您有非常好的理由时使用。现在首选Sysfs。 USB的电源管理回调在Power Management for USB中单独介绍。

### 调用约定

所有回调函数都是互斥的。没有必要针对其他USB回调进行锁定。所有回调都从任务上下文中调用。你可以睡觉。但是，所有睡眠都有一个很小的固定时间上限是很重要的。特别是您不应调用用户空间并等待结果。

### 热插拔回调

这些回调旨在将驱动程序与接口相关联和取消关联。驱动程序与接口的绑定是独占的。

#### probe（）回调

int (\*probe) (struct usb\_interface \*intf,

const struct usb\_device\_id \*id);

接受或拒绝接口。如果您接受设备，则返回0，否则为-ENODEV或-ENXIO。仅在初始化期间发生真正的错误阻止了驱动程序接受否则将被接受的设备时，应使用其他错误代码。强烈建议您使用usbcore的设施usb\_set\_intfdata（）与接口相关联的数据结构，以便您知道与特定接口相关联的内部状态和身份。设备不会被暂停，您可以进行针对您调用的接口和设备端点0的IO。建议在这里进行不太长时间的设备初始化。

#### disconnect（）回调

void (\*disconnect) (struct usb\_interface \*intf);

此回调是与接口断开任何连接的信号。在从此回调返回后，您不允许对设备进行任何IO。您也不能执行可能干扰到绑定到接口的其他驱动程序的任何其他操作，例如电源管理操作。如果由于物理断开而被调用，所有URB都将被usbcore杀死。请注意，在这种情况下，断开连接将在物理断开后的一段时间内被调用。因此，您的驱动程序必须准备好在回调之前甚至处理失败的IO。

### 设备级回调

#### pre\_reset

int (\*pre\_reset)(struct usb\_interface \*intf);

驱动程序或用户空间正在触发设备上的重置，该设备包含作为参数传递的接口。停止IO，等待所有未完成的URB完成，并保存您需要恢复的任何设备状态。在调用post\_reset方法之前，不得提交任何其他URB。

如果您需要在此处分配内存，请使用GFP\_NOIO或GFP\_ATOMIC，如果您处于原子上下文中，请使用GFP\_ATOMIC。

#### post\_reset

int (\*post\_reset)(struct usb\_interface \*intf);

重置已完成。恢复任何保存的设备状态并开始再次使用设备。

如果您需要在此处分配内存，请使用GFP\_NOIO或GFP\_ATOMIC，如果您处于原子上下文中，请使用GFP\_ATOMIC。

### 调用序列

对于未绑定到您的驱动程序的接口，不会调用除probe之外的任何回调。

仅在接口绑定到驱动程序之后，disconnect才会在再次探测相同接口之前被调用。

一旦将驱动程序绑定到接口，除了pre\_reset和post\_reset之间之外，disconnect可以随时调用。无论重置失败还是设备已被拔出，pre\_reset总是由post\_reset后跟，。

暂停始终后跟以下情况之一恢复，重置恢复或断开连接。

## USB DMA

在Linux 2.5内核（及更高版本）中，USB设备驱动程序对如何使用DMA执行I / O操作具有额外的控制权。 API的详细信息在内核USB编程指南（源代码的kerneldoc）中详细说明。

### API概述

大局是USB驱动程序可以继续忽略大多数DMA问题，虽然它们仍然必须提供DMA-ready缓冲区（请参阅动态DMA映射指南）。那是它们通过2.4（以及更早）内核的工作方式，或者它们现在可以是DMA-aware。

DMA感知的USB驱动程序：

1）新的调用使DMA感知驱动程序能够分配dma缓冲区并管理现有的dma-ready缓冲区的dma映射（见下文）。

2）URBs有一个额外的“transfer\_dma”字段，以及一个transfer\_flags位，用于指示它是否有效。（控制请求也有“setup\_dma”，但驱动程序不得使用它。）

3）如果DMA感知驱动程序没有首先执行这些操作并设置URB\_NO\_TRANSFER\_DMA\_MAP，那么“usbcore”将映射此DMA地址。HCD不管理URB的DMA映射。

4）有一个新的“通用DMA API”，其中的某些部分可由USB设备驱动程序使用。永远不要在任何USB接口或设备上使用dma\_set\_mask()，因为那可能会破坏所有共享该总线的设备。

### 消除复制

避免让CPU无谓地复制数据是很好的。成本可能会累积，并且像高速缓存崩溃之类的影响会产生微妙的惩罚。

1）如果您一直从同一缓冲区传输大量小数据，那么在使用IOMMU管理DMA映射的系统上，这可能会占用大量资源。每个请求都设置和撤下IOMMU映射的成本可能比执行I/O高得多！

对于这些特定情况，USB有原始函数来分配更便宜的内存。它们的工作方式类似于kmalloc和kfree版本，可以给出要存储在urb->transfer\_buffer和urb->transfer\_dma中的正确类型的地址。您还将在urb->transfer\_flags中设置URB\_NO\_TRANSFER\_DMA\_MAP:

void \*usb\_alloc\_coherent (struct usb\_device \*dev, size\_t size,

int mem\_flags, dma\_addr\_t \*dma);

void usb\_free\_coherent (struct usb\_device \*dev, size\_t size,

void \*addr, dma\_addr\_t dma);

大多数驱动程序不应该使用这些原始函数；它们不需要使用这种类型的内存（“dma-coherent”），从kmalloc()返回的内存完全可以正常工作。

返回的内存缓冲区是“dma-coherent”的；有时您可能需要使用内存屏障来强制一致的内存访问顺序。它不使用流DMA映射，因此在I/O会否毁掉IOMMU映射的系统上进行小型传输很好（有关“连贯”和“流媒体”DMA映射的定义，请参见动态DMA映射指南）。

请求1/N页（以及请求N页）是相当节省空间的。

在大多数系统上，返回的内存将是未缓存的，因为DMA一致内存的语义要么绕过CPU缓存，要么使用带有总线探测支持的高速缓存硬件。虽然x86硬件具有这样的总线探测，但许多其他系统使用软件来刷新缓存行以防止DMA冲突。

2）某些EHCI控制器上的设备可以处理对/从高内存的DMA。

不幸的是，当前的Linux DMA基础结构没有合理的方式来展示这些功能...而且在任何情况下，HIGHMEM大多是特定于x86\_32的设计瑕疵。因此，最好确保您从不将highmem缓冲区传递到USB驱动程序中。这很容易；它是默认行为。只需不要覆盖它；例如使用NETIF\_F\_HIGHDMA。

这可能会迫使您的调用者进行一些反弹缓冲，从高内存复制到“正常”的DMA存储器。如果您能找到一种解决此问题的好方法（针对具有1 GByte以上内存的x86\_32机器），请随时提交补丁。

### 使用现有缓冲区

在将现有缓冲区映射到设备的DMA地址空间之前，这些现有缓冲区无法用于DMA。但是，大多数传递给驱动程序的缓冲区可以安全地使用这种DMA映射。（请参见动态DMA映射指南的第一部分，标题为“什么内存可以进行DMA？”）。

1）当您使用散点列表时，您可以一次映射所有内容。在某些系统上，这会触发IOMMU，并将散点列表变成单个DMA事务:

int usb\_buffer\_map\_sg (struct usb\_device \*dev, unsigned pipe,

struct scatterlist \*sg, int nents);

void usb\_buffer\_dmasync\_sg (struct usb\_device \*dev, unsigned pipe,

struct scatterlist \*sg, int n\_hw\_ents);

void usb\_buffer\_unmap\_sg (struct usb\_device \*dev, unsigned pipe,

struct scatterlist \*sg, int n\_hw\_ents);

使用新的usb\_sg\_\*()调用可能更容易，它们会进行DMA映射并应用其他调整，以使scatterlist i/o变得快速。

2）某些驱动程序可能更喜欢使用它们正在映射大缓冲区并同步它们的安全重用模型。（如果没有重用，则让usbcore进行映射/取消映射。）巨大的周期性传输在这里是好例子，因为仅同步缓冲区比每次urb完成时对其进行取消映射并在重新提交期间重新映射它更便宜。

当使用这些调用时，所有初始化的urbs都能够正常工作urb->dev，urb->pipe，urb->transfer\_buffer和urb->transfer\_buffer\_length。如果urb是控制请求，那么urb->setup\_packet也必须有效:

struct urb \*usb\_buffer\_map (struct urb \*urb);

void usb\_buffer\_dmasync (struct urb \*urb);

void usb\_buffer\_unmap (struct urb \*urb);

这些调用将为您管理urb->transfer\_dma，并设置URB\_NO\_TRANSFER\_DMA\_MAP，以便usbcore不会映射或取消映射缓冲区。它们不能用于控制请求中的设置数据包缓冲区。

请注意，目前已注释掉了其中一些接口，因为它们没有当前用户。请参阅源代码。除了dmasync调用（其中底层DMA原语已更改）外，如果您想使用它们，大多数调用都可以很容易地取消注释。

## USB请求块 (URB)

修订：2000-12-05

再次修订：2002-06-06

再次修订：2005-09-19

再次修订：2017-03-29

**注**

USB子系统现在在Linux-USB主机端API部分中具有重要内容，这些内容根据当前源代码生成。此特定文档文件不完整，可能没有更新到最后一个版本；除了快速概述外，请不要依赖它。

### 基本概念或“什么是URB？”

新驱动程序的基本思想是消息传递，消息本身称为USB请求块，简称为URB。

1. URB包含执行任何USB事务并传递数据和状态的所有相关信息。
2. 执行URB本质上是异步操作，即usb\_submit\_urb（）调用在成功排队所请求的动作之后立即返回。
3. 可以随时使用usb\_unlink\_urb（）取消一组URB的传输。
4. 每个URB都有一个完成处理程序，该程序在成功完成或取消操作后被调用。 URB还包含用于向完成处理程序传递信息的上下文指针。
5. 每个设备的每个端点逻辑上支持请求队列。您可以填充该队列，以便USB硬件在驱动程序处理另一个完成之时仍然可以将数据传输到端点。这最大化了USB带宽的使用，并在使用周期传输模式时支持对设备的流畅数据流。

### URB结构

struct urb中的一些字段为:

struct urb

{

// (IN) device and pipe specify the endpoint queue

struct usb\_device \*dev; // pointer to associated USB device

unsigned int pipe; // endpoint information

unsigned int transfer\_flags;//URB\_ISO\_ASAP, URB\_SHORT\_NOT\_OK, etc.

// (IN) all urbs need completion routines

void \*context; // context for completion routine

usb\_complete\_t complete; // pointer to completion routine

// (OUT) status after each completion

int status; // 返回ed status

// (IN) buffer used for data transfers

void \*transfer\_buffer; // associated data buffer

u32 transfer\_buffer\_length; // data buffer length

int number\_of\_packets; // size of iso\_frame\_desc

// (OUT) sometimes only part of CTRL/BULK/INTR transfer\_buffer is used

u32 actual\_length; // actual data buffer length

// (IN) setup stage for CTRL (pass a struct usb\_ctrlrequest)

unsigned char \*setup\_packet; // setup packet (control only)

// Only for PERIODIC transfers (ISO, INTERRUPT)

// (IN/OUT) start\_frame is set unless URB\_ISO\_ASAP isn't set

int start\_frame; // start frame

int interval; // polling interval

// ISO only: packets are only "best effort"; each can have errors

int error\_count; // number of errors

struct usb\_iso\_packet\_descriptor iso\_frame\_desc[0];

};

您的驱动程序必须使用所需接口中的适当端点说明符中的值创建“管道”值。

### 如何获得URB？

通过调用usb\_alloc\_urb（）来分配URB

struct urb \*usb\_alloc\_urb(int isoframes, int mem\_flags)

返回是分配的URB的指针，如果分配失败，则返回0。参数isoframes指定要调度的等时传输帧的数量。对于CTRL / BULK / INT，请使用0。mem\_flags参数具有标准的内存分配标志，让您控制（among other things）底层代码是否可以阻止。

要释放URB，请使用usb\_free\_urb（）:

void usb\_free\_urb(struct urb \*urb)

您可以释放已提交但尚未在完成回调中返回给您的URB。当不再使用时，它将自动被释放。

### 什么必须填写？

根据事务类型，定义了一些内联函数，例如usb\_fill\_control\_urb（），usb\_fill\_bulk\_urb（）和usb\_fill\_int\_urb（），可简化初始化。总的来说，它们需要USB设备指针，管道（通常从usb.h中的一般格式），传输缓冲区，所需传输长度，完成处理程序及其上下文。看看一些现有驱动程序，看看它们如何使用。

标志：

1）对于ISO，有两种启动行为指定的start\_frame或ASAP。

2）对于ASAP，将URB\_ISO\_ASAP设置在transfer\_flags中。

如果不应容忍短包，请在transfer\_flags中设置URB\_SHORT\_NOT\_OK。

### 如何提交URB？

只需调用usb\_submit\_urb（）:

int usb\_submit\_urb(struct urb \*urb, int mem\_flags)

mem\_flags参数（例如GFP\_ATOMIC）控制内存分配，例如当内存紧张时低层是否可以阻止。

它立即返回，状态为0（请求排队）或某个错误代码，通常由以下原因引起：

内存不足（-ENOMEM）

设备拔出（-ENODEV）

端点停滞（-EPIPE）

太多排队的ISO传输（-EAGAIN）

请求的ISO帧数太多（-EFBIG）

无效的INT间隔（-EINVAL）

对于iso点，完成处理程序应该使用多缓冲器（re）将URB提交到相同的端点，并使用URB\_ISO\_ASAP标志，以获得无缝的ISO流。

### 如何取消已运行的URB？

有两种方法可以取消您提交但尚未返回到驱动程序的URB。对于异步取消，调用usb\_unlink\_urb（）

int usb\_unlink\_urb(struct urb \*urb)

它从内部列表中删除urb并释放所有分配的HW说明符。状态已更改以反映取消链接。请注意，通常情况下，当usb\_unlink\_urb（）返回时，URB尚未完成；您仍然必须等待调用完成处理程序。

要同步取消URB，请调用usb\_kill\_urb（）

void usb\_kill\_urb(struct urb \*urb)

它执行usb\_unlink\_urb（）所做的所有事情，并且此外它等待URB已返回并且完成处理程序已完成。它还将将URB标记为暂时无法使用，因此如果完成处理程序或任何其他人尝试重新提交它，则会收到-EPERM错误。因此，当usb\_kill\_urb（）返回时，您可以确保URB完全空闲。

需要考虑寿命问题。 URB可以随时完成，完成处理程序可以释放URB。如果在运行usb\_unlink\_urb（）或usb\_kill\_urb（）时发生这种情况，它将导致内存访问违规。驱动程序负责避免这种情况，这通常意味着需要某种锁来防止在URB仍在使用时将其释放。

另一方面，由于usb\_unlink\_urb可能会调用完成处理程序，因此当调用usb\_unlink\_urb时，处理程序不得获取任何保持的锁。解决此问题的通用解决方案是在保持锁定的同时递增URB的引用计数，然后放弃锁定并调用usb\_unlink\_urb或usb\_kill\_urb，然后递减URB的引用计数。您通过调用:c:func`usb\_get\_urb`来递增引用计数:

struct urb \*usb\_get\_urb（struct urb \*urb）

（忽略返回；它与参数相同），通过调用usb\_free\_urb（）来递减引用计数。当然，如果完成处理程序不会释放URB，则所有这些都是不必要的。

### 完成处理程序怎么样？

处理程序具有以下类型

typedef void (\*usb\_complete\_t)(struct urb \*)

即它获得导致完成调用的URB。在完成处理程序中，您应该查看urb->status以检测任何USB错误。由于上下文参数包含在URB中，因此可以将信息传递给完成处理程序。

请注意，即使报告了错误（或取消链接），数据也可能已传输。这是因为USB传输被分包处理；在调用完成之前，可能需要传输十六个数据包才能传输您的1KByte缓冲区中的十个成功传输。

**警告**

永远不要在完成处理程序中休眠。

这些经常在原子上下文中调用。

在当前内核中，完成处理程序在本地中断禁用的情况下运行，但在未来，这将被更改，因此不要假设本地IRQ始终在完成处理程序内部禁用。

### 如何进行等时（ISO）传输？

除了存在于批量传输中的字段外，对于ISO，您还必须将urb->interval设置为何时进行传输的时间间隔；通常是每帧一次（对于高速设备，这是每个微帧一次）。实际使用的间隔将是您指定的不大于的2的幂。您可以使用usb\_fill\_int\_urb（）宏来填充大多数ISO传输字段。

对于ISO传输，您还必须填写单个usb\_iso\_packet\_descriptor结构，该结构由usb\_alloc\_urb（）在URB的末尾分配，用于要安排的每个数据包。

usb\_submit\_urb（）调用会修改urb->interval以实现间隔值，该值小于或等于请求的间隔值。如果使用URB\_ISO\_ASAP调度，则还会更新urb->start\_frame。

对于每个条目，必须指定此帧的数据偏移量（基于transfer\_buffer），并指定要写入/期望读取的长度。完成后，actual\_length包含实际传输长度，status包含此帧的ISO传输的结果状态。允许从帧到帧指定可变长度（例如用于音频同步/自适应传输速率）。您还可以使用长度0来省略一个或多个帧（条纹）。

为了调度，您可以选择自己的起始帧或URB\_ISO\_ASAP。如前所述，如果您始终保留至少一个URB排队，且完成保持（重新）提交稍后的URB，则会获得平滑的ISO流（如果USB带宽利用允许）。

如果指定自己的起始帧，请确保它比当前帧提前几个帧。如果您正在将ISO数据与某些其他事件流同步，则可能需要此模式。

### 如何启动中断（INT）传输？

中断传输与等时传输一样是周期性的，并以2的幂（1、2、4等单位）为间隔发生。完整和低速设备的单位是帧，高速设备的单位是微帧。您可以使用usb\_fill\_int\_urb（）宏填充INT传输字段。

usb\_submit\_urb（）调用修改urb->interval为已实现的间隔值，该值小于或等于请求的间隔值。

在Linux 2.6中，与早期版本不同，中断URB在完成后不会自动重新启动。它们在调用完成处理程序时结束，就像其他URB一样。如果您希望中断URB重新启动，则必须通过您的完成处理程序重新提交它。

## USB电源管理

### 什么是电源管理？

电源管理（PM）是一种通过暂停计算机系统的某些部分来节省能源的实践。当组件处于暂停状态时，它处于非功能性低功率状态；它甚至可能完全关闭。当内核需要使用它时，可以恢复暂停的组件（返回到功能全功率状态）。（还有一些PM形式，其中组件被放置在较不功能但仍可用的状态而不是被暂停；例如减少CPU的时钟速率。本文档不讨论那些其他形式。)

当被暂停的部件包括CPU和大部分其余系统时，我们将其称为“系统暂停”。当在整个系统仍在运行的情况下关闭特定设备时，我们称其为“动态暂停”（也称为“运行时暂停”或“选择性暂停”）。本文主要集中讨论USB子系统中如何实现动态PM，尽管对系统PM也有一定涵盖（有关更多有关系统PM的信息，请参见Documentation/power/\*.rst）。

只有内核启用了CONFIG\_SUSPEND或CONFIG\_HIBERNATION选项才支持系统PM。内核启用CONFIG\_PM选项时，USB的动态PM支持是存在的。

[历史上，只有内核已构建CONFIG\_USB\_SUSPEND选项（该选项取决于CONFIG\_PM\_RUNTIME）才支持USB的动态PM。从3.10内核版本开始，内核启用CONFIG\_PM\_RUNTIME选项时，USB的动态PM支持是存在的。CONFIG\_USB\_SUSPEND选项已被删除。]

### 什么是远程唤醒？

当设备被暂停时，它通常不会恢复，直到计算机要求它恢复。同样，如果整个计算机被暂停，通常不会恢复，直到用户告诉它，例如通过按电源按钮或打开盖子。

但是，有些设备具有自行恢复、请求内核恢复它们甚至告诉整个计算机恢复的功能。这种功能称为“远程唤醒”，有多个名称，例如“Wake On LAN”；我们将其通用地称为“远程唤醒”。当设备启用了远程唤醒并且被暂停时，它可能会对某些外部事件作出恢复恢复（或发送恢复请求）。例如，键盘被暂停时按下一个键就会恢复，或者USB集线器被暂停时插入设备就会恢复。

### 何时算作USB设备处于空闲状态？

当内核认为USB设备没有忙于执行任何重要任务并因此将其暂停时，设备处于空闲状态。确切的定义取决于设备的驱动程序。即使没有实际通信发生，驱动程序也可以声明设备不是空闲状态。 （例如，除非该集线器中所有已插入该集线器的设备都已暂停，否则不认为集线器处于空闲状态。）此外，只要程序保持其usbfs文件打开，无论是否会进行任何I/O操作，设备都不会被视为空闲。如果USB设备没有驱动程序，其usbfs文件没有打开，并且没有通过sysfs进行访问，则它肯定处于空闲状态。

### 动态PM的形式

动态暂停是指内核决定暂停空闲设备的情况，简称为自动暂停。通常情况下，设备除非空闲一段最小时间（即所谓的空闲延迟时间），否则不会被自动暂停。

当然，内核自主操作不应妨碍计算机或其设备的正常工作。如果设备已经被自动暂停，并且程序尝试使用它，内核将自动恢复该设备（自动恢复）。出于同样的原因，如果设备支持远程唤醒，自动暂停的设备通常会启用远程唤醒。

值得一提的是，许多USB驱动程序不支持自动暂停。事实上，在此篇文章撰写时（Linux 2.6.23），唯一支持它的驱动程序是hub驱动程序、kaweth、asix、usblp、usblcd和usb-skeleton（不计算在内）。如果绑定到设备的是不支持的驱动程序，则设备不会被自动暂停。实际上，内核会假装该设备从未处于空闲状态。

我们可以将电源管理事件分为两大类外部和内部。外部事件是由USB堆栈之外的某个代理触发的系统挂起/恢复（由用户空间触发）、手动动态恢复（也由用户空间触发）和远程唤醒（由设备触发）。内部事件是由USB堆栈内部触发的自动暂停和自动恢复。请注意，所有动态暂停事件都是内部事件；外部代理不允许发出动态暂停。

### 动态PM的用户界面

控制动态PM的用户界面位于每个USB设备的sysfs目录的power/子目录中，即/sys/bus/usb/devices/.../power/，其中“…”是设备的ID。相关属性文件包括wakeup、control和autosuspend\_delay\_ms。（可能还有一个名为level的文件；这个文件已经在2.6.35内核中被废弃，并被control文件所取代。在2.6.38中，autosuspend文件将被废弃，并被autosuspend\_delay\_ms文件所取代。唯一的区别是新文件用毫秒表示延迟，而旧文件使用秒。令人困惑的是，两个文件都存在于2.6.37中，但只有autosuspend有效。）

power/wakeup

如果设备不支持远程唤醒，则此文件为空。否则，文件包含单词“enabled”或“disabled”，您可以将这些单词写入文件中。设置确定设备在下次挂起时是否启用远程唤醒。（如果在设备挂起时更改了设置，则更改将在下次挂起之前不会生效。）

power/control

此文件包含两个单词之一on或auto。您可以将这些单词写入文件中以更改设备的设置。

1）on意味着应恢复设备，并且不允许自动暂停。（当然，仍可以允许系统挂起。）

2）auto是内核允许自动暂停和自动恢复设备的正常状态。（在2.6.32之前的内核版本中，您还可以指定suspend，这意味着设备应保持挂起状态，而自动恢复是不允许的。这个设置已不再被支持。）

power/autosuspend\_delay\_ms

此文件包含一个整数值，表示设备在内核自动暂停它之前应保持空闲的毫秒数（空闲延迟时间）。默认值为2000。0表示一旦设备进入空闲状态就自动暂停，负数表示永远不会自动暂停。您可以将数字写入文件中以更改自动暂停的空闲延迟时间。

将-1写入power/autosuspend\_delay\_ms并在power/control中写入on实际上是相同的操作——它们都防止设备被自动暂停。是的，这在API中是冗余的。

（在2.6.21中，将0写入power/autosuspend将防止设备被自动暂停；这种行为在2.6.22中被更改。在2.6.21之前不存在power/autosuspend属性，2.6.22之前不存在power/level属性，2.6.34中添加了power/control属性，2.6.37中添加了power/autosuspend\_delay\_ms属性，但直到2.6.38才变得实用。）

### 更改默认的空闲延迟时间

usbcore模块参数控制默认的自动暂停空闲延迟时间（以秒为单位）。您可以在加载usbcore时指定该值。例如，将其设置为5秒而不是2秒为：

modprobe usbcore autosuspend=5

同样，您可以在/etc/modprobe.d中的配置文件中添加一行，说

options usbcore autosuspend=5

一些发行版在启动过程中非常早地加载usbcore模块，通过从initramfs映像运行的程序或脚本来完成。要更改参数值，必须重新构建该映像。

如果usbcore被编译到内核中而不是作为可加载模块构建，则可以将以下内容添加到内核的引导命令行中：

usbcore.autosuspend=5

最后，可以在系统运行时更改参数值。如果执行：

echo 5 >/sys/module/usbcore/parameters/autosuspend

然后每个新的USB设备的autosuspend空闲延迟都将初始化为5。 （已经存在设备的空闲延迟值不会受影响。）

将初始默认空闲延迟设置为-1将防止任何USB设备的自动挂起。这具有使您能够仅对选择的设备启用自动挂起的好处。

### 警告

USB规范规定，所有USB设备都必须支持电源管理。尽管如此，悲惨的事实是许多设备支持得不是很好。您可以将它们挂起，但是当尝试恢复它们时，它们会断开与USB总线的连接或完全停止工作。虽然打印机和扫描仪似乎特别容易出现此问题，但许多其他类型的设备也存在同样的缺陷。

因此，内核默认情况下禁用除集线器以外的所有设备的自动挂起（power / control属性初始化为on）。至少，集线器在这方面似乎表现得比较好。

（在2.6.21和2.6.22中不是这种情况。几乎所有USB设备默认启用了自动挂起。因此，许多人经历了问题。）

这意味着，除非用户或程序明确启用它，否则不会自动挂起非集线器设备。就我所知，目前没有广泛使用的程序可以做到这一点，我们希望在不久的将来，设备管理器（如HAL）将承担这项额外的责任。同时，您始终可以手动执行必要的操作或将它们添加到udev脚本中。您还可以更改空闲延迟时间。 2秒并不是每个设备的最佳选择。

如果驱动程序知道其设备具有适当的挂起/恢复支持，它可以自行启用自动挂起。例如，笔记本电脑的网络摄像头视频驱动程序可能会这样做（在最近的内核中，他们确实这样做了），因为这些设备很少使用，因此通常应该自动挂起。

有时候，即使设备使用自动挂起表现良好，仍会出现问题。例如，管理键盘和鼠标的usbhid驱动程序具有自动挂起支持。使用多个键盘进行测试，打字会导致键盘进行远程唤醒，但仍然经常会导致丢失的按键。使用鼠标的测试表明，其中一些将对按钮按下发出远程唤醒请求，但不会对运动产生反应，而另一些则对两者都不会做出反应。

内核不会阻止您在无法处理它的设备上启用自动挂起。理论上，甚至可能在错误的时间挂起设备而损坏设备。 （高度不可能，但有可能。）小心。

### 电源管理的驱动程序界面

USB驱动程序支持外部电源管理的要求非常简单，驱动程序仅需在其usb\_driver结构中定义:

.suspend

.resume

.reset\_resume

方法，并且reset\_resume方法是可选的。方法的工作非常简单：

1）挂起方法被调用以警告驱动程序设备将被挂起。如果驱动程序返回负错误代码，则挂起将被中止。通常，驱动程序将返回0，在这种情况下，它必须取消所有未完成的URB（usb\_kill\_urb（）），并且不需要提交任何更多的URB。

2）恢复方法被调用以告诉驱动程序设备已经恢复，并且驱动程序可以返回到正常操作。可以再次提交URB。

3）reset\_resume方法被调用以告诉驱动程序设备已经恢复，并且它也已被重置。由于设备可能已经失去了大部分或全部状态（虽然接口将与挂起之前的相同），因此驱动程序应重新进行任何必要的设备初始化。

如果设备在挂起时断开连接或关闭电源，则将调用disconnect方法而不是resume或reset\_resume方法。当从休眠中唤醒时，这也很可能发生，因为许多系统在休眠期间不保持USB主机控制器的挂起电流。（可以通过使用USB Persist工具解决休眠强制断开问题。）

reset\_resume方法由USB Persist工具（参见系统挂起期间的USB设备持久化）使用，并且在未启用CONFIG\_USB\_PERSIST的某些情况下也可以使用。目前，在恢复期间重置设备并且驱动程序没有reset\_resume方法时，驱动程序不会收到有关恢复的任何通知。较新的内核将调用驱动程序的断开方法；2.6.23不会这样做。

USB驱动程序绑定到接口，因此当接口挂起或恢复时，它们的挂起和恢复方法会被调用。原则上，人们可能希望挂起设备上的某些接口（即，强制为那些接口停止所有活动的驱动程序），而不挂起其他接口。USB核心不允许这样做；当设备本身挂起时，所有接口都被挂起，当设备恢复时，所有接口都被恢复。无法挂起或恢复设备的全部而不是部分接口。您唯一能够接近这一点是取消接口的绑定驱动程序。

### 自动挂起和自动恢复的驱动程序接口

为了支持自动挂起和自动恢复，驱动程序应实现上述三种方法。此外，驱动程序通过在其usb\_driver结构中设置.supports\_autosuspend标志来指示其支持自动挂起。因此，一旦驱动程序的任一接口变得繁忙或空闲，它就有责任通知USB核心。驱动程序通过调用以下六个函数来实现:

int usb\_autopm\_get\_interface(struct usb\_interface \*intf);

void usb\_autopm\_put\_interface(struct usb\_interface \*intf);

int usb\_autopm\_get\_interface\_async(struct usb\_interface \*intf);

void usb\_autopm\_put\_interface\_async(struct usb\_interface \*intf);

void usb\_autopm\_get\_interface\_no\_resume(struct usb\_interface \*intf);

void usb\_autopm\_put\_interface\_no\_suspend(struct usb\_interface \*intf);

这些函数通过在usb\_interface的嵌入式设备结构中维护使用计数器来工作。当计数器>0时，接口被视为繁忙，内核将不会自动挂起接口的设备。当使用计数器=0时，接口被视为空闲，内核可能会自动挂起设备。

驱动程序必须小心平衡其对使用计数器的总体更改。当驱动程序从其接口取消绑定时，“get”保持不平衡将继续生效，如果接口再次绑定到驱动程序，则会阻止设备进入运行时挂起。另一方面，只要驱动程序保留接口的活动引用（通过usb\_get\_intf和usb\_put\_intf），即使从其断开例程返回，也允许使用usb\_autopm\_\*函数来实现此平衡 - 例如从工作队列例程中。

使用异步例程的驱动程序负责其自身的同步和互斥。

usb\_autopm\_get\_interface()增加使用计数器并自动恢复设备（如果设备已挂起）。如果自动恢复失败，则计数器将被递减。

usb\_autopm\_put\_interface()递减使用计数器并在新值=0时尝试自动挂起。

usb\_autopm\_get\_interface\_async()和usb\_autopm\_put\_interface\_async()的操作与非异步对应物差不多。最大的区别在于它们使用工作队列执行作业的恢复或挂起部分。因此，它们可以在原子上下文中调用，例如URB的完成处理程序，但它们返回时设备通常还没有达到所需的状态。

usb\_autopm\_get\_interface\_no\_resume()和usb\_autopm\_put\_interface\_no\_suspend()仅递增或递减使用计数器；它们不尝试执行自动恢复或自动挂起。因此，它们可以在原子上下文中调用。

最简单的使用模式是驱动程序在其打开例程中调用usb\_autopm\_get\_interface()并在其关闭或释放例程中调用usb\_autopm\_put\_interface()。但是也可以使用其他模式。

上述自动挂起尝试通常会因某种原因而失败。例如，可能将power/control属性设为on，或者同一设备中的另一个接口可能不空闲。这是完全正常的。如果失败的原因是设备还没有空闲足够长的时间，定时器将被调度自动执行操作，以便在auto挂起空闲延迟到期时执行。

自动恢复尝试也可能会失败，尽管失败意味着设备不再存在或不正常运行。与自动挂起不同，自动恢复没有空闲延迟。

### 驱动程序接口的其他部分

如果驱动程序知道设备可以正确地挂起和恢复，则可以在其探测()例程中调用:

usb\_enable\_autosuspend(struct usb\_device \*udev);

以启用其设备的自动挂起。这等效于将“auto”写入设备的power/control属性。同样，驱动程序可以通过调用来禁用自动挂起:

usb\_disable\_autosuspend(struct usb\_device \*udev);

这与将“on”写入power/control属性完全相同。

有时驱动程序需要确保在自动暂停期间启用远程唤醒。例如，如果用户无法通过键盘上的键入触发远程唤醒，则自动暂停键盘没有太大意义。如果驱动程序将intf->needs\_remote\_wakeup设置为1，则如果远程唤醒不可用，内核将不会自动暂停设备。（但是，如果设备已经自动暂停，则设置此标志不会导致内核自动恢复它。通常，驱动程序会在其探测方法中设置此标志，在此时设备保证不会自动暂停。）

如果驱动程序在中断上下文中异步进行其I/O，则应在开始输出之前调用usb\_autopm\_get\_interface\_async()，并在输出队列排空时调用usb\_autopm\_put\_interface\_async()。当它接收到输入事件时，应调用

usb\_mark\_last\_busy(struct usb\_device \*udev);

在事件处理程序中。这告诉PM核心该设备刚刚繁忙，因此下一个自动暂停空闲延迟到期应被推迟。许多usb\_autopm\_\*例程也会进行此调用，因此仅当中断驱动输入到达时驱动程序才需要考虑这一点。

异步操作始终会产生竞争。例如，当核心刚刚决定设备已经足够长时间处于空闲状态但尚未调用驱动程序的挂起方法时，驱动程序可能会在此时调用usb\_autopm\_get\_interface\_async()例程。挂起方法必须负责与I/O请求例程和URB完成处理程序同步；如果驱动程序需要使用设备，则它应使自动挂起失败并带有-EBUSY。

永远不应允许外部挂起调用以这种方式失败，仅应自动暂停调用。驱动程序可以通过将PMSG\_IS\_AUTO()宏应用于挂起方法的消息参数来区分它们；它将为内部PM事件（自动暂停）返回True，并为外部PM事件返回False。

### 相互排斥

对于外部事件而言-但不一定是对于自动暂停或自动恢复-设备信号量（udev->dev.sem）将在调用挂起或恢复方法时保持。这意味着外部挂起/恢复事件与探测、断开连接、pre\_reset和post\_reset的调用相互排斥；USB核心也保证这对于自动暂停/自动恢复事件同样成立。

如果驱动程序想要在某个关键部分阻止所有挂起/恢复调用，则最好的方式是锁定设备并调用usb\_autopm\_get\_interface()（并在关键部分结束时执行相反操作）。持有设备信号量将阻止所有外部PM调用，而usb\_autopm\_get\_interface()将防止任何内部PM调用，即使失败。（练习为什么？）

### 动态PM和系统PM之间的交互

动态电源管理和系统电源管理可以以几种方式互动。

首先，系统挂起发生时，设备可能已经自动暂停。由于系统挂起应尽可能透明，因此设备应在系统恢复后保持暂停状态。但是，在实践中，这种理论可能并不奏效；随着时间的推移，内核在这方面的行为发生了变化。从2.6.37开始，策略是在系统恢复期间恢复所有设备，并让它们处理自己的运行时挂起。

其次，在进行系统暂停时，可能会发生动态电源管理事件。这个窗口很短，因为系统挂起不需要太长时间（通常几秒钟），但是它可能会发生。例如，当系统正在暂停时，暂停的设备可能会发送远程唤醒信号。远程唤醒可能会成功，这将导致系统挂起中止。如果远程唤醒不成功，它仍可能保持活动状态，从而在系统挂起完成后导致系统恢复。或者远程唤醒可能会失败并且丢失。发生的结果取决于时间以及硬件和固件设计。

### xHCI硬件链路PM

xHCI主机控制器为支持链路PM的usb2.0（xHCI 1.0功能）和usb3.0设备提供硬件链路功率管理。通过启用硬件LPM，主机可以自动将设备置于较低的电源状态（对于usb2.0设备为L1，对于usb3.0设备为U1 / U2），设备可以非常快速地进入和恢复。

控制硬件LPM的用户界面位于每个USB设备的sysfs目录下的power /子目录中，即在/ sys / bus / usb / devices /… / power /中，“...”是设备的ID。相关的属性文件是usb2\_hardware\_lpm和usb3\_hardware\_lpm。

power / usb2\_hardware\_lpm

当一个支持LPM的USB2设备插入到支持软件LPM的xHCI主机根集线器中时，主机将对其运行软件LPM测试；如果设备进入L1状态并成功恢复，并且主机支持USB2硬件LPM，则该文件将显示出来，并且驱动程序将启用设备上的硬件LPM。您可以将y / Y / 1或n / N / 0写入文件，以手动启用/禁用USB2硬件LPM。这主要是为了测试目的。

power/usb3\_hardware\_lpm\_u1 power/usb3\_hardware\_lpm\_u2

当将支持链接电源管理的xHCI主机插入支持USB 3.0 lpm的设备时，它会检查BOS说明符中是否已设置U1和U2退出延迟；如果检查通过并且主机支持USB3硬件LPM，则将为该设备启用USB3硬件LPM，并创建这些文件。文件保存一个字符串值（启用或禁用），表示是否为该设备启用了USB3硬件LPM U1或U2。

### USB端口电源控制

除了暂停端点设备和启用硬件控制的链接电源管理外，USB子系统还具有在某些条件下禁用端口电源的能力。通过向集线器发送Set/ClearPortFeature(PORT\_POWER)请求来控制电源。在根或平台内部集线器的情况下，主机控制器驱动程序将PORT\_POWER请求转换为平台固件（ACPI）方法调用以设置端口电源状态。有关更多背景，请参见2012 Linux Plumbers Conference幻灯片1和视频2

在接收到ClearPortFeature(PORT\_POWER)请求后，USB端口逻辑关闭，并可能触发对端口3电源（VBUS）的实际丧失。在一组多个端口汇集为共享电源井的情况下，可以保持VBUS，从而使电源保持到关闭组中的所有端口。集线器端口也可以配置为充电应用程序而保持VBUS。无论如何，逻辑关闭的端口将失去与其设备的连接，不会响应热插拔事件，也不会响应远程唤醒事件。

**警告**

关闭端口可能会导致无法热插入设备。有关详情，请参见“端口电源控制的用户界面”。

就设备本身的影响而言，它类似于设备在系统挂起期间经历的情况，即电源会失去连接。任何与系统挂起一起表现不良的USB设备或驱动程序都将受到端口电源循环事件的影响。因此，为此原因，该实现与集线器的系统恢复路径共享相同的设备恢复路径（并遵守相同的怪癖）。

1. <http://dl.dropbox.com/u/96820575/sarah-sharp-lpt-port-power-off2-mini.pdf>
2. <http://linuxplumbers.ubicast.tv/videos/usb-port-power-off-kerneluserspace-api/>
3. USB 3.1 Section 10.12

唤醒提示如果设备配置为发送唤醒事件，则端口电源控制实现将阻止该端口的关机尝试。

### 端口电源控制的用户界面

端口电源控制机制使用PM运行时系统。通过清除端口设备的power/pm\_qos\_no\_power\_off标志来请求关机（默认为1）。如果端口已断开连接，则将立即接收到ClearPortFeature(PORT\_POWER)请求。否则，它将遵守PM运行时规则，并要求附加的子设备和所有后代设备都处于挂起状态。该机制依赖于集线器在其集线器说明符（wHubCharacteristics逻辑电源切换模式字段）中发布端口电源切换。

请注意，一些接口设备/驱动程序不支持自动挂起。用户空间可能需要解除接口驱动程序与usb\_device挂起。解除绑定的接口设备默认处于挂起状态。在取消绑定时，请注意解除接口驱动程序，并不是父usb设备的驱动程序。另外，不要解除集线器接口驱动程序的绑定。如果取消了usb设备（而不是接口）的驱动程序绑定，内核将无法恢复该设备。如果解除了集线器接口驱动程序的绑定，将失去对其子端口的控制，并且所有附加的子设备都将断开连接。一个好的经验法则是，如果一个设备的“driver/module”链接指向/sys/module/usbcore，则解除绑定它将干扰端口电源控制。

端口电源控制的相关文件示例。请注意，在此示例中，这些文件与usb hub设备（前缀）相关:

prefix=/sys/devices/pci0000:00/0000:00:14.0/usb3/3-1

attached child device +

hub port device + |

hub interface device + | |

v v v

$prefix/3-1:1.0/3-1-port1/device

$prefix/3-1:1.0/3-1-port1/power/pm\_qos\_no\_power\_off

$prefix/3-1:1.0/3-1-port1/device/power/control

$prefix/3-1:1.0/3-1-port1/device/3-1.1:<intf0>/driver/unbind

$prefix/3-1:1.0/3-1-port1/device/3-1.1:<intf1>/driver/unbind

...

$prefix/3-1:1.0/3-1-port1/device/3-1.1:<intfN>/driver/unbind

除了这些文件外，一些端口可能还具有与其他集线器上的端口相对应的“对等”链接的可能性。预期是所有超速端口都具有高速对等体:

$prefix/3-1:1.0/3-1-port1/peer -> ../../../../usb2/2-1/2-1:1.0/2-1-port1

../../../../usb2/2-1/2-1:1.0/2-1-port1/peer -> ../../../../usb3/3-1/3-1:1.0/3-1-port1

与‘伴随端口’或‘ehci/xhci共享转换端口’不同，同级端口只是将高速和超高速接口引脚合并成一个USB3连接器。同级端口共享同一祖先XHCI设备。

当超高速端口关闭时，设备可以降级连接尝试连接至高速接口。实施措施可以防止这种情况的发生：

1）端口挂起按顺序执行，确保在允许同级超高速端口关闭之前高速端口已被关闭。含义是将pm\_qos\_no\_power\_off设置为0的超高速端口可能不会关机，直到其高速同级进入其运行时挂起状态。如果希望保证超高速端口关机，用户空间必须注意对挂起的顺序进行排序。

2）端口恢复按顺序执行，强制超高速端口在其高速同级之前启动。

3）口恢复始终触发连接的子设备进行恢复。在失去电源后，设备可能已被移除或需要重置。当父级端口恢复电源时恢复子设备可解决这些状态并将最大端口电源周期频率锁定在子设备可以挂起（自动挂起延迟）和恢复（重置恢复延迟）的速率上。

用于端口电源控制的Sysfs文件:

<hubdev-portX>/power/pm\_qos\_no\_power\_off:

此可写标志控制闲置端口的状态。一旦所有子孙节点都已挂起，端口便可挂起/关机，前提是pm\_qos\_no\_power\_off为‘0’。如果pm\_qos\_no\_power\_off为‘1’，输出将保持活动/供电，而不考虑后代的状态。默认值为1。

<hubdev-portX>/power/runtime\_status:

此文件反映端口是否‘活动’（开启电源）或‘挂起’（逻辑关闭）。用户空间没有指示VBUS是否仍然供应的任何迹象。

<hubdev-portX>/connect\_type:

此为用户空间提供的建议只读标志，表示端口的位置和连接类型。它返回四个值之一‘热插拔’，‘固定电缆’，‘未使用’和‘未知’。除‘未知’外，所有值都由平台固件设置。

‘热插拔’表示平台上可外部连接/可见端口。通常，用户空间会选择保持这种端口接通，以处理新设备连接事件。

‘固定电缆’指的是不可见，但可连接的端口。例如，用于蓝牙的USB内部端口可以通过外部开关断开连接，或具有带连接的USB相机的端口。如果pm\_qos\_no\_power\_off与控制连接的任何开关协调，预期安全的是允许这些端口进入挂起状态。在关闭端口电源之前，用户空间必须安排好连接设备，或在启用切换连接之前激活端口。

‘未使用’指的是预计永远不会连接设备的内部端口。这些可能是空的内部端口，或在平台上未物理暴露的端口。始终关闭电源被认为是安全的。

‘未知’意味着平台固件不提供此端口的信息。最常见的是外部集线器端口，应视为‘热插拔’以进行策略决策。

**注意**

1）由于我们依赖BIOS正确获取此ACPI信息，USB端口说明可能丢失或错误。

2）慎重清除pm\_qos\_no\_power\_off的值。一旦断电，此端口将不会响应新的连接事件。

附加子设备后，在允许端口关闭电源之前将应用其他约束。

<child>/power/control:

必须为auto，并且端口不会关机，直到<child>/power/runtime\_status反映‘已挂起’状态。默认值由子设备驱动程序控制。

<child>/power/persist:

这适用于大多数设备，默认值为1，表示内核是否可以在断电/端口电源事件（挂起）的情况下保持设备的配置。当此值为0（反常设备）时，禁用端口关机。

<child>/driver/unbind:

具有唤醒能力的设备将阻止端口关闭电源。此时，清除接口设备的usb-internal唤醒能力的唯一机制是取消绑定其驱动程序。

相对于端口设备的关机前提设置概述:

echo 0 > power/pm\_qos\_no\_power\_off

echo 0 > peer/power/pm\_qos\_no\_power\_off # if it exists

echo auto > power/control # this is the default value

echo auto > <child>/power/control

echo 1 > <child>/power/persist # this is the default value

### 建议用户空间端口电源策略

如上所述，用户空间需要注意并谨慎选择哪些端口启用断电功能。

默认配置是所有端口都以power/pm\_qos\_no\_power\_off设置为1，导致端口始终保持活动状态。

在对端口的说明（ACPI \_PLD 记录用于填充“connect\_type”）的平台固件的说明具有信心的情况下，用户空间可以清除所有“未使用”的端口的 pm\_qos\_no\_power\_off。对于“硬连接”端口，只要与端口的任何连接切换协调好了，也可以采取同样的方法。

更积极的用户空间策略是在某些外部因素表明用户已停止与系统交互时启用所有端口的USB端口断电（将<hubdev-portX>/power/pm\_qos\_no\_power\_off设置为0）。例如，发行版可能希望在屏幕空白时打开所有USB端口的电源，并在屏幕重新变为活动状态后重新通电。智能手机和平板电脑可能希望在用户按下电源按钮时关闭USB端口电源。

## USB热插拔

### Linux热插拔

在可热插拔的总线上，如USB（和Cardbus PCI），最终用户在通电状态下将设备插入总线中。大多数情况下，用户希望立即使用这些设备。这意味着系统必须进行许多操作，包括：

1. 查找可以处理设备的驱动程序。这可能涉及加载内核模块；新驱动程序可以使用模块-init-tools向用户实用程序发布其设备（和类）支持。
2. 将驱动程序绑定到该设备。总线框架使用设备驱动程序的probe()例程将其绑定。
3. 告诉其他子系统配置新设备。可能需要启用打印队列、启动网络、挂载磁盘分区等。在某些情况下，这些是特定于驱动程序的操作。

这涉及到内核模式和用户模式操作的混合。让设备立即可用意味着任何用户模式操作都不能等待管理员完成它们内核必须触发它们，无论是被动地（触发某些监视守护程序来调用帮助程序），还是主动地（直接调用这样的用户模式帮助程序）。

那些触发的操作必须支持系统的管理政策；这些程序在这里被称为“策略代理”。通常它们涉及调度到更熟悉的管理工具的shell脚本。

因为其中一些操作依赖于关于驱动程序（元数据）的信息，这些信息目前仅在动态链接驱动程序时可用，因此当您配置高度模块化的系统时，您可以得到最佳的热插拔。

### 内核热插拔助手（/sbin/hotplug）

有一个内核参数：/proc/sys/kernel/hotplug，通常包含路径名/sbin/hotplug。该参数命名了一个程序，内核可以在各种时候调用它。

/sbin/hotplug程序可以由任何子系统调用，作为其对配置更改的反应之一，从该子系统的线程中调用。只需要一个参数通知某个内核事件的子系统的名称。该名称用作进一步事件分派的第一个关键字；由发起该调用的子系统指定任何其他参数和环境参数。

热插拔软件和其他资源可在以下网址找到：

http://linux-hotplug.sourceforge.net

该网站还提供邮件列表信息。

### USB策略代理

USB子系统当前在添加或删除USB设备时调用/sbin/hotplug。内核集线器工作队列[hub\_wq]或者作为根集线器初始化的一部分（由init、modprobe、kapmd等完成）执行调用。它的单个命令行参数是字符串“usb”，并传递了以下环境变量

|  |  |
| --- | --- |
| ACTION | add, remove |
| PRODUCT | USB vendor, product, and version codes (hex) |
| TYPE | device class codes (decimal) |
| INTERFACE | interface 0 class codes (decimal) |

如果配置了“usbdevfs”，则还会传递DEVICE和DEVFS。DEVICE是设备的路径名，对于具有多个和/或备用接口的设备，这很有用，使驱动程序的选择变得复杂。按设计，USB热插拔独立于usbdevfs您可以在不使用该文件系统的情况下完成USB设备设置的大多数基本部分，而无需运行用于检测系统配置更改的用户模式守护程序。

当前可用的策略代理实现可以为模块加载驱动程序，并调用特定于驱动程序的设置脚本。最新的代理利用了USB模块 - init-tools支持。稍后的代理可能会卸载驱动程序。

### USB模块支持

module-init-tools的当前版本将创建一个modules.usbmap文件，其中包含来自每个驱动程序的MODULE\_DEVICE\_TABLE条目。各种用户模式策略代理可以使用这些文件，以确保所有正确的驱动程序模块在引导时或稍后加载。

有关此类表项的完整信息，请参见linux/usb.h；或查看现有驱动程序。每个表项说明了用于将驱动程序与设备或设备类匹配时要使用的一个或多个条件。特定的条件通过在“匹配标志”中设置的位与字段值配对来识别。您可以直接构造这些条件，也可以使用诸如此类的宏，并使用driver\_info存储更多信息：

USB\_DEVICE（vendorId，productId）

......匹配指定厂商和产品ID的设备

USB\_DEVICE\_VER（vendorId，productId，lo，hi）

......与USB\_DEVICE相似，但lo <= productversion <= hi

USB\_INTERFACE\_INFO（class，subclass，protocol）

......匹配指定的接口类信息

USB\_DEVICE\_INFO（class，subclass，protocol）

......匹配指定的设备类信息

例如，支持几种特定USB设备和它们的怪癖的驱动程序，可能有这样的:

static const struct usb\_device\_id mydriver\_id\_table[] = {

{ USB\_DEVICE (0x9999, 0xaaaa), driver\_info: QUIRK\_X },

{ USB\_DEVICE (0xbbbb, 0x8888), driver\_info: QUIRK\_Y|QUIRK\_Z },

...

{ } /\* end with an all-zeroes entry \*/

};

MODULE\_DEVICE\_TABLE(usb, mydriver\_id\_table);

大多数USB设备驱动程序都应将这些表传递给USB子系统以及模块管理子系统。但并非所有驱动程序框架都使用在USB上分层的接口进行连接，因此它们不需要此类struct usb\_driver。

直接连接到USB子系统的驱动程序应声明如下:

static struct usb\_driver mydriver = {

.name = "mydriver",

.id\_table = mydriver\_id\_table,

.probe = my\_probe,

.disconnect = my\_disconnect,

/\*

if using the usb chardev framework:

.minor = MY\_USB\_MINOR\_START,

.fops = my\_file\_ops,

if exposing any operations through usbdevfs:

.ioctl = my\_ioctl,

\*/

};

当USB子系统了解驱动程序的设备ID表时，它在选择要probe（）的驱动程序时使用该表。执行新设备处理的线程对MODULE\_DEVICE\_TABLE中的驱动程序设备ID条目与设备的接口和设备说明符进行匹配。仅当存在匹配项时，才会调用probe（），并且probe（）的第三个参数将是匹配的条目。

如果未为驱动程序提供id\_table，则可能会为每个新设备探测驱动程序； probe（）的第三个参数将为NULL。

## 系统暂停期间的USB设备持久性

### 问题是什么？

根据USB规范，当USB总线挂起时，总线必须继续提供挂起电流（约1-5 mA）。这是为了设备可以维护其内部状态，以及集线器可以检测连接更改事件（设备被插入或拔出）。技术术语是“电源会话”。

如果USB设备的电源会话中断，则系统需要像设备已被拔出一样运行。这是一种保守的方法；在缺乏挂起电流的情况下，计算机无法知道实际发生了什么。也许仍连接相同的设备，或者它被拔出并插入了不同的设备。系统必须做最坏的情况。

默认情况下，Linux按照规范进行操作。如果USB主机控制器在系统挂起期间失去电源，则当系统唤醒时，附加到该控制器的所有设备都被视为已断开连接。这始终是安全的，也是“官方正确”的做法。

对于许多类型的设备，此行为根本不重要。如果内核希望在系统睡眠时相信您的USB键盘已被拔出并在系统醒来时插入了一个新键盘，那又怎样呢？当您在上面输入时，它仍将正常工作。

不幸的是，会出现问题，特别是与大容量存储设备有关。效果与设备在系统挂起期间实际上被拔出的效果完全相同。如果您在设备上有已挂载的文件系统，则运气不佳-该文件系统中的所有内容现在无法访问。如果您的根文件系统位于该设备上，则特别令人讨厌，因为您的系统将立即崩溃。

丢失电源不是唯一需要担心的机制。任何中断电源会话的东西都会具有相同的效果。例如，即使在系统睡眠时保持挂起电流，但在许多系统中，在唤醒的初始阶段，固件（即BIOS）会重置主板的USB主机控制器。结果所有电源会话都被破坏，一次又一次地好像您拔了所有USB设备一样。是的，这完全是BIOS的错，但是这对您没有任何好处，除非您能说服BIOS供应商解决这个问题（祝你好运！）。

对于许多系统，在挂起到RAM后，USB主机控制器将被重置。在几乎所有系统中，休眠期间没有挂起电流可用（也称为swsusp或挂起到磁盘）。您可以在恢复后检查内核日志，以查看是否发生了其中任何一种情况；寻找“根集线器失去电源或被重置”的行。

实际上，人们被迫在挂起之前卸载USB设备上的任何文件系统。如果根文件系统位于USB设备上，则无法挂起系统。 （所有正确，它可以挂起-但是它一醒来就会崩溃，这也不好。）

### 这是什么解决方案？

内核包括一个名为USB-persist的功能。它通过允许核心USB设备数据结构跨越电源中断保持不变来解决这些问题。

它的工作方式是这样的。如果内核看到USB主机控制器在恢复时不处于预期状态（例如，如果控制器被重置或失去电源），则它对每个下面控制器的被设置为“persist”属性的USB设备应用持续性检查。它不会尝试恢复设备；当电源会话消失时，这样做是不可能的。相反，它发出USB端口重置，然后再次enum设备。（这正是每当重置USB设备时发生的事情。）如果重新enum显示现在附加到该端口的设备具有与之前相同的说明符，包括厂商和产品ID，则内核将继续使用相同的设备结构。实际上，内核将设备视为仅已被重置，而不是被拔掉。

如果主机控制器处于预期状态，但USB设备已被拔掉并重新插入，或者USB设备未能执行正常恢复，则会发生同样的事情。

如果现在没有设备连接到端口上，或者说明符与内核记忆的不同，则处理方式符合预期。内核销毁旧设备结构，并像旧设备已拔出并插入新设备一样行事。

最终的结果是USB设备仍然可用且可用。文件系统挂载和内存映射不受影响，世界现在是一个良好而幸福的地方。

请注意，“USB-persist”功能仅适用于启用该功能的那些设备。您可以通过执行以下操作打开该功能（作为root用户）

echo 1 >/sys/bus/usb/devices/.../power/persist

其中“…”应填写设备的ID。将0写入以代替1来禁用该功能。对于集线器，该功能将自动且永久启用，并且power/persist文件甚至不存在，因此您只需要关注在其确实很重要的设备上设置它。

### 这是最佳解决方案吗？

也许不是。可以说，跟踪挂载的文件系统和跨设备断开连接的内存映射应该由一个集中的逻辑卷管理器来处理。这样的解决方案将允许您插入USB闪存设备，创建与之关联的持久卷，拔出闪存设备，稍后将其插入，并仍然具有与该设备关联的相同持久卷。因此，它比USB-persist更具有广泛影响。

另一方面，编写一个持久卷管理器是一个巨大的工作，并且使用它需要用户的大量输入。这个解决方案更快、更容易 - 而且现在已存在，这是它最大的优点！

此外，“USB-persist”功能适用于所有USB设备，而不仅仅是大容量存储设备。它可能同样适用于其他设备类型，例如网络接口。

### 警告USB-persist可能危险！

在恢复中断的电源会话时，内核会尽最大努力确保USB设备没有更改；也就是说，相同的设备仍然插入端口。但是，检查并不能保证100％准确。

如果您用同一类型的另一个USB设备（同一制造商，相同的ID等）替换一个USB设备，则内核无法检测到更改，这是一个很好的机会。串行号字符串和其他说明符与内核存储的值进行比较，但这可能没有帮助，因为制造商经常在他们的设备中完全省略序列号。

此外，您完全可以让一个USB设备保持不变，同时更换介质。如果您在系统处于睡眠状态时更换USB读卡器中的闪存存储卡，则内核将无法知道您这样做了。内核将假定没有发生任何事情，并将继续使用旧卡的分区表，inode和内存映射。

如果内核以这种方式上当受骗，几乎肯定会导致数据损坏并使系统崩溃。你只能怪自己。

对于设置为避免重置异常属性的那些设备，持久可能会失败，因为它们可能在重置后发生变化。

注意使用USB-persist可能会很危险！请自行承担风险！

话虽如此，大多数情况下都不应该出现什么问题。 USB持续功能非常有用，请充分利用。

## USB错误代码

修订日期2004年10月21日

这是usbcore可能返回的所有可能错误代码（及其解释）的文档。

其中一些是由主机控制器驱动程序（HCD）返回的，这些驱动程序只通过usbcore看到。通常来说，除了传输速度相关的行为和报告某些故障的方式之外，所有HCD都应该表现相同。

### 由usb\_submit\_urb()返回的错误代码

非USB特定：

|  |  |
| --- | --- |
| 0 | URB submission went fine |
| -ENOMEM | no memory for allocation of internal structures |

USB专用:

|  |  |
| --- | --- |
| -EBUSY | The URB is already active. |
| -ENODEV | specified USB-device or bus doesn’t exist |
| -ENOENT | specified interface or endpoint does not exist or is not enabled |
| -ENXIO | host controller driver does not support queuing of this type of urb. (treat as a host controller bug.) |
| -EINVAL | 1）Invalid transfer type specified (or not supported)  2）Invalid or unsupported periodic transfer interval  3）ISO: attempted to change transfer interval  4）ISO: number\_of\_packets is < 0  5）various other cases |
| -EXDEV | ISO: URB\_ISO\_ASAP wasn’t specified and all the frames the URB would be scheduled in have already expired. |
| -EFBIG | Host controller driver can’t schedule that many ISO frames. |
| -EPIPE | The pipe type specified in the URB doesn’t match the endpoint’s actual type. |
| -EMSGSIZE | 1）endpoint maxpacket size is zero; it is not usable in the current interface altsetting.  2）ISO packet is larger than the endpoint maxpacket.  3）requested data transfer length is invalid: negative or too large for the host controller. |
| -EBADR | The wLength value in a control URB’s setup packet does not match the URB’s transfer\_buffer\_length. |
| -ENOSPC | This request would overcommit the usb bandwidth reserved for periodic transfers (interrupt, isochronous). |
| -ESHUTDOWN | The device or host controller has been disabled due to some problem that could not be worked around. |
| -EPERM | Submission failed because urb->reject was set. |
| -EHOSTUNREACH | URB was rejected because the device is suspended. |
| -ENOEXEC | A control URB doesn’t contain a Setup packet. |

### 在urb->status中返回的错误代码或在iso\_frame\_desc[n].status中返回的错误代码（用于ISO）

USB设备驱动程序只能在完成处理程序中测试urb状态值。这是因为否则在一个CPU上更新这些值的HCD和在另一个CPU上测试它们的设备驱动程序之间会产生竞争。

即使报告了错误，传输的实际长度也可能为正。这是因为传输通常涉及多个数据包，因此在错误停止进一步的端点I/O之前，一个或多个数据包可能已经完成。

对于等时USB，仅当URB取消连接，设备被移除，主机控制器被禁用或总传输长度小于请求的长度并且设置了URB\_SHORT\_NOT\_OK标志时，urb状态值才为非零。等时URB的完成处理程序应该只看到urb->status设置为零、-ENOENT、-ECONNRESET、-ESHUTDOWN或-EREMOTEIO的情况。单个帧说明符状态字段可能报告更多的状态代码。

|  |  |
| --- | --- |
| 0 | Transfer completed successfully |
| -ENOENT | URB was synchronously unlinked by [usb\_unlink\_urb()](https://docs.kernel.org/driver-api/usb/usb.html" \l "c.usb_unlink_urb" \o "usb_unlink_urb) |
| -EINPROGRESS | URB still pending, no results yet (That is, if drivers see this it’s a bug.) |
| -EPROTO1077[1,2] | 1. bitstuff error 2. no response packet received within the prescribed bus turn-around time 3. unknown USB error |
| -EILSEQ [1,2] | 1. CRC mismatch 2. no response packet received within the prescribed bus turn-around time 3. unknown USB error   注 that often the controller hardware does not distinguish among cases 1), 2), and 3), so a driver cannot tell whether there was a protocol error, a failure to respond (often caused by device disconnect), or some other fault. |
| -ETIME [1] | No response packet received within the prescribed bus turn-around time. This error may instead be reported as -EPROTO or -EILSEQ. |
| -ETIMEDOUT | Synchronous USB message functions use this code to indicate timeout expired before the transfer completed, and no other error was reported by HC. |
| -EPIPE [2] | Endpoint stalled. For non-control endpoints, reset this status with [usb\_clear\_halt()](https://docs.kernel.org/driver-api/usb/usb.html" \l "c.usb_clear_halt" \o "usb_clear_halt). |
| -ECOMM | During an IN transfer, the host controller received data from an endpoint faster than it could be written to system memory |
| -ENOSR | During an OUT transfer, the host controller could not retrieve data from system memory fast enough to keep up with the USB data rate |
| -EOVERFLOW [1](https://docs.kernel.org/driver-api/usb/error-codes.html" \l "f1) | The amount of data 返回ed by the endpoint was greater than either the max packet size of the endpoint or the remaining buffer size. “Babble”. |
| -EREMOTEIO | The data read from the endpoint did not fill the specified buffer, and URB\_SHORT\_NOT\_OK was set in urb->transfer\_flags. |
| -ENODEV | Device was removed. Often preceded by a burst of other errors, since the hub driver doesn’t detect device removal events immediately. |
| -EXDEV | ISO transfer only partially completed (only set in iso\_frame\_desc[n].status, not urb->status) |
| -EINVAL | ISO madness, if this happens: Log off and go home |
| -ECONNRESET | URB was asynchronously unlinked by [usb\_unlink\_urb()](https://docs.kernel.org/driver-api/usb/usb.html" \l "c.usb_unlink_urb" \o "usb_unlink_urb) |
| -ESHUTDOWN | The device or host controller has been disabled due to some problem that could not be worked around, such as a physical disconnect. |

1. (1,2,3):像-EPROTO、-EILSEQ和-EOVERFLOW这样的错误代码通常表示硬件问题，例如错误的设备（包括固件）或电缆。
2. (1,2,3,4):这也是不同类型的主机控制器用来指示传输因设备断开而失败的几个代码之一。在集线器驱动程序开始断开处理之前的间隔期间，设备可能会接收到每个请求的此类故障报告。

### 由usbcore函数返回的错误代码

**注意**

也期望其他提交和传输状态代码。

usb\_register():

|  |  |
| --- | --- |
| -EINVAL | error during registering new driver |

usb\_get\_\*/usb\_set\_\*(), usb\_control\_msg(), usb\_bulk\_msg():

|  |  |
| --- | --- |
| -ETIMEDOUT | Timeout expired before the transfer completed. |

编写USB设备驱动程序

简介

Linux USB子系统已经从2.2.7内核中仅支持两种不同类型的设备（鼠标和键盘），增长到2.4内核中支持超过20种不同类型的设备。Linux目前支持几乎所有USB类设备（键盘、鼠标、调制解调器、打印机和扬声器等标准类型的设备）以及越来越多的厂商特定设备（例如USB串口转换器、数码相机、以太网设备和MP3播放器）。有关当前支持的不同USB设备的完整列表，请参见资源。

在Linux上没有支持的其余类型的USB设备几乎都是厂商特定的设备。每个供应商都决定实现一种自**定义**协议以与其设备通信，因此通常需要创建一个自**定义**驱动程序。一些供应商公开其USB协议并协助创建Linux驱动程序，而其他供应商则不公开它们，并且开发人员不得不进行逆向工程。有关一些有用的逆向工程工具的链接，请参见资源。

由于每种不同的协议都会导致创建新的驱动程序，因此我编写了一个通用的USB驱动程序框架，该框架是根据内核源代码树中的pci-skeleton.c文件模拟的，许多PCI网络驱动程序都是基于此文件创建的。这个USB框架可以在内核源代码树中的drivers/usb/usb-skeleton.c中找到。在本文中，我将介绍框架驱动程序的基础知识，解释不同的部分以及需要进行哪些操作才能将其自**定义**为特定设备。

Linux USB基础知识

如果您要编写Linux USB驱动程序，请熟悉USB协议规范。它可以在USB主页（请参见资源）上找到，以及许多其他有用的文档。在USB工作设备列表（请参见资源）中可以找到Linux USB子系统的优秀介绍。它解释了Linux USB子系统的结构，并向读者介绍了USB urbs（USB请求块）的概念，这对于USB驱动程序是必不可少的。

Linux USB驱动程序需要做的第一件事是向Linux USB子系统注册自己，提供一些与驱动程序支持的设备和插入或从系统中卸载支持驱动程序的设备时要调用的函数有关的信息。所有这些信息都在usb\_driver结构中传递给USB子系统。框架驱动程序声明一个usb\_driver如下:

static struct usb\_driver skel\_driver = {

.name = "skeleton",

.probe = skel\_probe,

.disconnect = skel\_disconnect,

.fops = &skel\_fops,

.minor = USB\_SKEL\_MINOR\_BASE,

.id\_table = skel\_table,

};

变量名称是**说明**驱动程序的字符串，用于打印到系统日志中的信息性消息。当看到或移除与id\_table变量提供的信息匹配的设备时，将调用probe和disconnect函数指针。

fops和minor变量是可选的。大多数USB驱动程序钩入其他内核子系统，如SCSI、网络或TTY子系统。这些类型的驱动程序在其他内核子系统中注册自己，并通过该接口提供任何用户空间交互。但是，对于没有匹配内核子系统的驱动程序，例如MP3播放器或扫描仪，需要一种与用户空间交互的方法。USB子系统提供了一种注册次要设备号和一组file\_operations函数指针的方法，以启用这种用户空间交互。骨架驱动程序需要这种接口，因此它提供了一个次要起始号和指向其file\_operations函数的指针。

然后，通过调用usb\_register()函数向USB驱动程序注册，通常在驱动程序的init函数中进行，如下所示:

static int \_\_init usb\_skel\_init(void)

{

int result;

/\* register this driver with the USB subsystem \*/

result = usb\_register(&skel\_driver);

if (result < 0) {

err("usb\_register failed for the "\_\_FILE\_\_ "driver."

"Error number %d", result);

return -1;

}

return 0;

}

module\_init(usb\_skel\_init);

当驱动程序从系统中卸载时，需要使用usb\_deregister()函数将其自身与USB子系统取消注册:

static void \_\_exit usb\_skel\_exit(void)

{

/\* deregister this driver with the USB subsystem \*/

usb\_deregister(&skel\_driver);

}

module\_exit(usb\_skel\_exit);

为了使linux-hotplug系统在插入设备时自动加载驱动程序，需要创建MODULE\_DEVICE\_TABLE。以下代码告诉热插拔脚本，该模块支持具有特定供应商和产品ID的单个设备:

/\* table of devices that work with this driver \*/

static struct usb\_device\_id skel\_table [] = {

{USB\_DEVICE(USB\_SKEL\_VENDOR\_ID,USB\_SKEL\_PRODUCT\_ID) },

{ } /\* Terminating entry \*/

};

MODULE\_DEVICE\_TABLE (usb, skel\_table);

还有其他可以用于说明支持整个USB驱动程序类别的struct usb\_device\_id的宏。有关此内容的更多信息，请参见usb.h。

### 设备操作

当插入与USB核心注册的设备ID模式匹配的设备时，将调用probe函数。将usb\_device结构、接口号和接口ID传递给函数:

static int skel\_probe(struct usb\_interface \*interface,

const struct usb\_device\_id \*id)

驱动程序现在需要验证这个设备是否实际上可以接受。如果是，它将返回0。如果不是，或者在初始化期间发生任何错误，则从probe函数返回错误代码(例如-ENOMEM或-ENODEV)。

在骨架驱动程序中，我们确定哪些端点被标记为批量输入和输出。我们创建缓冲区来容纳将要从设备发送和接收的数据，初始化用于将数据写入设备的USB urb。

相反，在从USB总线中移除设备时，将使用设备指针调用disconnect函数。此时，驱动程序需要清除分配的任何私有数据，并关闭在USB系统中未决的urb。

现在，设备已经插入系统并且驱动程序已经绑定到设备上，任何从用户程序尝试与设备交互的file\_operations结构的函数都将被调用。首先调用的函数是open，因为程序尝试打开设备进行I/O操作。我们增加了私有使用计数，并将指针保存在文件结构的内部结构中。这样做是为了使将来调用文件操作时，驱动程序能够确定用户正在处理的设备。所有这些都是使用以下代码完成的:

/\* increment our usage count for the device \*/

++skel->open\_count;

/\* save our object in the file's private structure \*/

file->private\_data = dev;

调用打开函数后，将调用读取和写入函数以接收和发送数据到设备。在skel\_write函数中，我们接收到一个指向用户想要发送到设备的数据的指针和数据的大小。该函数基于它创建的写urb的大小(这个大小取决于设备具有的批量输出端点的大小)确定它可以发送多少数据到设备。然后它将数据从用户空间复制到内核空间，将urb指向数据，并将urb提交给USB子系统。这可以在以下代码中看到:

/\* we can only write as much as 1 urb will hold \*/

bytes\_written = (count > skel->bulk\_out\_size) ? skel->bulk\_out\_size : count;

/\* copy the data from user space into our urb \*/

copy\_from\_user(skel->write\_urb->transfer\_buffer, buffer, bytes\_written);

/\* set up our urb \*/

usb\_fill\_bulk\_urb(skel->write\_urb,

skel->dev,

usb\_sndbulkpipe(skel->dev, skel->bulk\_out\_endpointAddr),

skel->write\_urb->transfer\_buffer,

bytes\_written,

skel\_write\_bulk\_callback,

skel);

/\* send the data out the bulk port \*/

result = usb\_submit\_urb(skel->write\_urb);

if (result) {

err("Failed submitting write urb, error %d", result);

}

当使用usb\_fill\_bulk\_urb（）函数将写urb填充了适当的信息后，我们将urb的完成回调指向调用我们自己的skel\_write\_bulk\_callback函数。当urb由USB子系统完成时，就会调用这个函数。回调函数在中断**上下文**中调用，因此在此时要小心不要做太多的处理。我们的skel\_write\_bulk\_callback的实现只是报告urb是否成功完成，然后返回。

读函数与写函数的工作方式略有不同，我们不使用urb从设备向驱动程序传输数据。相反，我们调用usb\_bulk\_msg（）函数，该函数可以用于从设备发送或接收数据，而无需创建urb并处理urb完成回调函数。我们调用usb\_bulk\_msg（）函数，将缓冲区指定为从设备接收任何数据的位置，并指定超时值。如果超时期限到期而没有从设备接收到任何数据，该函数将失败并返回错误消息。可以使用下面的代码进行演示:

/\* do an immediate bulk read to get data from the device \*/

retval = usb\_bulk\_msg (skel->dev,

usb\_rcvbulkpipe (skel->dev,

skel->bulk\_in\_endpointAddr),

skel->bulk\_in\_buffer,

skel->bulk\_in\_size,

&count, HZ\*10);

/\* if the read was successful, copy the data to user space \*/

if (!retval) {

if (copy\_to\_user (buffer, skel->bulk\_in\_buffer, count))

retval = -EFAULT;

else

retval = count;

}

usb\_bulk\_msg（）函数非常有用，可用于从设备进行单次读取或写入。但是，如果需要不断地读取或写入设备，则建议设置自己的urb并将它们提交给USB子系统。

当用户程序释放用于与设备通信的文件句柄时，驱动程序中的release函数将被调用。在此函数中，我们递减私有使用计数并等待可能的待处理写入:

/\* decrement our usage count for the device \*/

--skel->open\_count;

USB驱动程序必须顺利处理的更难的问题之一是USB设备随时可能从系统中移除，即使程序目前正在与它通信。它需要能够关闭任何当前的读写并通知用户空间程序该设备不再存在。以下代码（skel\_delete函数）是如何执行此操作的示例:

static inline void skel\_delete (struct usb\_skel \*dev)

{

kfree (dev->bulk\_in\_buffer);

if (dev->bulk\_out\_buffer != NULL)

usb\_free\_coherent (dev->udev, dev->bulk\_out\_size,

dev->bulk\_out\_buffer,

dev->write\_urb->transfer\_dma);

usb\_free\_urb (dev->write\_urb);

kfree (dev);

}

如果程序当前打开了设备句柄，我们将标志device\_present重置为false。对于每个读取、写入、释放和其他需要存在设备的功能，驱动程序首先检查此标志以查看设备是否仍然存在。如果不存在，则释放该设备已消失，并向用户空间程序返回-ENODEV错误。当最终调用释放函数时，它确定是否存在设备，如果没有，则执行清理工作，此情况下相当于没有打开设备（请参见列表5）。

### 同步数据

此usb-skeleton驱动程序没有示例展示如何发送中断或等时数据到设备或从设备接收数据。中断数据几乎与批量数据相同，有少许差异。等时数据通过连续流从设备发送到或接收数据的方式工作。音频和视频相机驱动程序是处理等时数据的很好的示例，如果您也需要处理等时数据，那么这些驱动程序会很有用。

### 结论

编写Linux USB设备驱动程序并不是一项困难的任务，正如usb-skeleton驱动程序所示。该驱动程序与其他当前的USB驱动程序相结合，应该足够提供足够的示例，以帮助初学者在最短的时间内创建一个工作驱动程序。linux-usb-devel邮件列表档案还包含许多有用的信息。

### 资源

Linux USB项目：http://www.linux-usb.org/

Linux 热插拔项目：http://linux-hotplug.sourceforge.net/

Linux USB 工作设备列表： <http://www.qbik.ch/usb/devices/>

linux-usb-devel 邮件列表档案： [http://marc.theaimsgroup.com/ ?l=linux-usb-devel](http://marc.theaimsgroup.com/?l=linux-usb-devel)

Linux USB 设备驱动程序编程指南： <http://lmu.web.psi.ch/docu/manuals/software_manuals/linux_sl/usb_linux_programming_guide.pdf>

USB主页https://www.usb.org

## Synopsys DesignWare Core超速USB 3.0控制器

### 介绍

Synopsys DesignWare Core超速USB 3.0控制器（以下简称DWC3）是一个符合USB超速标准的控制器，可以配置为以下四种方式之一：

1）仅外围设备配置

2）仅主机配置

3）双角色配置

4）集线器配置

Linux目前支持该控制器的多个版本。在此写作时，已知测试版本范围从2.02a到3.10a。一般来说，任何版本高于2.02a都应该可以可靠地工作。

目前，该驱动程序有许多已知的用户，按字母顺序排列

Cavium

Intel公司

高通

瑞芯微

ST

三星

德州仪器

赛灵思

### 功能概述

有关DWC3版本支持的功能的详细信息，请查阅您的IP团队和/或Synopsys DesignWare Core超速USB 3.0控制器数据手册。以下是驱动程序在此写作时支持的功能列表：

最多16个双向端点（包括控制管道- ep0）

灵活的端点配置

同时支持IN和OUT传输

分散列表支持

每个端点高达256个TRB[1]

支持所有传输类型（控制、块、中断和等时）

超速块流

链路电源管理

用于调试的跟踪事件

DebugFS[3]接口

这些功能都与许多内置的小工具驱动程序一起使用。我们已经验证了ConfigFS[4]和传统小工具驱动程序。

### 驱动程序设计

DWC3驱动程序位于drivers/usb/dwc3/目录下。与此驱动程序相关的所有文件都在此一个目录下。这使新手可以轻松阅读代码并了解其行为方式。

由于DWC3的配置灵活性，驱动程序在某些地方有点复杂，但应该非常直观易懂。

该驱动程序的最大部分涉及小工具API。

### 已知限制

像任何其他硬件一样，DWC3具有自己的一组限制。为避免关于此类问题的不断提问，我们决定在此文档中进行记录，并有单一位置供我们可以指向用户。

#### OUT传输大小要求

根据Synopsys的数据手册，所有OUT传输TRB[1]的大小字段必须设置为端点的wMaxPacketSize的整数倍。这意味着例如为接收Mass Storage CBW[5]，req->length必须设置为wMaxPacketSize的整数倍的值（超速为1024，高速为512等），或者DWC3驱动程序必须添加指向扔掉的缓冲区的链式TRB以用于剩余长度。没有这个，OUT传输将不会启动。

值得注意的是，截至此写作时，这不会成为问题，因为DWC3完全能够附加链接的TRB以用于剩余长度，并从小工具驱动程序完全隐藏此详细信息。仍然值得一提，因为这似乎是关于DWC3和非工作传输的最大查询来源。

#### TRB环大小限制

我们目前对每个端点有256个TRB[1]的硬限制，最后一个TRB是指向第一个的Link TRB 2。这个限制是任意的，但它有助于添加恰好4096字节或1页。

DWC3驱动程序将尽力应对超过255个请求，并且在大多数情况下，它应该正常工作。但是，这不是经常性练习的事情。如果您遇到任何问题，请参见下面的报告错误部分。

### 报告错误

每当您遇到DWC3的问题时，首先要确保：

您正在运行来自Linus的最新标签；

您可以在DWC3没有任何树外更改的情况下重现错误；

您已经检查了它不是主机机器的故障。

在验证了所有这些之后，以下是如何捕获足够的信息以便我们能够帮助您的方式。

#### 所需信息

DWC3完全依赖跟踪事件进行调试。一切都在那里公开，还会向DebugFS 3公开一些额外的位。

为了捕获DWC3的跟踪事件，您应在将USB电缆插入主机机器之前运行以下命令:

# mkdir -p /d

# mkdir -p /t

# mount -t debugfs none /d

# mount -t tracefs none /t

# echo 81920 > /t/buffer\_size\_kb

# echo 1 > /t/events/dwc3/enable

完成上述步骤后，您可以连接您的USB电缆并重现问题。在问题重现后，按照以下方式复制跟踪文件和寄存器转储:

# cp /t/trace /root/trace.txt

# cat /d/\*dwc3\*/regdump > /root/regdump.txt

请确保压缩trace.txt和regdump.txt到一个tar包中，并将邮件发送到我的邮箱，并抄送linux-usb。如果您想确保我会帮助您，可以在主题行中按照以下格式写

[BUG报告]USBdwc3在执行XYZ时出现Bug

在邮件正文中，请详细说明你所做的操作，使用了哪个gadget驱动程序，如何重现问题，使用了哪种SoC，主机机器上运行哪个操作系统（及其版本）。

有了所有这些信息，我们应该能够理解正在发生什么，并对您提供帮助。

### 调试

首先，声明一下

免责声明DebugFS和/或TraceFS中可用的信息可能会随着任何Linux核心主要版本的发布而随时更改。如果编写脚本，请勿假定当前格式中可用的信息存在。

话说回来，让我们继续进行。

如果您想要调试自己的问题，您就应该会受到掌声吧。

不管怎样，没有什么需要说的，除了跟踪事件在解决DWC3问题时非常有帮助外。此外，访问Synopsys Databook对此案件非常有价值。

USB信号分析器有时会有帮助，但并不是必需的，不看线路也可以理解很多内容。

如有需要，请随时向我发送电子邮件，并抄送linux-usb，我会提供帮助。

#### DebugFS

DebugFS非常适合收集DWC3和/或任何端点的快照。

在DWC3的DebugFS目录下，您将找到以下文件和目录:

ep[0..15] {in，out} / link\_state regdump testmode

**link\_state**

读取link\_state时，将打印出U0、U1、U2、U3、SS.Disabled、RX.Detect、SS.Inactive、Polling、Recovery、Hot Reset、Compliance、Loopback、Reset、Resume或UNKNOWN链路状态之一。

此文件也可以写入，以将链接强制为上述状态之一。

**regdump**

文件名不言自明。读取regdump时，将打印出DWC3的寄存器转储。请注意，可以使用grep查找所需的信息。

**testmode**

读取testmode时，将打印出指定的USB 2.0测试模式（test\_j、test\_k、test\_se0\_nak、test\_packet、test\_force\_enable）中的一个名称，或者在当前未执行任何测试的情况下输出字符串“no test”。

为此，可以将相同的字符串写入文件开始任何这些测试模式，DWC3将进入请求的测试模式。

**ep[0..15]{in,out}**

为每个端点，我们公开一个遵循命名约定ep$num$dir（ep0in，ep0out，ep1in，...）的目录。在每个这些目录中，您将找到以下文件

descriptor\_fetch\_queue event\_queue rx\_fifo\_queue rx\_info\_queue rx\_request\_queue transfer\_type trb\_ring tx\_fifo\_queue tx\_request\_queue

有访问Synopsys Databook，您可以对它们上面的信息进行解码。

**transfer\_type**

读取transfer\_type时，将根据端点说明符输出control、bulk、interrupt或isochronous之一。如果尚未启用端点，则将打印--。

**trb\_ring**

读取trb\_ring时，将打印有关环上所有TRB的详细信息。它还会告诉您我们的enqueue和dequeue指针在环中的位置:

buffer\_addr,size,type,ioc,isp\_imi,csp,chn,lst,hwo

000000002c754000,481,normal,1,0,1,0,0,0

000000002c75c000,481,normal,1,0,1,0,0,0

000000002c780000,481,normal,1,0,1,0,0,0

000000002c788000,481,normal,1,0,1,0,0,0

000000002c78c000,481,normal,1,0,1,0,0,0

000000002c754000,481,normal,1,0,1,0,0,0

000000002c75c000,481,normal,1,0,1,0,0,0

000000002c784000,481,normal,1,0,1,0,0,0

000000002c788000,481,normal,1,0,1,0,0,0

000000002c78c000,481,normal,1,0,1,0,0,0

000000002c790000,481,normal,1,0,1,0,0,0

000000002c758000,481,normal,1,0,1,0,0,0

000000002c780000,481,normal,1,0,1,0,0,0

000000002c788000,481,normal,1,0,1,0,0,0

000000002c790000,481,normal,1,0,1,0,0,0

000000002c758000,481,normal,1,0,1,0,0,0

000000002c780000,481,normal,1,0,1,0,0,0

000000002c784000,481,normal,1,0,1,0,0,0

000000002c788000,481,normal,1,0,1,0,0,0

000000002c78c000,481,normal,1,0,1,0,0,0

000000002c754000,481,normal,1,0,1,0,0,0

000000002c758000,481,normal,1,0,1,0,0,0

000000002c780000,481,normal,1,0,1,0,0,0

000000002c784000,481,normal,1,0,1,0,0,0

000000002c78c000,481,normal,1,0,1,0,0,0

000000002c790000,481,normal,1,0,1,0,0,0

000000002c758000,481,normal,1,0,1,0,0,0

000000002c780000,481,normal,1,0,1,0,0,0

000000002c788000,481,normal,1,0,1,0,0,0

000000002c790000,481,normal,1,0,1,0,0,0

000000002c758000,481,normal,1,0,1,0,0,0

000000002c780000,481,normal,1,0,1,0,0,0

000000002c788000,481,normal,1,0,1,0,0,0

000000002c790000,481,normal,1,0,1,0,0,0

000000002c758000,481,normal,1,0,1,0,0,0

000000002c780000,481,normal,1,0,1,0,0,0

000000002c788000,481,normal,1,0,1,0,0,0

000000002c790000,481,normal,1,0,1,0,0,0

000000002c758000,481,normal,1,0,1,0,0,0

000000002c780000,481,normal,1,0,1,0,0,0

000000002c788000,481,normal,1,0,1,0,0,0

000000002c790000,481,normal,1,0,1,0,0,0

000000002c758000,481,normal,1,0,1,0,0,0

000000002c780000,481,normal,1,0,1,0,0,0

000000002c788000,481,normal,1,0,1,0,0,0

000000002c790000,481,normal,1,0,1,0,0,0

000000002c758000,481,normal,1,0,1,0,0,0

000000002c780000,481,normal,1,0,1,0,0,0

000000002c788000,481,normal,1,0,1,0,0,0

000000002c790000,481,normal,1,0,1,0,0,0

000000002c758000,481,normal,1,0,1,0,0,0

000000002c780000,481,normal,1,0,1,0,0,0

000000002c788000,481,normal,1,0,1,0,0,0

000000002c790000,481,normal,1,0,1,0,0,0

000000002c758000,481,normal,1,0,1,0,0,0

000000002c780000,481,normal,1,0,1,0,0,0

000000002c78c000,481,normal,1,0,1,0,0,0

000000002c784000,481,normal,1,0,1,0,0,0

000000002c788000,481,normal,1,0,1,0,0,0

000000002c78c000,481,normal,1,0,1,0,0,0

000000002c754000,481,normal,1,0,1,0,0,0

000000002c758000,481,normal,1,0,1,0,0,0

000000002c780000,481,normal,1,0,1,0,0,0

000000002c788000,481,normal,1,0,1,0,0,0

000000002c790000,481,normal,1,0,1,0,0,0

000000002c758000,481,normal,1,0,1,0,0,0

000000002c780000,481,normal,1,0,1,0,0,0

000000002c758000,481,normal,1,0,1,0,0,0

000000002c780000,481,normal,1,0,1,0,0,0

000000002c78c000,481,normal,1,0,1,0,0,0

000000002c75c000,481,normal,1,0,1,0,0,0

000000002c78c000,481,normal,1,0,1,0,0,0

000000002c780000,481,normal,1,0,1,0,0,0

000000002c754000,481,normal,1,0,1,0,0,0

000000002c788000,481,normal,1,0,1,0,0,0

000000002c754000,481,normal,1,0,1,0,0,0

000000002c780000,481,normal,1,0,1,0,0,0

000000002c788000,481,normal,1,0,1,0,0,0

000000002c78c000,481,normal,1,0,1,0,0,0

000000002c790000,481,normal,1,0,1,0,0,0

000000002c754000,481,normal,1,0,1,0,0,0

000000002c758000,481,normal,1,0,1,0,0,0

000000002c75c000,481,normal,1,0,1,0,0,0

000000002c780000,481,normal,1,0,1,0,0,0

000000002c784000,481,normal,1,0,1,0,0,0

000000002c788000,481,normal,1,0,1,0,0,0

000000002c78c000,481,normal,1,0,1,0,0,0

000000002c790000,481,normal,1,0,1,0,0,0

000000002c754000,481,normal,1,0,1,0,0,0

000000002c758000,481,normal,1,0,1,0,0,0

000000002c75c000,512,normal,1,0,1,0,0,1 D

0000000000000000,0,UNKNOWN,0,0,0,0,0,0 E

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

0000000000000000,0,UNKNOWN,0,0,0,0,0,0

00000000381ab000,0,link,0,0,0,0,0,1

#### 跟踪事件

DWC3 还提供了几个跟踪事件，可以帮助我们在运行时收集有关驱动程序行为的信息。

为了使用这些事件，您必须在内核配置中启用 CONFIG\_FTRACE。

有关如何启用 DWC3 事件的详细信息，请参见报告错误部分。

以下子节将提供有关 DWC3 定义的每个事件类和每个事件的详细信息。

**MMIO**

在寻找错误时，查看每个 MMIO 访问有时很有用。因此，DWC3 提供了两个跟踪事件（一个用于 dwc3\_readl()，一个用于 dwc3\_writel()）。TP\_printk 格式如下:

TP\_printk("addr %p value %08x", \_\_entry->base + \_\_entry->offset,

\_\_entry->value)

**中断事件**

每个 IRQ 事件都可以记录并解码为人类可读的字符串。因为每个事件都是不同的，我们不提供除 TP\_printk 格式（如下所示）之外的示例:

TP\_printk("event (%08x): %s", \_\_entry->event,

dwc3\_decode\_event(\_\_entry->event, \_\_entry->ep0state))

**控制请求**

每个 USB 控制请求都可以记录到跟踪缓冲区中。输出格式为:

TP\_printk("%s", dwc3\_decode\_ctrl(\_\_entry->bRequestType,

\_\_entry->bRequest, \_\_entry->wValue,

\_\_entry->wIndex, \_\_entry->wLength)

)

请注意，标准控制请求将被解码为相应参数的人类可读字符串。类和供应商请求将打印出一个16进制格式的8字节序列。

**struct usb\_request的生命周期**

struct usb\_request的整个生命周期可以在跟踪缓冲区中跟踪。我们为分配、释放、排队、出队和归还中的每一个事件都会有一个事件。输出格式为:

TP\_printk("%s: req %p length %u/%u %s%s%s ==> %d",

\_\_get\_str(name), \_\_entry->req, \_\_entry->actual, \_\_entry->length,

\_\_entry->zero ? "Z" : "z",

\_\_entry->short\_not\_ok ? "S" : "s",

\_\_entry->no\_interrupt ? "i" : "I",

\_\_entry->status

)

**通用命令**

我们可以记录和解码每个通用命令及其完成代码。格式为:

TP\_printk("cmd '%s' [%x] param %08x --> status: %s",

dwc3\_gadget\_generic\_cmd\_string(\_\_entry->cmd),

\_\_entry->cmd, \_\_entry->param,

dwc3\_gadget\_generic\_cmd\_status\_string(\_\_entry->status)

)

**端点命令**

端点命令也可以与完成代码一起记录。格式为:

TP\_printk("%s: cmd '%s' [%d] params %08x %08x %08x --> status: %s",

\_\_get\_str(name), dwc3\_gadget\_ep\_cmd\_string(\_\_entry->cmd),

\_\_entry->cmd, \_\_entry->param0,

\_\_entry->param1, \_\_entry->param2,

dwc3\_ep\_cmd\_status\_string(\_\_entry->cmd\_status)

)

**TRB的生命周期**

TRB的生命期很简单。我们要么准备TRB，要么完成它。通过这两个事件，我们可以看到一个TRB如何随时间而变化。格式为:

TP\_printk("%s: %d/%d trb %p buf %08x%08x size %s%d ctrl %08x (%c%c%c%c:%c%c:%s)",

\_\_get\_str(name), \_\_entry->queued, \_\_entry->allocated,

\_\_entry->trb, \_\_entry->bph, \_\_entry->bpl,

({char \*s;

int pcm = ((\_\_entry->size >> 24) & 3) + 1;

switch (\_\_entry->type) {

case USB\_ENDPOINT\_XFER\_INT:

case USB\_ENDPOINT\_XFER\_ISOC:

switch (pcm) {

case 1:

s = "1x ";

break;

case 2:

s = "2x ";

break;

case 3:

s = "3x ";

break;

}

default:

s = "";

} s; }),

DWC3\_TRB\_SIZE\_LENGTH(\_\_entry->size), \_\_entry->ctrl,

\_\_entry->ctrl & DWC3\_TRB\_CTRL\_HWO ? 'H' : 'h',

\_\_entry->ctrl & DWC3\_TRB\_CTRL\_LST ? 'L' : 'l',

\_\_entry->ctrl & DWC3\_TRB\_CTRL\_CHN ? 'C' : 'c',

\_\_entry->ctrl & DWC3\_TRB\_CTRL\_CSP ? 'S' : 's',

\_\_entry->ctrl & DWC3\_TRB\_CTRL\_ISP\_IMI ? 'S' : 's',

\_\_entry->ctrl & DWC3\_TRB\_CTRL\_IOC ? 'C' : 'c',

dwc3\_trb\_type\_string(DWC3\_TRBCTL\_TYPE(\_\_entry->ctrl))

)

**端点的生命期**

一个端点的生命周期由启用和禁用操作概括，两者都可以被跟踪。格式为:

TP\_printk("%s: mps %d/%d streams %d burst %d ring %d/%d flags %c:%c%c%c%c%c:%c:%c",

\_\_get\_str(name), \_\_entry->maxpacket,

\_\_entry->maxpacket\_limit, \_\_entry->max\_streams,

\_\_entry->maxburst, \_\_entry->trb\_enqueue,

\_\_entry->trb\_dequeue,

\_\_entry->flags & DWC3\_EP\_ENABLED ? 'E' : 'e',

\_\_entry->flags & DWC3\_EP\_STALL ? 'S' : 's',

\_\_entry->flags & DWC3\_EP\_WEDGE ? 'W' : 'w',

\_\_entry->flags & DWC3\_EP\_TRANSFER\_STARTED ? 'B' : 'b',

\_\_entry->flags & DWC3\_EP\_PENDING\_REQUEST ? 'P' : 'p',

\_\_entry->flags & DWC3\_EP\_END\_TRANSFER\_PENDING ? 'E' : 'e',

\_\_entry->direction ? '<' : '>'

)

### 结构体 、方法和定义

#### struct dwc3\_event\_buffer

软件事件缓冲区表示

**定义**

struct dwc3\_event\_buffer {

void \*buf;

void \*cache;

unsigned int length;

unsigned int lpos;

unsigned int count;

unsigned int flags;

#define DWC3\_EVENT\_PENDING BIT(0);

dma\_addr\_t dma;

struct dwc3 \*dwc;

};

**成员**

buf

该缓冲区

cache

线程中断中使用的缓存

length

缓冲区大小

lpos

事件偏移量

count

上一次读取事件计数器时的缓存

flags

与此事件缓冲区相关的标志

dma

dma\_addr\_t类型的地址

dwc

指向DWC控制器的指针

#### struct dwc3\_ep

设备端端点表示

**定义**

struct dwc3\_ep {

struct usb\_ep endpoint;

struct list\_head pending\_list;

struct list\_head started\_list;

wait\_queue\_head\_t wait\_end\_transfer;

spinlock\_t lock;

void \_\_iomem \*regs;

struct dwc3\_trb \*trb\_pool;

dma\_addr\_t trb\_pool\_dma;

struct dwc3 \*dwc;

u32 saved\_state;

unsigned flags;

#define DWC3\_EP\_ENABLED BIT(0);

#define DWC3\_EP\_STALL BIT(1);

#define DWC3\_EP\_WEDGE BIT(2);

#define DWC3\_EP\_TRANSFER\_STARTED BIT(3);

#define DWC3\_EP\_PENDING\_REQUEST BIT(5);

#define DWC3\_EP\_END\_TRANSFER\_PENDING BIT(7);

#define DWC3\_EP0\_DIR\_IN BIT(31);

u8 trb\_enqueue;

u8 trb\_dequeue;

u8 number;

u8 type;

u8 resource\_index;

u32 frame\_number;

u32 interval;

char name[20];

unsigned direction:1;

unsigned stream\_capable:1;

};

**成员**

endpoint

USB端点

pending\_list

该端点的挂起请求列表

started\_list

该端点上已开始的请求列表

wait\_end\_transfer

等待完成的wait\_queue\_head\_t

lock

用于端点请求队列遍历的自旋锁

regs

指向第一个端点寄存器的指针

trb\_pool

事务缓冲区数组

trb\_pool\_dma

TRB\_POOL的DMA地址

dwc

指向DWC控制器的指针

saved\_state

保存在休眠期间的端点状态

flags

端点标志（wedged, stalled, ...）

trb\_enqueue

将“指针”加入TRB数组

trb\_dequeue

将“指针”从TRB数组中删除

number

端点编号（1-15）

type

设置为bmAttributes＆USB\_ENDPOINT\_XFERTYPE\_MASK

resource\_index

资源传输索引

frame\_number

设置我们要开始这个传输的帧编号（ISOC）

interval

启动ISOC传输的间隔

name

可读性强的名称，例如ep1out-bulk

direction

TX为真，RX为假

stream\_capable

启用流时为真

#### struct dwc3\_trb

传输请求块（硬件格式）

**定义**

struct dwc3\_trb {

u32 bpl;

u32 bph;

u32 size;

u32 ctrl;

};

**成员**

bpl

DW0-3

bph

DW4-7

size

DW8-B

ctrl

DWC-F

#### struct dwc3\_hwparams

HWPARAMS寄存器的副本

**定义**

struct dwc3\_hwparams {

u32 hwparams0;

u32 hwparams1;

u32 hwparams2;

u32 hwparams3;

u32 hwparams4;

u32 hwparams5;

u32 hwparams6;

u32 hwparams7;

u32 hwparams8;

};

**成员**

hwparams0

GHWPARAMS0

hwparams1

GHWPARAMS1

hwparams2

GHWPARAMS2

hwparams3

GHWPARAMS3

hwparams4

GHWPARAMS4

hwparams5

GHWPARAMS5

hwparams6

GHWPARAMS6

hwparams7

GHWPARAMS7

hwparams8

GHWPARAMS8

#### struct dwc3\_request

传输请求的表示

**定义**

struct dwc3\_request {

struct usb\_request request;

struct list\_head list;

struct dwc3\_ep \*dep;

struct scatterlist \*sg;

struct scatterlist \*start\_sg;

unsigned num\_pending\_sgs;

unsigned int num\_queued\_sgs;

unsigned remaining;

u8 epnum;

struct dwc3\_trb \*trb;

dma\_addr\_t trb\_dma;

unsigned unaligned:1;

unsigned direction:1;

unsigned mapped:1;

unsigned started:1;

unsigned zero:1;

};

**成员**

request

待传输的 usb\_request 结构

list

用于请求排队的 list\_head

dep

拥有该请求的 dwc3\_ep 结构

sg

指向第一个未完成的 scatterlist

start\_sg

指向下一个将要排队的 scatterlist

num\_pending\_sgs

待处理 scatterlist 的计数器

num\_queued\_sgs

已排队的 scatterlist 的计数器

remaining

剩余数据量

epnum

该请求所涉及的端点号

trb

指向 struct dwc3\_trb 的指针

trb\_dma

trb 的 DMA 地址

unaligned

如果是长度不被 maxp 整除的 OUT 端点，则为 true

direction

IN 或 OUT 方向标志

mapped

如果请求已进行 DMA 映射，则为 true

started

已启动请求

zero

需要零长度数据包

#### struct dwc3

控制器的表示

**定义**

struct dwc3 {

struct work\_struct drd\_work;

struct dwc3\_trb \*ep0\_trb;

void \*bounce;

void \*scratchbuf;

u8 \*setup\_buf;

dma\_addr\_t ep0\_trb\_addr;

dma\_addr\_t bounce\_addr;

dma\_addr\_t scratch\_addr;

struct dwc3\_request ep0\_usb\_req;

struct completion ep0\_in\_setup;

spinlock\_t lock;

struct device \*dev;

struct device \*sysdev;

struct platform\_device \*xhci;

struct resource xhci\_resources[DWC3\_XHCI\_RESOURCES\_NUM];

struct dwc3\_event\_buffer \*ev\_buf;

struct dwc3\_ep \*eps[DWC3\_ENDPOINTS\_NUM];

struct usb\_gadget gadget;

struct usb\_gadget\_driver \*gadget\_driver;

struct clk\_bulk\_data \*clks;

int num\_clks;

struct reset\_control \*reset;

struct usb\_phy \*usb2\_phy;

struct usb\_phy \*usb3\_phy;

struct phy \*usb2\_generic\_phy;

struct phy \*usb3\_generic\_phy;

bool phys\_ready;

struct ulpi \*ulpi;

bool ulpi\_ready;

void \_\_iomem \*regs;

size\_t regs\_size;

enum usb\_dr\_mode dr\_mode;

u32 current\_dr\_role;

u32 desired\_dr\_role;

struct extcon\_dev \*edev;

struct notifier\_block edev\_nb;

enum usb\_phy\_interface hsphy\_mode;

u32 fladj;

u32 irq\_gadget;

u32 otg\_irq;

u32 current\_otg\_role;

u32 desired\_otg\_role;

bool otg\_restart\_host;

u32 nr\_scratch;

u32 u1u2;

u32 maximum\_speed;

u32 revision;

#define DWC3\_REVISION\_173A 0x5533173a;

#define DWC3\_REVISION\_175A 0x5533175a;

#define DWC3\_REVISION\_180A 0x5533180a;

#define DWC3\_REVISION\_183A 0x5533183a;

#define DWC3\_REVISION\_185A 0x5533185a;

#define DWC3\_REVISION\_187A 0x5533187a;

#define DWC3\_REVISION\_188A 0x5533188a;

#define DWC3\_REVISION\_190A 0x5533190a;

#define DWC3\_REVISION\_194A 0x5533194a;

#define DWC3\_REVISION\_200A 0x5533200a;

#define DWC3\_REVISION\_202A 0x5533202a;

#define DWC3\_REVISION\_210A 0x5533210a;

#define DWC3\_REVISION\_220A 0x5533220a;

#define DWC3\_REVISION\_230A 0x5533230a;

#define DWC3\_REVISION\_240A 0x5533240a;

#define DWC3\_REVISION\_250A 0x5533250a;

#define DWC3\_REVISION\_260A 0x5533260a;

#define DWC3\_REVISION\_270A 0x5533270a;

#define DWC3\_REVISION\_280A 0x5533280a;

#define DWC3\_REVISION\_290A 0x5533290a;

#define DWC3\_REVISION\_300A 0x5533300a;

#define DWC3\_REVISION\_310A 0x5533310a;

#define DWC3\_REVISION\_IS\_DWC31 0x80000000;

#define DWC3\_USB31\_REVISION\_110A (0x3131302a | DWC3\_REVISION\_IS\_DWC31);

#define DWC3\_USB31\_REVISION\_120A (0x3132302a | DWC3\_REVISION\_IS\_DWC31);

enum dwc3\_ep0\_next ep0\_next\_event;

enum dwc3\_ep0\_state ep0state;

enum dwc3\_link\_state link\_state;

u16 u2sel;

u16 u2pel;

u8 u1sel;

u8 u1pel;

u8 speed;

u8 num\_eps;

struct dwc3\_hwparams hwparams;

struct dentry \*root;

struct debugfs\_regset32 \*regset;

u8 test\_mode;

u8 test\_mode\_nr;

u8 lpm\_nyet\_threshold;

u8 hird\_threshold;

u8 rx\_thr\_num\_pkt\_prd;

u8 rx\_max\_burst\_prd;

u8 tx\_thr\_num\_pkt\_prd;

u8 tx\_max\_burst\_prd;

const char \*hsphy\_interface;

unsigned connected:1;

unsigned delayed\_status:1;

unsigned ep0\_bounced:1;

unsigned ep0\_expect\_in:1;

unsigned has\_hibernation:1;

unsigned sysdev\_is\_parent:1;

unsigned has\_lpm\_erratum:1;

unsigned is\_utmi\_l1\_suspend:1;

unsigned is\_fpga:1;

unsigned pending\_events:1;

unsigned pullups\_connected:1;

unsigned setup\_packet\_pending:1;

unsigned three\_stage\_setup:1;

unsigned usb3\_lpm\_capable:1;

unsigned disable\_scramble\_quirk:1;

unsigned u2exit\_lfps\_quirk:1;

unsigned u2ss\_inp3\_quirk:1;

unsigned req\_p1p2p3\_quirk:1;

unsigned del\_p1p2p3\_quirk:1;

unsigned del\_phy\_power\_chg\_quirk:1;

unsigned lfps\_filter\_quirk:1;

unsigned rx\_detect\_poll\_quirk:1;

unsigned dis\_u3\_susphy\_quirk:1;

unsigned dis\_u2\_susphy\_quirk:1;

unsigned dis\_enblslpm\_quirk:1;

unsigned dis\_rxdet\_inp3\_quirk:1;

unsigned dis\_u2\_freeclk\_exists\_quirk:1;

unsigned dis\_del\_phy\_power\_chg\_quirk:1;

unsigned dis\_tx\_ipgap\_linecheck\_quirk:1;

unsigned tx\_de\_emphasis\_quirk:1;

unsigned tx\_de\_emphasis:2;

unsigned dis\_metastability\_quirk:1;

u16 imod\_interval;

};

**成员**

drd\_work

用于角色交换的工作队列

ep0\_trb

trb 用于 ctrl\_req

bounce

反弹缓冲区的地址

scratchbuf

暂存缓冲区的地址

setup\_buf

在处理 STD USB 请求时使用

ep0\_trb\_addr

ep0\_trb的 DMA 地址

bounce\_addr

反弹的DMA地址

scratch\_addr

scratchbuf 的 DMA 地址

ep0\_usb\_req

处理 STD USB 请求时使用的虚拟请求

ep0\_in\_setup

一次控制交接完成，进入设置阶段

lock

用于同步

dev

指向我们的结构设备的指针

sysdev

指向支持 DMA 的设备的指针

xhci

指向我们的 xHCI 孩子的指针

xhci\_resources

为我们的xhci孩子构建资源

ev\_buf

结构 dwc3\_event\_buffer 指针

eps

端点数组

gadget

外围控制器的设备端表示

gadget\_driver

指向小工具驱动程序的指针

clks

时钟阵列

num\_clks

时钟数量

reset

重置控制

usb2\_phy

指向 USB2 PHY 的指针

usb3\_phy

指向 USB3 PHY 的指针

usb2\_generic\_phy

指向 USB2 PHY 的指针

usb3\_generic\_phy

指向 USB3 PHY 的指针

phys\_ready

指示 PHY 已准备就绪的标志

ulpi

指向 ulpi 接口的指针

ulpi\_ready

指示 ULPI 已初始化的标志

regs

我们寄存器的基地址

regs\_size

地址空间大小

dr\_mode

请求的操作模式

current\_dr\_role

在双角色模式下操作的当前角色

desired\_dr\_role

在双角色模式下所需的操作角色

edev

分机句柄

edev\_nb

外部通知程序

hsphy\_mode

UTMI phy 模式，以下之一： - USBPHY\_INTERFACE\_MODE\_UTMI - USBPHY\_INTERFACE\_MODE\_UTMIW

fladj

帧长调整

irq\_gadget

外围控制器的 IRQ 号

otg\_irq

OTG IRQ 的 IRQ 号

current\_otg\_role

使用 OTG 块时的当前操作角色

desired\_otg\_role

使用 OTG 块时所需的操作角色

otg\_restart\_host

OTG 控制器需要重启主机的标志

nr\_scratch

临时缓冲区的数量

u1u2

仅用于版本 <1.83a 的解决方法

maximum\_speed

请求的最大速度（主要用于测试目的）

revision

修订登记内容

ep0\_next\_event

举办下一个预期活动

ep0state

端点零状态

link\_state

链接状态

u2sel

来自 Set SEL 请求的参数。

u2pel

来自 Set SEL 请求的参数。

u1sel

来自 Set SEL 请求的参数。

u1pel

来自 Set SEL 请求的参数。

speed

设备速度（超、高、全、低）

num\_eps

端点数

hwparams

hwparams 寄存器的副本

root

debugfs 根文件夹指针

regset

指向 regdump 文件的 debugfs 指针

test\_mode

当我们进入 USB 测试模式时为真

test\_mode\_nr

测试特征选择器

lpm\_nyet\_threshold

LPM NYET 响应阈值

hird\_threshold

HIRD 阈值

rx\_thr\_num\_pkt\_prd

定期 ESS 接收数据包计数

rx\_max\_burst\_prd

最大周期性 ESS 接收突发大小

tx\_thr\_num\_pkt\_prd

周期性 ESS 传输数据包计数

tx\_max\_burst\_prd

最大周期性 ESS 传输突发大小

hsphy\_interface

“utmi”或“ulpi”

connected

当我们连接到主机时为真，否则为假

delayed\_status

当小工具驱动程序请求延迟状态时为真

ep0\_bounced

当我们使用反弹缓冲区时为真

ep0\_expect\_in

当我们期望 DATA IN 传输时为真

has\_hibernation

当 dwc3 配置为休眠时为真

sysdev\_is\_parent

当 dwc3 设备具有父驱动程序时为真

has\_lpm\_erratum

当内核配置有 LPM Erratum 时为真。请注意，现在软件可以在运行时检测到这一点。

is\_utmi\_l1\_suspend

内核断言输出信号 0 - utmi\_sleep\_n 1 - utmi\_l1\_suspend\_n

is\_fpga

当我们使用 FPGA 板时为真

pending\_events

当我们有待处理的 IRQ 需要处理时为真

pullups\_connected

设置运行/停止位时为真

setup\_packet\_pending

当 FIFO 中有设置数据包时为真。解决方法

three\_stage\_setup

如果我们执行三相设置，请设置

usb3\_lpm\_capable

设置 hadrware 是否支持链路电源管理

disable\_scramble\_quirk

如果我们启用禁用加扰怪癖，请设置

u2exit\_lfps\_quirk

设置我们是否启用 u2exit lfps 怪癖

u2ss\_inp3\_quirk

设置我们是否为 U2/SS Inactive quirk 启用 P3 OK

req\_p1p2p3\_quirk

设置我们是否启用请求 p1p2p3 怪癖

del\_p1p2p3\_quirk

设置是否启用延迟 p1p2p3 怪癖

del\_phy\_power\_chg\_quirk

设置是否启用延迟 phy 电源更改怪癖

lfps\_filter\_quirk

设置是否启用 LFPS 过滤器怪癖

rx\_detect\_poll\_quirk

设置我们是否启用 rx\_detect 以轮询 lfps 怪癖

dis\_u3\_susphy\_quirk

设置我们是否禁用 usb3 挂起 phy

dis\_u2\_susphy\_quirk

设置我们是否禁用 usb2 挂起 phy

dis\_enblslpm\_quirk

如果我们清除 GUSB2PHYCFG 中的 enblslpm，则设置，禁用对 PHY 的挂起信号。

dis\_rxdet\_inp3\_quirk

如果我们在 P3 中禁用 Rx.Detect，则设置

dis\_u2\_freeclk\_exists\_quirk

如果我们清除 GUSB2PHYCFG 中的 u2\_freeclk\_exists，则设置 USB2 PHY 不提供自由运行的 PHY 时钟。

dis\_del\_phy\_power\_chg\_quirk

如果我们禁用延迟 phy 电源更改怪癖，请设置。

dis\_tx\_ipgap\_linecheck\_quirk

设置我们是否在 HS 传输期间禁用 u2mac 线路状态检查。

tx\_de\_emphasis\_quirk

如果我们启用 Tx 去加重怪癖则设置

tx\_de\_emphasis

Tx 去加重值 0 - -6dB 去加重 1 - -3.5dB 去加重 2 - 无去加重 3 - 保留

dis\_metastability\_quirk

设置为禁用亚稳态怪癖。

imod\_interval

以 250ns 的增量设置中断调节间隔或 0 禁用。

#### struct dwc3\_event\_depevt

设备端点事件

**定义**

struct dwc3\_event\_depevt {

u32 one\_bit:1;

u32 endpoint\_number:5;

u32 endpoint\_event:4;

u32 reserved11\_10:2;

u32 status:4;

#define DEPEVT\_STATUS\_TRANSFER\_ACTIVE BIT(3);

#define DEPEVT\_STATUS\_BUSERR BIT(0);

#define DEPEVT\_STATUS\_SHORT BIT(1);

#define DEPEVT\_STATUS\_IOC BIT(2);

#define DEPEVT\_STATUS\_LST BIT(3) ;

#define DEPEVT\_STATUS\_MISSED\_ISOC BIT(3) ;

#define DEPEVT\_STREAMEVT\_FOUND 1;

#define DEPEVT\_STREAMEVT\_NOTFOUND 2;

#define DEPEVT\_STATUS\_CONTROL\_DATA 1;

#define DEPEVT\_STATUS\_CONTROL\_STATUS 2;

#define DEPEVT\_STATUS\_CONTROL\_PHASE(n) ((n) & 3);

#define DEPEVT\_TRANSFER\_NO\_RESOURCE 1;

#define DEPEVT\_TRANSFER\_BUS\_EXPIRY 2;

u32 parameters:16;

#define DEPEVT\_PARAMETER\_CMD(n) (((n) & (0xf << 8)) >> 8);

};

**成员**

vone\_bit

指示这是端点事件（未使用）

endpoint\_number

端点数量

endpoint\_event

我们的事件0x00 - 保留0x01 - XferComplete 0x02 - XferInProgress 0x03 - XferNotReady 0x04 - RxTxFifoEvt（IN->Underrun，OUT->Overrun）0x05 - 保留0x06 - StreamEvt 0x07 - EPCmdCmplt

reserved11\_10

保留，请勿使用。

status

指示事件的状态。有关更多信息，请参阅数据手册。

parameters

当前事件的参数。有关更多信息，请参阅数据手册。

#### struct dwc3\_event\_devt

设备事件

**定义**

struct dwc3\_event\_devt {

u32 one\_bit:1;

u32 device\_event:7;

u32 type:4;

u32 reserved15\_12:4;

u32 event\_info:9;

u32 reserved31\_25:7;

};

**成员**

one\_bit

指示这是非端点事件（未使用）

device\_event

指示它是设备事件。应该读取为0x00

type

指示设备事件的类型。0 - DisconnEvt 1 - USBRst 2 - ConnectDone 3 - ULStChng 4 - WkUpEvt 5 - 保留6 - 暂停（在2.10a及以前版本上为EOPF）7 - SOF 8 - 保留9 - ErrticErr 10 - CmdCmplt 11 - EvntOverflow 12 - VndrDevTstRcved

reserved15\_12

保留，未使用

event\_info

关于此事件的信息

reserved31\_25

保留，未使用

#### struct dwc3\_event\_gevt

其他核心事件

**定义**

struct dwc3\_event\_gevt {

u32 one\_bit:1;

u32 device\_event:7;

u32 phy\_port\_number:4;

u32 reserved31\_12:20;

};

**成员**

one\_bit

表示这是非端点事件（未使用）

device\_event

表示它是（0x03）Carkit或（0x04）I2C事件。

phy\_port\_number

不言自明

reserved31\_12

保留，未使用。

#### union dwc3\_event

事件缓冲区内容的表示

**定义**

union dwc3\_event {

u32 raw;

struct dwc3\_event\_type type;

struct dwc3\_event\_depevt depevt;

struct dwc3\_event\_devt devt;

struct dwc3\_event\_gevt gevt;

};

**成员**

raw

原始的32位事件

type

事件的类型

depevt

设备端点事件

devt

设备事件

gevt

全局事件

#### struct dwc3\_gadget\_ep\_cmd\_params

端点命令参数的表示

**定义**

struct dwc3\_gadget\_ep\_cmd\_params {

u32 param2;

u32 param1;

u32 param0;

};

**成员**

param2

第三个参数

param1

第二个参数

param0

第一个参数

#### struct dwc3\_request \*next\_request(struct list\_head \*list)

获取给定列表中的下一个请求

**参数**

struct list\_head \*list

要操作的请求列表

**说明**

调用方应该注意加锁。此函数返回NULL或列表中可用的第一个请求。

#### void dwc3\_gadget\_move\_started\_request(struct dwc3\_request \*req)

将req移动到已启动列表

**参数**

struct dwc3\_request \*req

要移动的请求

**说明**

调用方应该注意加锁。此函数将req从其当前列表中移动到端点的已启动列表。

#### void dwc3\_gadget\_ep\_get\_transfer\_index(struct dwc3\_ep \*dep)

从HW获取传输资源索引

**参数**

struct dwc3\_ep \*dep

dwc3端点

**说明**

调用方应该注意加锁。返回给定端点的传输资源索引。

#### int dwc3\_gadget\_set\_test\_mode(struct dwc3 \*dwc, int mode)

启用USB2测试模式

**参数**

struct dwc3 \*dwc

指向我们的上下文struct 的指针

int mode

要设置的模式（J、K SE0 NAK、Force Enable）

**说明**

调用方应该注意加锁。此函数将在成功时返回0或在传递错误的测试选择器时返回-EINVAL。

#### int dwc3\_gadget\_get\_link\_state(struct dwc3 \*dwc)

获取USB链接的当前状态

**参数**

struct dwc3 \*dwc

指向我们的上下文struct 的指针

**说明**

调用方应该注意加锁。此函数将在成功时返回链接状态（> = 0），或在超时时返回-ETIMEDOUT。

#### int dwc3\_gadget\_set\_link\_state(struct dwc3 \*dwc, enum dwc3\_link\_state state)

将USB链接设置为特定状态

**参数**

struct dwc3 \*dwc

指向我们的上下文struct 的指针

enum dwc3\_link\_state state

要将链接设置为的状态

**说明**

调用方应该注意加锁。此函数将在成功时返回0或在超时时返回-ETIMEDOUT。

#### void dwc3\_ep\_inc\_trb(u8 \*index)

增加trb索引

**参数**

u8 \*index

要增加的TRB索引的指针。

**说明**

索引永远不应指向链接TRB。增加后，如果其指向链接TRB，则回到开头。链接TRB始终位于最后一个TRB条目。

#### void dwc3\_ep\_inc\_enq(struct dwc3\_ep \*dep)

增加端点的入队指针

**参数**

struct dwc3\_ep \*dep

我们正在增加其入队指针的端点

#### void dwc3\_ep\_inc\_deq(struct dwc3\_ep \*dep)

增加端点的出队指针

**参数**

struct dwc3\_ep \*dep

我们正在增加其出队指针的端点

#### void dwc3\_gadget\_giveback(struct dwc3\_ep \*dep, struct dwc3\_request \*req, int status)

调用struct usb\_request的->complete回调

**参数**

struct dwc3\_ep \*dep

请求所属的端点

struct dwc3\_request \*req

我们正在回传的请求

int status

请求的完成状态代码

**说明**

必须在控制器的锁保持和中断禁用的情况下调用。此函数将取消映射req并调用其->complete（）回调以通知上层它已经完成。

#### int dwc3\_send\_gadget\_generic\_command(struct dwc3 \*dwc, unsigned int cmd, u32 param)

为控制器发出通用命令

**参数**

struct dwc3 \*dwc

指向控制器上下文的指针

unsigned int cmd

要发出的命令

u32 param

命令参数

**说明**

调用方应该注意加锁。使用给定的param向dwc发出cmd并等待其完成。

#### int dwc3\_send\_gadget\_ep\_cmd(struct dwc3\_ep \*dep, unsigned int cmd, struct dwc3\_gadget\_ep\_cmd\_params \*params)

发出端点命令

**参数**

struct dwc3\_ep \*dep

要发出命令的端点

unsigned int cmd

要发出的命令

struct dwc3\_gadget\_ep\_cmd\_params \*params

命令的参数

**说明**

调用方应该注意加锁。此函数将使用给定的参数cmd向dep发出命令并等待其完成。

#### int dwc3\_gadget\_start\_config(struct dwc3\_ep \*dep)

配置端点资源

**参数**

struct dwc3\_ep \*dep

正在启用的端点

**说明**

向dep发出DWC3\_DEPCMD\_DEPSTARTCFG命令。在命令完成后，它将为所有可用的端点设置传输资源。

传输资源的分配不能完全遵循数据手册，因为控制器驱动程序并没有所有先见之明的配置知识。复合设备框架会分别在每个SET\_CONFIGURATION和SET\_INTERFACE后将此信息分散给它。在此情况下尝试遵循数据手册编程模型可能会导致错误。由于两个原因

数据手册表示，对于每个USB\_REQ\_SET\_CONFIGURATION和USB\_REQ\_SET\_INTERFACE（8.1.5），都要执行DWC3\_DEPCMD\_DEPSTARTCFG。在多个接口的情况下，这是不正确的。数据手册未提及为新端点上的alt设置执行更多的DWC3\_DEPCMD\_DEPXFERCFG（8.1.6）。

使用以下简化方法：

所有硬件端点都可以分配传输资源，并且此设置将保持持久状态，直到核心重置或休眠。因此，每当我们执行“DWC3\_DEPCMD\_DEPSTARTCFG”（0）时，我们可以继续为每个硬件端点执行“DWC3\_DEPCMD\_DEPXFERCFG”。我们保证有与端点一样多的传输资源。每当启用一个端点时，就会调用这个函数，但只有在EP0-out被调用时才会触发，这总是首先发生的，并且只应在上述条件之一下发生。

#### int \_\_dwc3\_gadget\_ep\_enable（struct dwc3\_ep \* dep，unsigned int action）

初始化硬件端点

**参数**

struct dwc3\_ep \* dep

要初始化的端点

unsigned int action

INIT，MODIFY或RESTORE之一

**说明**

调用程序应注意锁定。执行所有必要的命令以初始化HW端点，以便可以由gadget驱动程序使用。

#### int \_\_dwc3\_gadget\_ep\_disable（struct dwc3\_ep \* dep）

禁用hw端点

**参数**

struct dwc3\_ep \* dep

要禁用的端点

**说明**

此函数撤消\_\_dwc3\_gadget\_ep\_enable的操作，并删除当前由硬件处理的请求以及尚未调度的请求。调用程序应注意锁定。

#### struct dwc3\_trb \*dwc3\_ep\_prev\_trb（struct dwc3\_ep \* dep，u8 index）

返回环中的上一个TRB

**参数**

struct dwc3\_ep \* dep

带有TRB环的端点

u8 index

环中当前的TRB的索引

**说明**

返回索引指向的TRB前面的TRB。如果索引为0，我们将向后回卷，跳过链路TRB，并返回它之前的TRB。

#### void dwc3\_prepare\_one\_trb（struct dwc3\_ep \* dep，struct dwc3\_request \* req，unsigned chain，unsigned node）

为一个请求设置一个TRB

**参数**

struct dwc3\_ep \* dep

准备此请求的端点

struct dwc3\_request \* req

dwc3\_request指针

unsigned int chain

这个TRB是否链接到下一个？

unsigned int node

仅适用于等时端点。第一个TRB需要不同类型。

#### void dwc3\_gadget\_setup\_nump（struct dwc3 \* dwc）

计算并初始化DWC3\_DCFG的NUMP字段

**参数**

struct dwc3 \* dwc

指向我们的上下文结构的指针

**说明**

以下看起来很复杂，但实际上非常简单。为了计算我们可以一次在OUT传输中突发的数据包数量，我们将使用RxFIFO大小。

为了计算RxFIFO大小，我们需要两个数字MDWIDTH =内存总线RAM2\_DEPTH的大小，以位为单位，内部RAM2的MDWIDTH（RxFIFO位于其中的深度）

给定这两个数字，公式很简单：

 RxFIFO 大小 = (RAM2\_DEPTH \* MDWIDTH / 8) - 24 - 16；

24字节是3个SETUP数据包

16字节是时钟域过渡容限

给定RxFIFO大小，

#### int dwc3\_gadget\_init（struct dwc3 \* dwc）

初始化与gadget相关的寄存器

**参数**

struct dwc3 \* dwc

指向我们的控制器上下文结构的指针

**说明**

成功返回0，否则返回负的errno。

#### DWC3\_DEFAULT\_AUTOSUSPEND\_DELAY()

DesignWare USB3 DRD控制器Core文件

#### int dwc3\_get\_dr\_mode（struct dwc3 \* dwc）

验证并设置dr\_mode

**参数**

struct dwc3 \* dwc

指向我们的上下文结构的指针

#### int dwc3\_core\_soft\_reset（struct dwc3 \* dwc）

发出核心软重置和PHY重置

**参数**

struct dwc3 \* dwc

指向我们的上下文结构的指针

#### void dwc3\_ref\_clk\_period（struct dwc3 \* dwc）

参考时钟周期配置默认参考时钟周期取决于硬件配置。对于参考时钟与默认值不同的系统，这将在DWC3\_GUCTL寄存器中设置时钟周期。

**参数**

struct dwc3 \* dwc

指向我们的控制器上下文结构的指针

#### void dwc3\_free\_one\_event\_buffer（struct dwc3 \* dwc，struct dwc3\_event\_buffer \* evt）

释放一个事件缓冲区

**参数**

struct dwc3 \* dwc

指向我们的控制器上下文结构的指针

struct dwc3\_event\_buffer \* evt

要释放的事件缓冲区的指针

#### struct dwc3\_event\_buffer \* dwc3\_alloc\_one\_event\_buffer（struct dwc3 \* dwc，unsigned int length）

分配一个事件缓冲区结构

**参数**

struct dwc3 \* dwc

指向我们的控制器上下文结构的指针

unsigned int length

事件缓冲区的大小

**说明**

在成功的情况下返回指向分配的事件缓冲区结构的指针，否则返回ERR\_PTR（errno）。

#### void dwc3\_free\_event\_buffers（struct dwc3 \* dwc）

释放所有已分配的事件缓冲区

**参数**

struct dwc3 \* dwc

指向我们的控制器上下文结构的指针

#### int dwc3\_alloc\_event\_buffers（struct dwc3 \* dwc，unsigned int length）

分配长度为length的num事件缓冲区

**参数**

struct dwc3 \* dwc

指向我们的控制器上下文结构的指针

unsigned int length

事件缓冲区的大小

**说明**

成功返回0，否则返回负的errno。在错误情况下，dwc可能包含已分配但未请求的某些缓冲区。

#### int dwc3\_event\_buffers\_setup（struct dwc3 \*dwc）

设置已分配的事件缓冲区

**参数**

struct dwc3 \*dwc

指向控制器上下文结构的指针

**说明**

成功返回0，否则返回负的错误号。

#### int dwc3\_phy\_setup(struct dwc3 \*dwc)

配置DWC3核的USB PHY接口

**参数**

struct dwc3 \*dwc

指向控制器上下文结构的指针

**说明**

成功返回0。 USB PHY接口已配置但未初始化。 PHY接口和PHY与核心一起在dwc3\_core\_init中初始化。

#### int dwc3\_core\_init(struct dwc3 \*dwc)

DWC3核的低级别初始化

**参数**

struct dwc3 \*dwc

指向控制器上下文结构的指针

**说明**

成功返回0，否则返回负的错误号。

1. （1、2、3）传输请求块
2. 传输请求块指向另一个传输请求块。
3. （1、2）调试文件系统
4. 配置文件系统
5. 命令块包装器

## 编写MUSB胶水层

### 介绍

Linux MUSB子系统是更大的Linux USB子系统的一部分。 它提供了对不使用通用主机控制器接口（UHCI）或开放主机控制器接口（OHCI）的嵌入式USB设备控制器（UDC）的支持。

相反，这些嵌入式UDC依赖于它们至少部分实现的USB On-the-Go（OTG）规范。 在大多数情况下使用的硅参考设计是Mentor Graphics Inventra™设计中找到的多点USB高速双重控制器（MUSB HDRC）。

作为一种自学练习，我编写了Ingenic JZ4740 SoC的MUSB胶水层，模仿内核源树中的许多MUSB胶水层。 此层可以在drivers / usb / musb / jz4740.c中找到。 在本文档中，我将讲解jz4740.c胶水层的基础知识，解释不同的部分以及编写自己的设备胶水层所需的内容。

### Linux MUSB基础知识

要开始学习主题，请阅读USB On-the-Go Basics（参见资源），该资源提供了USB OTG在硬件级别上的操作简介。 德州仪器和模拟器件的一些维基页面还提供了Linux内核MUSB配置的概述，尽管侧重于这些公司提供的一些特定设备。 最后，熟悉USB规范并提供通过编写USB设备驱动程序文档的实际示例可能会有所帮助（再次见资源）。

Linux USB堆栈是一种分层体系结构，在其中MUSB控制器硬件位于最低层。 MUSB控制器驱动程序将MUSB控制器硬件抽象到Linux USB堆栈:

------------------------

| | <------- drivers/usb/gadget

| Linux USB Core Stack | <------- drivers/usb/host

| | <------- drivers/usb/core

------------------------

⬍

--------------------------

| | <------ drivers/usb/musb/musb\_gadget.c

| MUSB Controller driver | <------ drivers/usb/musb/musb\_host.c

| | <------ drivers/usb/musb/musb\_core.c

--------------------------

⬍

---------------------------------

| MUSB Platform Specific Driver |

| | <-- drivers/usb/musb/jz4740.c

| aka "Glue Layer" |

---------------------------------

⬍

---------------------------------

| MUSB Controller Hardware |

---------------------------------

如上所述，胶水层实际上是控制器驱动程序和控制器硬件之间的平台特定代码。

就像Linux USB驱动程序需要向Linux USB子系统注册一样，MUSB胶水层首先需要向MUSB控制器驱动程序注册。这将使控制器驱动程序知道胶水层支持的设备及其检测到或释放支持的设备时调用的哪些函数；请记住，我们在这里谈论的是嵌入式控制器芯片，因此没有运行时插入或拔出。

所有这些信息都通过在胶水层中定义的platform\_driver结构传递给MUSB控制器驱动程序:

static struct platform\_driver jz4740\_driver = {

.probe = jz4740\_probe,

.remove = jz4740\_remove,

.driver = {

.name = "musb-jz4740",

},

};

在检测到匹配的设备时，将调用探测和删除函数指针，分别释放。名称字符串说明此胶水层支持的设备。在当前情况下，它与在arch / mips / jz4740 / platform.c中声明的platform\_device结构匹配。请注意，我们在这里没有使用设备树绑定。

为了向控制器驱动程序注册自己，胶水层需要经过几个步骤，基本上是分配控制器硬件资源并初始化几个电路。为此，它需要跟踪在这些步骤中使用的信息。这是通过定义一个私有的jz4740\_glue结构来完成的:

struct jz4740\_glue {

struct device \*dev;

struct platform\_device \*musb;

struct clk \*clk;

};

dev和musb成员都是设备结构变量。第一个保持有关设备的常规信息，因为它是基本设备结构，而后者保持与注册到的子系统更密切相关的信息。 clk变量保留与设备时钟操作有关的信息。

让我们通过领先胶水层向控制器驱动程序注册自己的探测函数的步骤。

**注意事项**

为了易读性，每个函数将被分为逻辑部分，每个部分都显示为与其他部分独立的部分。

static int jz4740\_probe(struct platform\_device \*pdev){

struct platform\_device \*musb;

struct jz4740\_glue \*glue;

struct clk \*clk;

int ret;

glue = devm\_kzalloc(&pdev->dev, sizeof(\*glue), GFP\_KERNEL);

if (!glue)

return -ENOMEM;

musb=platform\_device\_alloc("musb-hdrc", PLATFORM\_DEVID\_AUTO);

if (!musb) {

dev\_err(&pdev->dev, "failed to allocate musb device\n");

return -ENOMEM;

}

clk = devm\_clk\_get(&pdev->dev, "udc");

if (IS\_ERR(clk)) {

dev\_err(&pdev->dev, "failed to get clock\n");

ret = PTR\_ERR(clk);

goto err\_platform\_device\_put;

}

ret = clk\_prepare\_enable(clk);

if (ret) {

dev\_err(&pdev->dev, "failed to enable clock\n");

goto err\_platform\_device\_put;

}

musb->dev.parent = &pdev->dev;

glue->dev = &pdev->dev;

glue->musb = musb;

glue->clk = clk;

return 0;

err\_platform\_device\_put:

platform\_device\_put(musb);

return ret;

}

probe function的前几行分配并分配了黏合，musb和clk变量。GFP\_KERNEL标志（第8行）允许分配过程休眠并等待内存，因此可用于锁定情况。PLATFORM\_DEVID\_AUTO标志（第12行）允许自动分配和管理设备ID，以避免显式ID与设备命名空间冲突。使用devm\_clk\_get（）（第18行），黏合层分配时钟，并启用它，devm\_前缀表示由clk\_get（）管理当设备释放时，它会自动释放分配的时钟资源数据。

接下来是注册步骤:

static int jz4740\_probe(struct platform\_device \*pdev)

{

struct musb\_hdrc\_platform\_data \*pdata = &jz4740\_musb\_platform\_data;

pdata->platform\_ops = &jz4740\_musb\_ops;

platform\_set\_drvdata(pdev, glue);

ret = platform\_device\_add\_resources(musb, pdev->resource, pdev->num\_resources);

if (ret) {

dev\_err(&pdev->dev, "failed to add resources\n");

goto err\_clk\_disable;

}

ret = platform\_device\_add\_data(musb, pdata, sizeof(\*pdata));

if (ret) {

dev\_err(&pdev->dev, "failed to add platform\_data\n"); goto err\_clk\_disable;

}

return 0;

err\_clk\_disable:

clk\_disable\_unprepare(clk);

err\_platform\_device\_put:

platform\_device\_put(musb);

return ret;

}

第一步是通过platform\_set\_drvdata（）（第7行）将黏合层私有持有的设备数据传递给控制器驱动程序。接下来是通过platform\_device\_add\_resources（）（第9行）传递设备资源信息，同样在那一点上私有持有。

最后是通过platform\_device\_add\_data（）（第16行）将平台特定数据传递给控制器驱动程序（第5行的platform\_ops函数指针在musb\_hdrc\_platform\_data结构中（第3行））。这个函数指针允许MUSB控制器驱动程序知道要调用哪个函数进行设备操作。

static const struct musb\_platform\_ops jz4740\_musb\_ops = {

.init = jz4740\_musb\_init,

.exit = jz4740\_musb\_exit,

};

这里我们只有最小的情况，只有在需要时控制器驱动程序才调用init和exit函数。实际上，JZ4740 MUSB控制器是一个基本控制器，缺少其他控制器中找到的一些功能，否则我们也可能有几个其他函数的指针，例如功率管理函数或在OTG和非OTG模式之间切换的函数。

在注册过程的那一点上，控制器驱动程序实际上调用了init函数:

static int jz4740\_musb\_init(struct musb \*musb)

{

musb->xceiv = usb\_get\_phy(USB\_PHY\_TYPE\_USB2);

if (!musb->xceiv) {

pr\_err("HS UDC: no transceiver configured\n");

return -ENODEV; }

/\* Silicon does not implement ConfigData register.

\* Set dyn\_fifo to avoid reading EP config from hardware.

\*/

musb->dyn\_fifo = true;

musb->isr = jz4740\_musb\_interrupt;

return 0;

}

jz4740\_musb\_init的目的是获取MUSB控制器硬件的变送器驱动程序数据，并像往常一样将其传递给MUSB控制器驱动程序。变送器是负责发送/接收USB数据的控制器硬件内部电路。由于它是OSI模型的物理层的实现，因此变送器也称为PHY。

通过usb\_get\_phy（）获取MUSB PHY驱动程序数据，该函数返回包含驱动程序实例数据的结构的指针。下一对指令（第12行和第14行）用作小故障和设置IRQ处理。故障转移和IRQ处理将在后面讨论。

static int jz4740\_musb\_exit(struct musb \*musb)

{

usb\_put\_phy(musb->xceiv);

return 0;

}

作为init的反向，exit函数在控制器硬件本身即将被释放时释放MUSB PHY驱动程序。

再次注意，由于JZ4740控制器硬件的基本设备功能集，因此在这种情况下，init和exit相对简单。当为更复杂的控制器硬件编写musb黏合层时，您可能需要在这两个函数中进行更多处理。

从init函数返回,MUSB控制器驱动程序跳回到probe函数:

static int jz4740\_probe(struct platform\_device \*pdev)

{

ret = platform\_device\_add(musb);

if (ret) {

dev\_err(&pdev->dev, "failed to register musb device\n");

goto err\_clk\_disable;

}

return 0;

err\_clk\_disable:

clk\_disable\_unprepare(clk);

err\_platform\_device\_put:

platform\_device\_put(musb);

return ret;

}

这是设备注册过程的最后一部分，其中粘合层将控制器硬件设备添加到Linux内核设备层次结构中。在此阶段，有关设备的所有已知信息都被传递给了Linux USB核心堆栈:

static int jz4740\_remove(struct platform\_device \*pdev)

{

struct jz4740\_glue \*glue = platform\_get\_drvdata(pdev);

platform\_device\_unregister(glue->musb); clk\_disable\_unprepare(glue->clk);

return 0;

}

作为探测的对应部分，remove函数注销了MUSB控制器硬件（第5行），并禁用了时钟（第6行），从而使其处于关闭状态。

### 处理IRQs

除了对MUSB控制器硬件进行基本设置和注册外，粘合层还负责处理IRQs:

static irqreturn\_t jz4740\_musb\_interrupt(int irq, void \*\_\_hci)

{

unsigned long flags;

irqreturn\_t retval = IRQ\_NONE;

struct musb \*musb = \_\_hci;

spin\_lock\_irqsave(&musb->lock, flags);

musb->int\_usb = musb\_readb(musb->mregs, MUSB\_INTRUSB); musb->int\_tx = musb\_readw(musb->mregs, MUSB\_INTRTX); musb->int\_rx = musb\_readw(musb->mregs, MUSB\_INTRRX);

/\*

\* The controller is gadget only, the state of the host mode IRQ bits is \* undefined. Mask them to make sure that the musb driver core will \* never see them set

\*/

musb->int\_usb &= MUSB\_INTR\_SUSPEND | MUSB\_INTR\_RESUME | MUSB\_INTR\_RESET | MUSB\_INTR\_SOF;

if (musb->int\_usb || musb->int\_tx || musb->int\_rx)

retval = musb\_interrupt(musb);

spin\_unlock\_irqrestore(&musb->lock, flags);

return retval;

}

在这里，粘合层主要需要读取相关硬件寄存器的值，并将其传递给控制器驱动程序，以便处理触发IRQ的实际事件。

中断处理程序关键部分受到spin\_lock\_irqsave（）和相应的spin\_unlock\_irqrestore（）函数的保护（第7行和第24行），它们防止中断处理程序代码同时由两个不同的线程运行。

接着读取相关的中断寄存器（第9-11行）：

MUSB\_INTRUSB表示当前活动的USB中断，

MUSB\_INTRTX表示当前活动的TX端点中断中有哪些，

MUSB\_INTRRX表示当前活动的TX端点中断中有哪些。

注意，musb\_readb（）用于读取最多8位寄存器，而musb\_readw（）允许我们最多读取16位寄存器。根据您的设备寄存器的大小，还可以使用其他函数。有关更多信息，请参见musb\_io.h。

第18行的指令是JZ4740 USB设备控制器的另一个特殊性，稍后将在设备特殊性中讨论。

尽管这样，粘合层仍需注册IRQ处理程序。请记得在init函数的第14行的指令:

static int jz4740\_musb\_init(struct musb \*musb)

{

musb->isr = jz4740\_musb\_interrupt;

return 0;

}

此指令将指向粘合层IRQ处理程序函数的指针设置为控制器硬件，以便当来自控制器硬件的IRQ时，控制器硬件调用处理程序回调处理程序。中断处理程序现已实现和注册。

### 设备平台数据

为了编写MUSB粘合层，您需要拥有说明控制器硬件硬件能力的一些数据，这称为平台数据。

平台数据是特定于您的硬件的，尽管它可能涵盖广泛的设备，并且通常在arch/目录中找到，具体取决于您的设备架构。

例如，JZ4740 SoC的平台数据可在arch/mips/jz4740/platform.c中找到。在platform.c文件中，每个JZ4740 SoC的设备都通过一组结构说明。

这是覆盖USB设备控制器（UDC）的arch/mips/jz4740/platform.c的一部分:

/\* USB Device Controller \*/

struct platform\_device jz4740\_udc\_xceiv\_device = {

.name = "usb\_phy\_gen\_xceiv",

.id = 0,

};

static struct resource jz4740\_udc\_resources[] = {

[0] = {

.start = JZ4740\_UDC\_BASE\_ADDR,

.end = JZ4740\_UDC\_BASE\_ADDR + 0x10000 - 1,

.flags = IORESOURCE\_MEM,

},

[1]= {

.start = JZ4740\_IRQ\_UDC,

.end = JZ4740\_IRQ\_UDC,

.flags = IORESOURCE\_IRQ,

.name = "mc",

},

};

struct platform\_device jz4740\_udc\_device = {

.name = "musb-jz4740",

.id = -1,

.dev = {

.dma\_mask= &jz4740\_udc\_device.dev.coherent\_dma\_mask,

.coherent\_dma\_mask = DMA\_BIT\_MASK(32),

},

.num\_resources = ARRAY\_SIZE(jz4740\_udc\_resources),

.resource = jz4740\_udc\_resources,

};

jz4740\_udc\_xceiv\_device平台设备结构（第2行）通过名称和ID号说明了UDC转换器。

请注意，此编写时，usb\_phy\_gen\_xceiv是要用于所有转换器的特定名称，这些转换器或者使用参考USB IP内置或自主并且不需要任何PHY编程。您需要设置CONFIG\_NOP\_USB\_XCEIV=y在内核配置中，以使用相应的转换器驱动程序。ID字段可以设置为-1（等同于PLATFORM\_DEVID\_NONE）、-2（等同于PLATFORM\_DEVID\_AUTO）或从0开始，用于此类设备的第一个设备，如果我们想要特定的ID号。

jz4740\_udc\_resources资源结构（第7行）定义UDC寄存器的基址。

第一个数组（第9到11行）定义了UDC寄存器的基本内存地址start指向第一个寄存器内存地址，end指向最后一个寄存器内存地址，而flags成员定义了我们正在处理的资源类型。因此，IORESOURCE\_MEM用于定义寄存器内存地址。第二个数组（第14到17行）定义了UDC IRQ寄存器地址。由于JZ4740 UDC仅有一个IRQ寄存器可用，因此开始和结束都指向相同的地址。 IORESOURCE\_IRQ指示我们正在处理IRQ资源，并且mc名称实际上编码在MUSB核心中，以便控制器驱动程序通过名称查询此IRQ资源。

最后，jz4740\_udc\_device平台设备结构（第21行）说明了UDC本身。

musb-jz4740名称（第22行）定义了用于该设备的MUSB驱动程序；请记住，这实际上是我们在Linux MUSB Basics中在jz4740\_driver平台驱动程序结构中使用的名称。id字段（第23行）设置为-1（等同于PLATFORM\_DEVID\_NONE），因为我们不需要设备IDMUSB控制器驱动程序已设置在Linux MUSB Basics中分配自动ID。在此，我们关心DMA相关信息。 dma\_mask字段（第25行）定义要使用的DMA掩码的宽度，而coherent\_dma\_mask（第26行）具有相同的目的，但是针对alloc\_coherent DMA映射在这两种情况下，我们都使用32位掩码。然后，resource字段（第29行）只是一个指向先前定义的resource结构的指针，而num\_resources字段（第28行）跟踪在resource结构中定义的数组数（在这种情况下，在之前定义了两个资源数组）。

现在，完成了关于arch /级别的UDC平台数据的快速概述，让我们回到在drivers/usb/musb/jz4740.c中特定于MUSB粘合层的平台数据:

static struct musb\_hdrc\_config jz4740\_musb\_config = {

/\* Silicon does not implement USB OTG. \*/

.multipoint = 0,

/\* Max EPs scanned, driver will decide which EP can be used. \*/

.num\_eps = 4,

/\* RAMbits needed to configure EPs from table \*/

.ram\_bits = 9,

.fifo\_cfg = jz4740\_musb\_fifo\_cfg,

.fifo\_cfg\_size = ARRAY\_SIZE(jz4740\_musb\_fifo\_cfg),

};

static struct musb\_hdrc\_platform\_data jz4740\_musb\_platform\_data = {

.mode = MUSB\_PERIPHERAL,

.config = &jz4740\_musb\_config,

};

首先，粘合层配置与控制器驱动程序操作相关的控制器硬件特定方面。这是通过jz4740\_musb\_config musb\_hdrc\_config结构完成的。

定义控制器硬件的OTG能力，将multipoint成员（第3行）设置为0（相当于false），因为JZ4740 UDC不兼容OTG。然后num\_eps（第5行）定义控制器硬件的USB端点数，包括端点0这里我们有3个端点+端点0。接下来是ram\_bits（第7行），它是MUSB控制器硬件RAM地址总线的宽度。当控制器驱动程序无法通过读取相关的控制器硬件寄存器自动配置端点时，需要这些信息。在设备Quirks中将讨论此问题。最后两个字段（第8行和第9行）也与设备Quirks相关fifo\_cfg指向USB端点配置表，fifo\_cfg\_size则跟踪在该配置表中的条目数。稍后在设备Quirks中了解更多。

然后，此配置被嵌入在jz4740\_musb\_platform\_data musb\_hdrc\_platform\_data结构（第11行）中配置是指向配置结构本身的指针，而mode告诉控制器驱动程序，控制器硬件是否可以仅用作MUSB\_HOST、MUSB\_PERIPHERAL或MUSB\_OTG的双模式。

请记住，然后在Linux MUSB Basics中将jz4740\_musb\_platform\_data用于传递平台数据信息的探测功能。

### 设备怪癖

为了完成特定于设备的平台数据，您可能还需要编写一些代码在黏合层中解决某些设备特定限制的问题。这些奇怪之处可能是由于某些硬件错误，或仅仅是由于USB On-the-Go规范的不完整实现而产生的结果。

JZ4740 UDC展示了这样的怪癖，其中一些将在这里讨论，为了洞察力，即使这些在您正在处理的控制器硬件中可能不存在。

首先先回到init函数:

static int jz4740\_musb\_init(struct musb \*musb){

musb->xceiv = usb\_get\_phy(USB\_PHY\_TYPE\_USB2);

if (!musb->xceiv) {

pr\_err("HS UDC: no transceiver configured\n"); return -ENODEV;

}

/\* Silicon does not implement ConfigData register.

\* Set dyn\_fifo to avoid reading EP config from hardware.

\*/

musb->dyn\_fifo = true;

musb->isr = jz4740\_musb\_interrupt;

return 0;

}

第12行的指令帮助MUSB控制器驱动程序解决了控制器硬件缺少用于USB端点配置的寄存器的事实。

没有这些寄存器，控制器驱动程序无法从硬件中读取端点配置，因此我们使用第12行的指令来绕过从硅读取配置，并依靠一个说明端点配置的硬编码表:

static struct musb\_fifo\_cfg jz4740\_musb\_fifo\_cfg[] = {

{ .hw\_ep\_num = 1, .style = FIFO\_TX, .maxpacket = 512, },

{ .hw\_ep\_num = 1, .style = FIFO\_RX, .maxpacket = 512, },

{ .hw\_ep\_num = 2, .style = FIFO\_TX, .maxpacket = 64, },

};

查看上面的配置表，我们可以看到每个端点由三个字段说明hw\_ep\_num是端点号，样式是其方向（FIFO\_TX为控制器驱动程序在控制器硬件中发送数据包，而FIFO\_RX则从硬件接收数据包），而maxpacket定义每个可以在该端点上传输的数据包的最大大小。从表中读取，控制器驱动程序知道端点1可以用于发送和接收512字节的USB数据包（这实际上是一个批量输入/输出端点），而端点2可以用于发送64字节的数据包（这实际上是一个中断端点）。

请注意，此处没有有关端点0的信息每个硅设计都默认实现该端点，并根据USB规范预定义配置。有关端点配置表的更多示例，请参见musb\_core.c。

现在让我们回到中断处理程序功能:

static irqreturn\_t jz4740\_musb\_interrupt(int irq, void \*\_\_hci)

{

unsigned long flags;

irqreturn\_t retval = IRQ\_NONE;

struct musb \*musb = \_\_hci;

spin\_lock\_irqsave(&musb->lock, flags);

musb->int\_usb = musb\_readb(musb->mregs, MUSB\_INTRUSB);

musb->int\_tx = musb\_readw(musb->mregs, MUSB\_INTRTX);

musb->int\_rx = musb\_readw(musb->mregs, MUSB\_INTRRX);

/\*

\* The controller is gadget only, the state of the host mode IRQ bits is

\* undefined. Mask them to make sure that the musb driver core will

\* never see them set

\*/

musb->int\_usb &= MUSB\_INTR\_SUSPEND | MUSB\_INTR\_RESUME | MUSB\_INTR\_RESET | MUSB\_INTR\_SOF;

if (musb->int\_usb || musb->int\_tx || musb->int\_rx)

retval = musb\_interrupt(musb);

spin\_unlock\_irqrestore(&musb->lock, flags);

return retval;

}

第18行的指令是控制器驱动程序解决问题的一种方式，即由于MUSB\_INTRUSB寄存器中缺少用于USB主机模式操作的一些中断位，因此处于未定义的硬件状态，因为此MUSB控制器硬件仅以外围模式使用。因此，胶层层层掩盖了这些缺失的位，以避免通过在从MUSB\_INTRUSB读取的值和实际实现在寄存器中的位之间执行逻辑AND操作来产生寄生中断。

这些只是在JZ4740 USB设备控制器中发现的一些奇怪情况。一些其他情况直接在MUSB核心中处理，因为修复够通用，最终提供了更好的问题处理，以供其他控制器硬件使用。

### 结论

编写Linux MUSB粘合层应该是一项更易于访问的任务，因为此文档尝试展示此练习的内部和外部。JZ4740 USB设备控制器相当简单，希望它的粘合层能为好奇心提供良好的示例。与当前MUSB粘合层一起使用，此文档应提供足够的指导以开始工作。如果出了问题，Linux-USB邮件列表存档是另一个有用的浏览资源。

### 鸣谢

非常感谢Lars-Peter Clausen和Maarten ter Huurne在我编写JZ4740粘合层和帮助我使代码好状态时回答我的问题。我还要感谢Qi-Hardware社区为其快乐的指导和支持。

### 资源

USB首页https://www.usb.org

Linux-USB邮件列表存档https://marc.info/?l=linux-usb

USB On-the-Go基础知识https://www.maximintegrated.com/app-notes/index.mvp/id/1822

编写USB设备驱动程序

德州仪器USB配置Wiki页面http://processors.wiki.ti.com/index.php/Usbgeneralpage

## USB Type-C连接器类

### 介绍

typec类用于以统一方式向用户空间说明系统中的USB Type-C端口。该类旨在仅提供用户空间接口实现，希望它可以在尽可能多的平台上利用。

期望在平台上注册每个USB Type-C端口。在正常情况下，注册将由USB Type-C或PD PHY驱动程序完成，但它可能是用于固件接口的驱动程序（例如UCSI），用于USB PD控制器的驱动程序，甚至用于Thunderbolt3控制器的驱动程序。本文档将使用将注册USB Type-C端口的组件称为“端口驱动程序”。

除了显示功能以外，该类还在其驱动程序支持这些功能的情况下，提供用户空间对端口、伙伴和电缆插头的角色和备用模式的控制。

该类提供了本文档中说明的端口驱动程序的API。这些属性在Documentation / ABI / testing / sysfs-class-typec中说明。

### 用户空间界面

每个端口都将在/sys/class/typec/下呈现为自己的设备。第一个端口将被命名为“port0”，第二个将被命名为“port1”，依此类推。

连接时，合作伙伴还将作为其自己的设备呈现在/sys/class/typec/下。合作伙伴设备的父代始终是其连接的端口。连接到端口“port0”的合作伙伴将被命名为“port0-partner”。设备的完整路径将是/sys/class/typec/port0/port0-partner/。

电缆和其中的两个插头也可以作为自己的设备（可选）呈现在/sys/class/typec/下。连接到端口“port0”的电缆将被命名为port0-cable，SOP Prime端（请参见USB功率传递规范ch.2.4）上的插头将被命名为“port0-plug0”，SOP双重主要端上的插头为“port0-plug1”。电缆的父类始终是端口，电缆插头的父类始终是电缆。

如果端口、伙伴或电缆插头支持备用模式，那么每个支持的备用模式SVID都将有其自己的设备来说明它们。请注意，备用模式设备将不会附加到typec类别上。备用模式的父级将是支持它的设备，例如port0-partner的备用模式将在/sys/class/typec/port0-partner/下呈现。每个支持的模式都将在备用模式设备下的“mode<index>”命名的组中拥有自己的组，例如/sys/class/typec/port0/<alternate mode>/mode1/。进入/退出模式的请求可以使用该组中的“active”属性文件完成。

### 驱动程序API

#### 注册端口

端口驱动程序将使用struct typec\_capability数据结构来说明它们控制的每个Type-C端口，并使用以下API进行注册

**struct typec\_port \*typec\_register\_port（struct device \*parent，const struct typec\_capability \* cap）**

注册USB Type-C端口

1）参数

struct device \*parent

父设备

const struct typec\_capability \* cap

端口说明

2）说明

注册说明为cap的USB Type-C端口的设备。

成功时返回端口句柄或失败时返回ERR\_PTR。

**void typec\_unregister\_port（struct typec\_port \*port）**

取消注册USB Type-C端口

1）参数

struct typec\_port \*port

要取消注册的端口

2）说明

取消使用typec\_register\_port（）创建的设备。

在注册端口时，struct typec\_capability中的prefer\_role成员值得特别注意。如果要注册的端口没有初始角色偏好设置，这意味着该端口默认情况下不执行Try.SNK或Try.SRC，则该成员必须具有VALUEC\_NO\_PREFERRED\_ROLE的值。否则，如果端口默认执行Try.SNK，则成员必须具有TYPEC\_DEVICE的值，并且使用Try.SRC值必须为TYPEC\_HOST。

#### 注册合作伙伴

连接合作伙伴成功后，端口驱动程序需要向类注册合作伙伴。需要在struct typec\_partner\_desc中说明有关合作伙伴的详细信息。类在注册期间复制合作伙伴的详细信息。该类提供了用于注册/取消注册合作伙伴的以下API。

**struct typec\_partner \*typec\_register\_partner（struct typec\_port \*port，struct typec\_partner\_desc \*desc）**

注册USB Type-C伙伴

1）参数

struct typec\_port \*port

连接合作伙伴的USB Type-C端口

struct typec\_partner\_desc \*desc

合作伙伴的说明

2）说明

注册说明为desc的USB Type-C合作伙伴设备。

成功时返回合作伙伴句柄或失败时返回ERR\_PTR。

**void typec\_unregister\_partner（struct typec\_partner \*partn**er）

取消注册USB Type-C合作伙伴

1）参数

struct typec\_partner \*partner

要取消注册的合作伙伴

2）说明

取消使用typec\_register\_partner（）创建的设备。

如果注册成功，则该类将提供一个struct typec\_partner句柄，否则为NULL。

如果合作伙伴能够支持USB电源传递命令的“Discover Identity”，并且端口驱动程序能够显示其结果，则合作伙伴说明符结构应包括对struct usb\_pd\_identity实例的句柄。然后，该类将为合作伙伴设备下的标识创建sysfs目录。可以使用以下API报告“Discover Identity”命令的结果。

**int typec\_partner\_set\_identity（struct typec\_partner \*partner）**

报告“Discover Identity”命令的结果

1）参数

struct typec\_partner \*partner

合作伙伴更新的标识值

2）说明

此例程用于报告“Discover Identity USB电源传递命令”的结果已可用。

#### 注册电缆

成功连接支持USB电源传递结构VDM“Discover Identity”的电缆后，端口驱动程序需要注册电缆和一个或两个插头，这取决于电缆中是否存在CC Double Prime控制器。因此，仅具有SOP Prime通信功能但不具有SOP Double Prime通信功能的电缆应仅注册一个插头。有关SOP通信的更多信息，请阅读最新的USB电源传递规范中有关它的章节。

插头代表其自己的设备。首先注册电缆，然后注册电缆插头。电缆将是插头的父设备。需要在struct typec\_cable\_desc中说明有关电缆的详细信息，在struct typec\_plug\_desc中说明插头的详细信息。该类在注册期间复制详细信息。该类提供用于注册/取消注册电缆及其插头的以下API

**struct typec\_plug \*typec\_register\_plug（struct typec\_cable \*cable，struct typec\_plug\_desc \*desc）**

注册USB Type-C电缆插头

1）参数

struct typec\_cable \*cable

带有插头的USB Type-C电缆

struct typec\_plug\_desc \*desc

电缆插头的说明

2）说明

为desc中说明的USB Type-C电缆插头注册设备。 USB Type-C电缆插头代表具有电子元件的插头，可以响应USB电源传递SOP Prime或SOP Double Prime包。

成功时返回电缆插头的句柄，失败则返回ERR\_PTR。

**void typec\_unregister\_plug（struct typec\_plug \*plug）**

注销USB Type-C电缆插头

1）参数

struct typec\_plug \*plug

要注销的电缆插头

2）说明

注销使用typec\_register\_plug()创建的设备。

**struct typec\_cable \*typec\_register\_cable（struct typec\_port \*port，struct typec\_cable\_desc \*desc）**

注册USB Type-C电缆

1）参数

struct typec\_port \*port

电缆连接的USB Type-C端口

struct typec\_cable\_desc \*desc

电缆的说明

2）说明

注册在desc中说明的USB Type-C电缆设备。电缆将是可选电缆插头设备的父设备。

成功时返回电缆的句柄，失败则返回ERR\_PTR。

**void typec\_typetryc\_unregister\_cable(struct typec\_cable \*cable)**

注销USB Type-C电缆

1）参数

struct typec\_cable \*cable

要注销的电缆

2）说明

注销使用typec\_register\_cable()创建的设备。

如果注册成功，该类将为struct typec\_cable和struct typec\_plug提供句柄；如果注册失败，则返回NULL。

如果电缆具有USB Power Delivery功能，并且端口驱动程序能够显示Discover Identity命令的结果，则电缆说明符结构应包括对struct usb\_pd\_identity实例的句柄。然后，该类将创建标识的sysfs目录，位于电缆设备下。可以使用以下API报告Discover Identity命令的结果

**int typec\_cable\_set\_identity（struct typec\_cable \*cable）**

报告Discover Identity命令的结果

1）参数

struct typec\_cable \*cable

已更新的电缆标识值

2）说明

此例程用于报告Discover Identity USB电源传递命令的结果已经可用。

#### 通知

当伙伴执行角色更变或在连接伙伴或电缆期间默认角色更变时，端口驱动程序必须使用以下API向类报告

**void typec\_set\_data\_role（struct typec\_port \*port，enum typec\_data\_role role）**

报告数据角色更变

1）参数

struct typec\_port \*port

更改角色的USB Type-C端口

enum typec\_data\_role role

新的数据角色

2）说明

端口驱动程序使用此例程报告数据角色更改。

**void typec\_set\_pwr\_role（struct typec\_port \*port，enum typec\_role role）**

报告电源角色更变

1）参数

struct typec\_port \*port

更改角色的USB Type-C端口

enum typec\_role role

新的数据角色

2）说明

端口驱动程序使用此例程报告电源角色更改。

**void typec\_set\_vconn\_role（struct typec\_port \*port，enum typec\_role role）**

报告VCONN源更变

1）参数

struct typec\_port \*port

VCONN角色更改的USB Type-C端口

enum typec\_role role

在端口提供VCONN时为Source，否则为Sink

2）说明

端口驱动程序使用此例程报告VCONN源是否更改。

**void typec\_set\_pwr\_opmode（struct typec\_port \*port，enum typec\_pwr\_opmode opmode）**

报告电源操作模式更变

1）参数

struct typec\_port \*port

模式更改的USB Type-C端口

enum typec\_pwr\_opmode opmode

新的电源操作模式

2）说明

端口驱动程序使用此API报告端口上的电源操作模式更变。模式为USB（默认）、1.5A、3.0A，如USB Type-C规格中所定义，以及使用USB Power Delivery规格中定义的方法协商电源级别时的“USB Power Delivery”。

#### 备用模式

USB Type-C端口、伙伴和电缆插头可能支持备用模式。每个备用模式都会有一个称为SVID的标识符，该标识符是由USB-IF或供应商ID给出的标准ID，并且每个支持的SVID可以有1-6种模式。该类为说明SVID的单个模式提供struct typec\_mode\_desc，而struct typec\_altmode\_desc则是所有支持的模式的容器。

支持备用模式的端口需要使用以下API注册它们支持的每个SVID

**struct typec\_altmode \*typec\_port\_register\_altmode（struct typec\_port \*port，const struct typec\_altmode\_desc \*desc）**

注册USB Type-C端口备用模式

1）参数

struct typec\_port \*port

支持备用模式的USB Type-C端口

const struct typec\_altmode\_desc \*desc

备用模式的说明

2）说明

此例程用于注册端口能够支持的备用模式。

成功时返回备用模式的句柄，失败则返回ERR\_PTR。

如果伙伴或电缆插头响应USB Power Delivery结构化VDM Discover SVIDs消息提供SVID列表，则需要注册每个SVID。

用于伙伴的API

**struct typec\_altmode \*typec\_partner\_register\_altmode（struct typec\_partner \*partner，const struct typec\_altmode\_desc \*desc）**

注册USB Type-C伙伴备用模式

1）参数

struct typec\_partner \*partner

支持备用模式的USB Type-C伙伴

const struct typec\_altmode\_desc \*desc

备用模式说明

2）说明

此例程用于独立注册合作伙伴响应Discover SVIDs命令中列出的每个备用模式。需要在desc中的数组中列出响应Discover Modes命令的SVID的模式。

成功返回备选模式的句柄，失败返回ERR\_PTR。

电缆插头的API

**struct typec\_altmode \*typec\_plug\_register\_altmode（struct typec\_plug \* plug，const struct typec\_altmode\_desc \* desc）**

注册USB Type-C电缆插头备用模式

1）参数

struct typec\_plug \* plug

支持备用模式的USB Type-C电缆插头

const struct typec\_altmode\_desc \* desc

备用模式的说明

2）说明

此例程用于独立注册插头响应Discover SVIDs命令中列出的每个备用模式。需要在desc中的数组中列出插头响应Discover Modes命令中列出的SVID的模式。

成功返回备选模式的句柄，失败返回ERR\_PTR。

因此，端口，合作伙伴和电缆插头将使用各自的功能注册备用模式，但是注册始终会在成功时返回struct typec\_altmode的句柄，或在失败时返回NULL。注销将使用相同的功能完成

**void typec\_unregister\_altmode（struct typec\_altmode \* adev）**

注销备用模式

1）参数

struct typec\_altmode \* adev

要注销的备用模式

2）说明

注销使用typec\_partner\_register\_altmode（），typec\_plug\_register\_altmode（）或typec\_port\_register\_altmode（）创建的设备。

如果合作伙伴或电缆插头进入或退出模式，则端口驱动程序需要使用以下API通知类

**void typec\_altmode\_update\_active（struct typec\_altmode \* adev，bool active）**

报告进入/退出模式

1）参数

struct typec\_altmode \* adev

备用模式的句柄

bool active

模式已进入时为True

2）说明

如果合作伙伴或电缆插头成功执行进入/退出模式命令，则驱动程序使用此例程报告模式的更新状态。

#### 复用器/解复用器开关

USB Type-C连接器可能在其后面有一个或多个复用器/解复用器开关。由于插头可以右面朝上或朝下插入，因此需要一个开关将连接器上的正确数据对从连接器路由到USB控制器。如果支持备用或配件模式，则需要另一个开关，可以将连接器上的引脚路由到除USB以外的其他组件。 USB Type-C Connector Class提供了用于注册这些开关的API。

**int typec\_switch\_register（struct typec\_switch \* sw）**

注册USB Type-C方向开关

1）参数

struct typec\_switch \* sw

USB Type-C方向开关

2）说明

此函数注册可以用于根据电缆插头方向从USB Type-C连接器将正确的数据对路由到USB控制器的开关。 USB Type-C插头可以右面朝上或朝下插入。

**void typec\_switch\_unregister（struct typec\_switch \* sw）**

注销USB Type-C方向开关

1）参数

struct typec\_switch \* sw

USB Type-C方向开关

2）说明

注销使用typec\_switch\_register（）注册的开关。

**int typec\_mux\_register（struct typec\_mux \* mux）**

注册多路复用器路由USB Type-C引脚

1）参数

struct typec\_mux \* mux

USB Type-C连接器多路复用器/解复用器

2）说明

当支持配件/备用模式时，USB Type-C连接器可用于除USB以外的操作的备用模式。对于其中一些模式，需要重新配置连接器上的引脚。此函数注册多路复用器开关，以路由连接器上的引脚。

**void typec\_mux\_unregister（struct typec\_mux \* mux）**

注销复用器开关

1）参数

struct typec\_mux \* mux

USB Type-C连接器多路复用器/解复用器

2）说明

注销使用typec\_mux\_register（）注册的mux。

在大多数情况下，同一个物理复用器将处理方向和模式。但是，由于端口驱动程序负责方向，备用模式驱动程序负责模式，因此始终将两者分成自己的逻辑组件“mux”用于模式，“switch”用于方向。

注册端口时，USB Type-C Connector Class请求端口的复用器和开关。然后，驱动程序可以使用以下API来控制它们

**int typec\_set\_orientation（struct typec\_port \* port，enum typec\_orientation orientation）**

设置USB Type-C电缆插头方向

1）参数

struct typec\_port \* port

USB Type-C端口

enum typec\_orientation orientation

USB Type-C电缆插头方向

2）说明

设置端口的电缆插头方向。

**int typec\_set\_mode（struct typec\_port \* port，int mode）**

设置USB Type-C连接器的操作模式

1）参数

struct typec\_port \* port

USB Type-C连接器

int mode

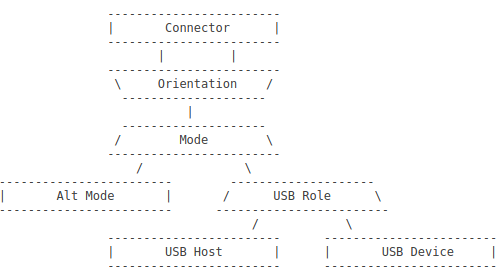
配件模式，USB操作或安全状态

2）说明

为配件模式配置端口。此函数将为模式配置所需的复用器。

如果连接器具有双重角色能力，则可能还有用于数据角色的开关。 USB Type-C Connector Class不提供它们的单独API。端口驱动程序可以与它们一起使用USB Role Class API。

支持替代模式的连接器后面的muxes的示意图



## USB3 调试端口

### 概述

本文介绍了如何在x86系统上使用USB3调试端口。

在使用基于USB3调试端口的任何内核调试功能之前，您需要：

检查系统中是否有可用的USB3调试端口；

检查用于调试目的的端口；

拥有一个USB 3.0超速A到A调试电缆。

### 介绍

xHCI调试功能(DbC)是xHCI主机控制器提供的一个可选但独立的功能。xHCI规范在第7.6节中说明了DbC。

当初始化并启用DbC时，它会通过调试端口(通常是第一个USB3超速端口)提供一个调试设备。调试设备完全符合USB框架，并提供了相当于调试目标(正在调试的系统)和调试主机之间的非常高性能的全双工串行链接的等效物。

### 早期印刷品

DbC已经设计好记录早期的printk消息。这个特性的一个用途是内核调试。例如，当您的机器在常规控制台代码初始化之前崩溃时。其他用途包括简单的、无锁日志记录，而不是一个完整的printk控制台驱动程序和klogd。

在调试目标系统上，您需要使用启用了CONFIG\_EARLY\_PRINTK\_USB\_XDBC的自定义调试内核。然后，添加以下内核引导参数

"earlyprintk=xdbc"

如果系统中有多个xHCI控制器，则可以将主机控制器索引附加到内核参数中。该索引从0开始。

当前的设计不支持DbC运行时暂停/恢复。因此，您最好通过添加以下内核引导参数来禁用USB子系统的运行时电源管理

"usbcore.autosuspend=-1"

在启动调试目标之前，您应该将调试端口连接到调试主机上的一个USB端口(根端口或任何外部集线器端口)。连接这两个端口使用的电缆应该是一个USB 3.0超速A到A调试电缆。

在调试目标的早期引导期间，DbC将被检测并初始化。初始化后，调试主机应该能够enum调试目标中的调试设备。调试主机将随后将调试设备绑定到usb\_debug驱动程序模块，并创建/dev/ttyUSB设备。

如果调试设备enum顺利，那么您应该能够在调试主机上看到以下内核消息:

# tail -f /var/log/kern.log

[ 1815.983374] usb 4-3: new SuperSpeed USB device number 4 using xhci\_hcd

[ 1815.999595] usb 4-3: LPM exit latency is zeroed, disabling LPM.

[ 1815.999899] usb 4-3: New USB device found, idVendor=1d6b, idProduct=0004

[ 1815.999902] usb 4-3: New USB device strings: Mfr=1, Product=2, SerialNumber=3

[ 1815.999903] usb 4-3: Product: Remote GDB

[ 1815.999904] usb 4-3: Manufacturer: Linux

[ 1815.999905] usb 4-3: SerialNumber: 0001

[ 1816.000240] usb\_debug 4-3:1.0: xhci\_dbc converter detected

[ 1816.000360] usb 4-3: xhci\_dbc converter now attached to ttyUSB0

您可以使用任何通信程序，例如minicom，来读取和查看消息。以下简单的bash脚本可以帮助您检查设置的正确性。

===== start of bash scripts =============

#!/bin/bash

while true ; do

while [ ! -d /sys/class/tty/ttyUSB0 ] ; do

:

done

cat /dev/ttyUSB0

done

===== end of bash scripts ===============

### 串行TTY

DbC支持已添加到xHCI驱动程序中。您可以在运行时获取由DbC提供的调试设备。

为了使用它，您需要确保您的内核已配置为支持USB\_XHCI\_DBGCAP。xHCI设备节点下的一个sysfs属性用于启用或禁用DbC。默认情况下，DbC被禁用:

root@target:/sys/bus/pci/devices/0000:00:14.0# cat dbc

disabled

使用以下命令启用DbC:

root@target:/sys/bus/pci/devices/0000:00:14.0# echo enable > dbc

您可以随时检查DbC状态

root@target:/sys/bus/pci/devices/0000:00:14.0# cat dbc

enabled

使用USB 3.0超速A到A调试电缆将调试目标连接到调试主机。您将在调试目标上看到创建/dev/ttyDBC0。您将看到以下内核信息:

root@target: tail -f /var/log/kern.log

[ 182.730103] xhci\_hcd 0000:00:14.0: DbC connected

[ 191.169420] xhci\_hcd 0000:00:14.0: DbC configured

[ 191.169597] xhci\_hcd 0000:00:14.0: DbC now attached to /dev/ttyDBC0

因此，DbC状态已更新为:

root@target:/sys/bus/pci/devices/0000:00:14.0# cat dbc

configured

在调试主机上，您将看到已列举出调试设备。您将看到以下内核消息行:

root@host: tail -f /var/log/kern.log

[ 79.454780] usb 2-2.1: new SuperSpeed USB device number 3 using xhci\_hcd

[ 79.475003] usb 2-2.1: LPM exit latency is zeroed, disabling LPM.

[ 79.475389] usb 2-2.1: New USB device found, idVendor=1d6b, idProduct=0010

[ 79.475390] usb 2-2.1: New USB device strings: Mfr=1, Product=2, SerialNumber=3

[ 79.475391] usb 2-2.1: Product: Linux USB Debug Target

[ 79.475392] usb 2-2.1: Manufacturer: Linux Foundation

[ 79.475393] usb 2-2.1: SerialNumber: 0001

调试设备现在可以正常使用。您可以使用任何通信或调试程序在主机和目标之间进行通讯或调试。

# PCI 支持库

## unsigned char pci\_bus\_max\_busnr(struct pci\_bus \*bus)

返回给定总线子级的最大PCI总线编号

**参数**

struct pci\_bus \*bus

要搜索的PCI总线结构的指针

**说明**

给定一个PCI总线，返回包括给定PCI总线及其子PCI总线列表在内的集合中存在的最高PCI总线号。

## int pci\_find\_capability（struct pci\_dev \*dev，int cap）

查询设备的功能

**参数**

struct pci\_dev \*dev

要查询的PCI设备

int cap

能力代码

**说明**

告诉设备是否支持给定的PCI功能。返回设备的PCI配置空间中所请求的能力结构的地址，或者在设备不支持它的情况下返回0。cap的可能值包括

PCI\_CAP\_ID\_PM 电源管理PCI\_CAP\_ID\_AGP 加速图形端口PCI\_CAP\_ID\_VPD 关键产品数据PCI\_CAP\_ID\_SLOTID 插槽识别PCI\_CAP\_ID\_MSI 消息信号中断

PCI\_CAP\_ID\_CHSWP CompactPCI HotSwapPCI\_CAP\_ID\_PCIX PCI-XPCI\_CAP\_ID\_EXP PCI Express

## int pci\_bus\_find\_capability（struct pci\_bus \*bus，unsigned int devfn，int cap）

查询设备的功能

**参数**

struct pci\_bus \*bus

要查询的PCI总线

unsigned int devfn

要查询的PCI设备

int cap

能力代码

**说明**

与pci\_find\_capability（）类似，但适用于尚未设置pci\_dev结构的PCI设备。

返回设备的PCI配置空间中所请求的能力结构的地址，或者在设备不支持它的情况下返回0。

## int pci\_find\_next\_ext\_capability（struct pci\_dev \*dev，int start，int cap）

查找扩展功能

**参数**

struct pci\_dev \*dev

要查询的PCI设备

int start

开始查找的地址（如果从列表开头开始，则为0）

int cap

能力代码

**说明**

返回设备的PCI配置空间中下一个匹配扩展功能结构的地址，如果设备不支持它，则返回0。某些功能可以多次出现，例如，特定于供应商的功能，这提供了一种找到它们所有的方法。

## int pci\_find\_ext\_capability（struct pci\_dev \*dev，int cap）

查找扩展功能

**参数**

struct pci\_dev \*dev

要查询的PCI设备

int cap

功能代码

**说明**

返回设备的PCI配置空间中所请求的扩展功能结构的地址，或者在设备不支持它的情况下返回0。cap的可能值包括

PCI\_EXT\_CAP\_ID\_ERR高级错误报告PCI\_EXT\_CAP\_ID\_VC虚拟通道PCI\_EXT\_CAP\_ID\_DSN设备序列号PCI\_EXT\_CAP\_ID\_PWR电源预算

## int pci\_find\_next\_ht\_capability（struct pci\_dev \*dev，int pos，int ht\_cap）

查询设备的HyperTransport功能

**参数**

struct pci\_dev \*dev

要查询的PCI设备

int pos

从哪里开始搜索

int ht\_cap

HyperTransport能力代码

**说明**

与pci\_find\_ht\_capability（）结合使用，以搜索匹配ht\_cap的所有功能。 pos应始终是从pci\_find\_ht\_capability（）返回的值。

注意。为了对抗损坏的PCI设备，调用者应采取措施避免无限循环。

## int pci\_find\_ht\_capability（struct pci\_dev \*dev，int ht\_cap）

查询设备的HyperTransport功能

**参数**

struct pci\_dev \*dev

要查询的PCI设备

int ht\_cap

HyperTransport能力代码

**说明**

告知设备是否支持给定的HyperTransport功能。返回设备的PCI配置空间中一个地址，或者在设备不支持请求功能的情况下返回0。地址指向PCI功能，类型为PCI\_CAP\_ID\_HT，具有与ht\_caps匹配的HyperTransport功能。

## struct resource \*pci\_find\_parent\_resource（const struct pci\_dev \*dev，struct resource \*res）

返回给定区域所在的父总线的资源区域

**参数**

const struct pci\_dev \*dev

PCI设备结构包含要搜索的资源

struct resource \*res

要查找父项的子资源记录

**说明**

对于给定设备的给定资源区域，请返回包含给定区域的父总线的资源区域。

## struct resource \*pci\_find\_resource（struct pci\_dev \*dev，struct resource \*res）

返回匹配的PCI设备资源

**参数**

struct pci\_dev \*dev

要查询的PCI设备

struct resource \*res

要查找的资源

**说明**

遍历标准PCI资源（BAR），检查给定的资源是否部分或完全包含在其中。如果是，则返回匹配的资源，否则返回NULL。

## struct pci\_dev \* pci\_find\_pcie\_root\_port（struct pci\_dev \* dev）

返回PCIe根端口

**参数**

struct pci\_dev \* dev

要查询的PCI设备

**说明**

遍历父链，并返回给定PCI设备的PCIe根端口PCI设备。

## int \_\_pci\_complete\_power\_transition(struct pci\_dev \* dev，pci\_power\_t state)

完成PCI设备的电源转换

**参数**

struct pci\_dev \* dev

要处理的PCI设备。

pci\_power\_t state

要将设备放入的状态。

**说明**

设备驱动程序不应直接调用此函数。

## int pci\_set\_power\_state(struct pci\_dev \* dev，pci\_power\_t state)

设置PCI设备的电源状态

**参数**

struct pci\_dev \* dev

要处理的PCI设备。

pci\_power\_t state

要将设备置于的PCI电源状态（D0，D1，D2，D3hot）。

**说明**

使用平台固件和/或设备的PCI PM寄存器将设备转换到新的电源状态。

返回如果请求的状态无效，则为-EINVAL。如果设备不支持PCI PM或其PM功能寄存器具有错误版本，或者设备不支持请求的状态，则为-EIO。如果状态转换为D1或D2但不支持D1和D2，则为0。如果设备已经处于请求的状态，则为0。如果过渡到D3但不支持D3，则为0。如果已成功更改设备的电源状态，则为0。

## pci\_power\_t pci\_choose\_state(struct pci\_dev \* dev，pm\_message\_t state)

选择PCI设备的电源状态

**参数**

struct pci\_dev \* dev

要挂起的PCI设备

pm\_message\_t state

整个系统的目标睡眠状态。这是传递给suspend（）函数的值。

**说明**

返回适合给定设备和给定系统消息的PCI电源状态。

## int pci\_save\_state(struct pci\_dev \* dev)

挂起前保存设备的PCI配置空间

**参数**

struct pci\_dev \* dev

我们正在处理的PCI设备

## void pci\_restore\_state(struct pci\_dev \* dev)

恢复PCI设备的保存状态

**参数**

struct pci\_dev \* dev

我们正在处理的PCI设备

## struct pci\_saved\_state \* pci\_store\_saved\_state（struct pci\_dev \* dev）

分配并返回包含设备保存状态的不透明结构。

**参数**

struct pci\_dev \* dev

我们正在处理的PCI设备

**说明**

如果没有状态或错误，则返回NULL。

## int pci\_load\_saved\_state（struct pci\_dev \* dev，struct pci\_saved\_state \* state）

重新加载提供的保存状态到结构pci\_dev中。

**参数**

struct pci\_dev \* dev

我们正在处理的PCI设备

struct pci\_saved\_state \* state

从pci\_store\_saved\_state（）返回的保存状态

## int pci\_load\_and\_free\_saved\_state（struct pci\_dev \* dev，struct pci\_saved\_state \*\* state）

重新加载由state指向的保存状态，并释放为其分配的内存。

**参数**

struct pci\_dev \* dev

我们正在处理的PCI设备

struct pci\_saved\_state \*\* state

从pci\_store\_saved\_state（）返回的指针保存状态

## int pci\_reenable\_device(struct pci\_dev \* dev)

恢复废弃的设备

**参数**

struct pci\_dev \* dev

要恢复的PCI设备

注意

此功能是pci\_default\_resume（）的后端，不应由正常代码调用，请编写适当的恢复处理程序并使用它。

## int pci\_enable\_device\_io(struct pci\_dev \* dev)

使用IO空间初始化设备

**参数**

struct pci\_dev \* dev

要初始化的PCI设备

**说明**

在驱动程序使用设备之前对其进行初始化。要求低级代码启用I / O资源。如果设备已处于挂起状态，请将其唤醒。请注意，此功能可能会失败。

## int pci\_enable\_device\_mem(struct pci\_dev \* dev)

初始化用于Memory空间的设备

**参数**

struct pci\_dev \* dev

要初始化的PCI设备

**说明**

在驱动程序使用设备之前对其进行初始化。要求低级代码启用Memory资源。如果设备已处于挂起状态，请将其唤醒。请注意，此功能可能会失败。

## int pci\_enable\_device(struct pci\_dev \* dev)

在驱动程序使用设备之前对其进行初始化。

**参数**

struct pci\_dev \* dev

要初始化的PCI设备

**说明**

在驱动程序使用设备之前对其进行初始化。要求低级代码启用I / O和Memory。如果设备已处于挂起状态，请将其唤醒。请注意，此功能可能会失败。

请注意，如果我们重复调用此函数，实际上不会多次启用设备（我们只会增加计数）。

## int pcim\_enable\_device(struct pci\_dev \* pdev)

管理pci\_enable\_device()

**参数**

struct pci\_dev \* pdev

要初始化的PCI设备

**说明**

管理pci\_enable\_device（）。

## void pcim\_pin\_device(struct pci\_dev \* pdev)

引脚管理的PCI设备

**参数**

struct pci\_dev \* pdev

要固定的PCI设备

**说明**

固定受管PCI设备pdev。 固定的设备不会在驱动程序分离时被禁用。 pdev必须已使用pcim\_enable\_device（）启用。

## void pci\_disable\_device(struct pci\_dev \*dev)

在使用完PCI设备后禁用它

**参数**

struct pci\_dev \*dev

要禁用的PCI设备

**说明**

向系统发出信号，表明PCI设备不再被系统使用。这只涉及到禁用PCI总线主控（如果已激活）。

请注意，直到所有的pci\_enable\_device()调用者都调用pci\_disable\_device()，我们才会真正地禁用设备。

## int pci\_set\_pcie\_reset\_state(struct pci\_dev \*dev, enum pcie\_reset\_state state)

为设备dev设置复位状态

**参数**

struct pci\_dev \*dev

要复位的PCIe设备

enum pcie\_reset\_state state

要进入的复位状态

**说明**

设置设备的PCIe复位状态。

## bool pci\_pme\_capable(struct pci\_dev \*dev, pci\_power\_t state)

检查PCI设备生成可屏蔽中断(PME#)的能力。

**参数**

struct pci\_dev \*dev

要处理的PCI设备。

pci\_power\_t state

设备将发出PME#的PCI状态。

## void pci\_pme\_active(struct pci\_dev \*dev, bool enable)

启用或禁用PCI设备的PME#功能

**参数**

struct pci\_dev \*dev

要处理的PCI设备。

bool enable

“true”以启用PME#生成；“false”为禁用它。

**说明**

在调用此函数之前，调用者必须验证设备是否能够生成PME#，并使enable等于‘true’。

## int pci\_enable\_wake(struct pci\_dev \*pci\_dev, pci\_power\_t state, bool enable)

更改PCI设备的唤醒设置

**参数**

struct pci\_dev \*pci\_dev

目标设备

pci\_power\_t state

设备将发出唤醒事件的PCI状态

bool enable

是否启用事件生成

**说明**

如果启用，则在对其调用\_\_pci\_enable\_wake()之前，检查设备的device\_may\_wakeup()。

## int pci\_wake\_from\_d3(struct pci\_dev \*dev, bool enable)

启用/禁用设备从D3\_hot或D3\_cold唤醒

**参数**

struct pci\_dev \*dev

要准备的PCI设备

bool enable

“True”以启用唤醒事件生成；“False”为禁用。

**说明**

许多驱动程序希望设备从D3\_hot或D3\_cold唤醒系统，此函数允许它们清理地设置 - 不应连续两次调用pci\_enable\_wake()以启用PCI PM vs ACPI排序约束的唤醒。

如果设备不允许从睡眠中唤醒系统，或者它不能从D3\_hot和D3\_cold同时生成PME#，并且平台无法为其启用唤醒电源，则此函数仅返回错误代码。

## int pci\_prepare\_to\_sleep(struct pci\_dev \*dev)

为进入睡眠状态的系统为PCI设备做准备

**参数**

struct pci\_dev \*dev

要处理的设备。

**说明**

根据设备是否能够唤醒系统和/或由平台管理（PCI\_D3hot是默认值），选择适当的电源状态，并将设备放入该状态。

## int pci\_back\_from\_sleep(struct pci\_dev \*dev)

在系统进入工作状态的过程中打开PCI设备

**参数**

struct pci\_dev \*dev

要处理的设备。

**说明**

禁用设备的系统唤醒功能，并将其放入D0。

## bool pci\_dev\_run\_wake(struct pci\_dev \*dev)

检查设备是否能够生成运行时唤醒事件。

**参数**

struct pci\_dev \*dev

要检查的设备。

**说明**

如果设备本身能够通过平台或使用本机PCIe PME生成唤醒事件，或者如果设备支持PME并且其上游桥能够生成唤醒事件，则返回真。

## void pci\_d3cold\_enable(struct pci\_dev \*dev)

为设备启用D3cold

**参数**

struct pci\_dev \*dev

要处理的PCI设备

**说明**

可以在驱动程序中使用此功能，以从处理的设备启用D3cold。 它还相应地更新上游PCI桥PM功能。

## void pci\_d3cold\_disable(struct pci\_dev \*dev)

为设备禁用D3cold

**参数**

struct pci\_dev \*dev

要处理的PCI设备

**说明**

可以在驱动程序中使用这个函数来禁用来自他们处理的设备的D3cold。 它还相应地更新上游PCI桥PM功能。

## int pci\_enable\_atomic\_ops\_to\_root(struct pci\_dev \*dev, u32 cap\_mask)

启用原子操作请求到根端口

**参数**

struct pci\_dev \*dev

PCI设备

u32 cap\_mask

所需原子操作大小掩码，包括一个或多个PCI\_EXP\_DEVCAP2\_ATOMIC\_COMP32、PCI\_EXP\_DEVCAP2\_ATOMIC\_COMP64 PCI\_EXP\_DEVCAP2\_ATOMIC\_COMP128

**说明**

如果所有上游桥都支持原子操作路由、所有上游端口的出口阻塞都被禁用，并且根端口支持所请求的完成能力（32位、64位和/或128位AtomicOp完成），则返回0；否则返回负数。

## u8 pci\_common\_swizzle(struct pci\_dev \*dev, u8 \*pinp)

将INTx全程打上特定的数据扰动

**参数**

struct pci\_dev \*dev

PCI设备

u8 \*pinp

指向INTx引脚值的指针（1 = INTA，2 = INTB，3 = INTD，4 = INTD）

**说明**

为设备执行INTx扰动。 这会穿过所有PCI到PCI桥，直到PCI根总线。

## void pci\_release\_region(struct pci\_dev \*pdev, int bar)

释放PCI某个编号区域

**参数**

struct pci\_dev \*pdev

之前通过pci\_request\_region()保留资源的PCI设备

整型bar

要释放的BAR

**说明**

释放之前通过成功调用pci\_request\_region()保留的PCI I/O和内存资源。仅在PCI区域的所有使用已停止后调用此函数。

## int pci\_request\_region(struct pci\_dev \*pdev，int bar，const char \*res\_name)

保留PCI I/O和内存资源

**参数**

struct pci\_dev \*pdev

要保留资源的PCI设备

int bar

要保留的BAR

const char \*res\_name

与资源关联的名称

**说明**

将与PCI设备pdev BAR bar关联的PCI区域标记为资源名称所有者所拥有。除非此调用成功返回，否则不要访问PCI区域内的任何地址。

成功返回0或错误时返回EBUSY。失败时还会打印警告消息。

独占性的主要区别在于，用户空间明确不允许通过/dev/mem或sysfs映射资源。

## void pci\_release\_selected\_regions(struct pci\_dev \*pdev，int bars)

释放选定的PCI I/O和内存资源

**参数**

struct pci\_dev \*pdev

以前预留资源的PCI设备

int bars

要释放的BAR的位掩码

**说明**

释放以前保留的选定PCI I/O和内存资源。仅在PCI区域的所有使用已停止后调用此函数。

## int pci\_request\_selected\_regions(struct pci\_dev \*pdev，int bars，const char \*res\_name)

保留选定的PCI I/O和内存资源

**参数**

struct pci\_dev \*pdev

要保留资源的PCI设备

int bars

要请求的BAR的位掩码

const char \*res\_name

与资源关联的名称

## void pci\_release\_regions(struct pci\_dev \*pdev)

释放以前通过pci\_request\_regions()成功保留的PCI I/O和内存资源

**参数**

struct pci\_dev \*pdev

以前预留资源的PCI设备

**说明**

释放以前通过pci\_request\_regions()成功保留的所有PCI I/O和内存资源。仅在PCI区域的所有使用已停止后调用此函数。

## int pci\_request\_regions(struct pci\_dev \*pdev，const char \*res\_name)

保留PCI I/O和内存资源

**参数**

struct pci\_dev \*pdev

要保留资源的PCI设备

const char \*res\_name

与资源相关联的名称。

**说明**

将PCI设备pdev的所有PCI区域标记为资源名称所有者所拥有。除非此调用成功返回，否则不要访问PCI区域内的任何地址。

成功返回0或错误时返回EBUSY。失败时还会打印警告消息。

## int pci\_request\_regions\_exclusive(struct pci\_dev \*pdev，const char \*res\_name)

保留PCI I/O和内存资源

**参数**

struct pci\_dev \*pdev

要保留资源的PCI设备

const char \*res\_name

与资源相关联的名称。

**说明**

将PCI设备pdev的所有PCI区域标记为资源名称所有者所拥有。除非此调用成功返回，否则不要访问PCI区域内的任何地址。

pci\_request\_regions\_exclusive()会标记该区域，使/dev/mem和sysfs MMIO访问不被允许。

成功返回0或错误时返回EBUSY。失败时还会打印警告消息。

## int pci\_remap\_iospace(const struct resource \*res，phys\_addr\_t phys\_addr)

重新映射内存映射的I/O空间

**参数**

const struct resource \*res

**说明**I/O空间的资源

phys\_addr\_t phys\_addr

要映射的范围的物理地址

**说明**

将由res和CPU物理地址phys\_addr说明的内存映射的I/O空间重新映射到虚拟地址空间。只有定义了内存映射IO函数（和定义了PCI\_IOBASE值）的架构才应调用此函数。

## void pci\_unmap\_iospace(struct resource \*res)

取消映射内存映射的I/O空间

**参数**

struct resource \*res

要取消映射的资源

**说明**

从虚拟地址空间中取消映射CPU虚拟地址res。只有定义了内存映射IO函数（和定义了PCI\_IOBASE值）的架构才应调用此函数。

## int devm\_pci\_remap\_iospace(struct device \*dev，const struct resource \*res，phys\_addr\_t phys\_addr)

托管pci\_remap\_iospace()

**参数**

struct device \*dev

要为之重新映射IO地址的通用设备

const struct resource \*res

说明I/O空间的资源

phys\_addr\_t phys\_addr

要映射的范围的物理地址

**说明**

托管pci\_remap\_iospace()。在驱动程序分离时自动取消映射。

## void \_\_iomem \* devm\_pci\_remap\_cfgspace(struct device \*dev，resource\_size\_t offset，resource\_size\_t size)

托管pci\_remap\_cfgspace()

**参数**

struct device \*dev

要为之重新映射IO地址的通用设备

resource\_size\_t offset

要映射的资源地址

resource\_size\_t size

映射大小

**说明**

托管pci\_remap\_cfgspace()。在驱动程序分离时自动取消映射。

void \_\_iomem \* devm\_pci\_remap\_cfg\_resource(struct device \*dev，struct resource \*res)

检查、请求区域和ioremap cfg资源

**参数**

struct device \*dev

要处理资源的通用设备

struct resource \*res

要处理的配置空间资源

**说明**

检查资源是否为有效的内存区域，请求内存区域并使用pci\_remap\_cfgspace() API ioremaps，该API确保保证了正确的PCI配置空间内存属性。

所有操作都由驱动程序管理，当驱动程序解除附着时将被撤消。

返回重新映射内存的指针或一个编码错误代码的ERR\_PTR()。用法示例：

res = platform\_get\_resource(pdev, IORESOURCE\_MEM, 0);

base = devm\_pci\_remap\_cfg\_resource(&pdev->dev, res);

if (IS\_ERR(base))

return PTR\_ERR(base);

## void pci\_set\_master(struct pci\_dev \*dev)

启用设备dev的总线主控

**参数**

struct pci\_dev \*dev

要启用的PCI设备

**说明**

在设备上启用总线主控，并调用pcibios\_set\_master()执行所需的特定于体系结构的设置。

## void pci\_clear\_master(struct pci\_dev \*dev)

禁用设备dev的总线主控

**参数**

struct pci\_dev \*dev

要禁用的PCI设备

## int pci\_set\_cacheline\_size(struct pci\_dev \*dev)

确保CACHE\_LINE\_SIZE寄存器已编程

**参数**

struct pci\_dev \*dev

需要启用MWI的PCI设备

**说明**

pci\_set\_mwi的辅助功能。最初从drivers/net/acenic.c中复制。版权所有1998-2001年Jes Sorensen，<jes\*\*trained\*\*-monkey.org>。

**返回**

出现错误时相应的-ERRNO错误值，成功时为零。

## int pci\_set\_mwi(struct pci\_dev \*dev)

启用内存写入使PCI事务失效

**参数**

struct pci\_dev \*dev

要启用MWI的PCI设备

**说明**

在PCI\_COMMAND中启用内存写入使事务失效。

**返回**

出现错误时相应的-ERRNO错误值，成功时为零。

## int pcim\_set\_mwi(struct pci\_dev \*dev)

设备管理的pci\_set\_mwi()

**参数**

struct pci\_dev \*dev

要启用MWI的PCI设备

**说明**

受管pci\_set\_mwi()。

**返回**

出现错误时相应的-ERRNO错误值，成功时为零。

## int pci\_try\_set\_mwi(struct pci\_dev \*dev)

启用内存写入使PCI事务失效

**参数**

struct pci\_dev \*dev

要启用MWI的PCI设备

**说明**

在PCI\_COMMAND中启用内存写入使事务失效。调用者无需检查返回。

**返回**

出现错误时相应的-ERRNO错误值，成功时为零。

## void pci\_clear\_mwi(struct pci\_dev \*dev)

禁用设备dev的内存写入使事务失效

**参数**

struct pci\_dev \*dev

要禁用的PCI设备

**说明**

在设备上禁用PCI Memory-Write-Invalidate事务

## void pci\_intx(struct pci\_dev \*pdev, int enable)

启用/禁用设备dev的PCI INTx

**参数**

struct pci\_dev \*pdev

要操作的PCI设备

int enable

布尔值是否启用或禁用PCI INTx

**说明**

启用/禁用设备pdev的PCI INTx

## bool pci\_check\_and\_mask\_intx(struct pci\_dev \*dev)

屏蔽待处理中断上的INTx

**参数**

struct pci\_dev \*dev

要操作的PCI设备

**说明**

检查设备dev是否已断开其INTx线路，如果是，则屏蔽该线路并返回true。如果没有挂起的中断，则返回false。

## bool pci\_check\_and\_unmask\_intx(struct pci\_dev \*dev)

如果没有挂起的中断，请解除屏蔽INTx

**参数**

struct pci\_dev \*dev

要操作的PCI设备

**说明**

检查设备dev是否已断开其INTx线路，如果没有，则解除该线路的屏蔽并返回true。如果仍然有中断挂起，则返回false并保留掩码。

## int pci\_wait\_for\_pending\_transaction(struct pci\_dev \*dev)

等待挂起的事务

**参数**

struct pci\_dev \*dev

要操作的PCI设备

**说明**

如果事务挂起，则返回0；否则返回1。

## bool pcie\_has\_flr(struct pci\_dev \* dev)

检查设备是否支持功能级别重置

**参数**

struct pci\_dev \* dev

检查设备

**描述**

如果设备宣传支持 PCIe 功能级别重置，则返回 true。

## int pcie\_flr(struct pci\_dev \*dev)

启动PCIe功能级别复位

**参数**

struct pci\_dev \*dev

要重置的设备

**说明**

无条件地在dev上启动函数级别复位，而不检查任何标志和DEVCAP

### int pci\_bridge\_secondary\_bus\_reset(struct pci\_dev \*dev)

重置PCI桥上的次要总线。

**参数**

struct pci\_dev \*dev

桥设备

**说明**

使用桥控制寄存器在次要总线上断开重置。次要总线上的设备处于通电状态。

### int \_\_pci\_reset\_function\_locked(struct pci\_dev \*dev)

在持有dev互斥锁的情况下重置PCI设备函数。

**参数**

struct pci\_dev \*dev

要重置的PCI设备

**说明**

某些设备允许单独重置一个函数，而不影响同一设备中的其他函数。为了使用此函数，PCI设备必须对PCI配置空间做出响应。

在调用此函数时，设备函数被假定未使用，并且调用者正在持有设备互斥锁。

重置设备将使PCI配置空间的内容随机，因此调用该程序的任何调用程序都必须准备重新初始化设备，包括MSI、总线主控、BAR、解码IO和内存空间等。

如果设备不支持重置单个函数，则返回0，否则返回负数。

## int pci\_reset\_function(struct pci\_dev \*dev)

静置并重置PCI设备函数

**参数**

struct pci\_dev \*dev

要重置的PCI设备

**说明**

某些设备允许单个函数被重置，而不影响同一设备中的其他函数。为了使用此函数，PCI设备必须对PCI配置空间做出响应。

此函数不仅重置设备的PCI部分，还清除与设备关联的所有状态。该函数与\_\_pci\_reset\_function\_locked()的区别在于它保存并恢复设备状态，并使用PCI设备锁。

如果设备函数成功重置，则返回0，否则返回负值，如果设备不支持重置单个函数。

## int pci\_reset\_function\_locked(struct pci\_dev \*dev)

暂停和重置PCI设备函数

**参数**

struct pci\_dev \*dev

要重置的PCI设备

**说明**

某些设备允许单独重置一个函数，而不影响同一设备中的其他函数。PCI设备必须响应PCI配置空间，才能使用此功能。

此函数不仅重置设备的PCI部分，还清除与设备关联的所有状态。这个函数与\_\_pci\_reset\_function\_locked()不同的是它在重置时保存并恢复设备状态。它也不同于pci\_reset\_function()，因为它需要持有PCI设备锁。

如果设备函数成功重置，则返回0，否则返回负值，如果设备不支持重置单个函数。

## int pci\_try\_reset\_function(struct pci\_dev \*dev)

暂停和重置PCI设备函数

**参数**

struct pci\_dev \*dev

要重置的PCI设备

**说明**

与上面相同，只是如果无法锁定设备，则返回-EAGAIN。

## int pci\_probe\_reset\_slot(struct pci\_slot \*slot)

探测PCI插槽是否可重置

**参数**

struct pci\_slot \*slot

要探测的PCI插槽

**说明**

如果可以重置插槽，则返回0；否则返回负值。

## int pci\_probe\_reset\_bus(struct pci\_bus \*bus)

探测PCI总线是否可重置

**参数**

struct pci\_bus \*bus

要探测的PCI总线

**说明**

如果可以重置总线，则返回0；否则返回负值。

## int pci\_reset\_bus(struct pci\_dev \*pdev)

尝试重置PCI总线

**参数**

struct pci\_dev \*pdev

通过插槽/总线重置的顶级PCI设备

**说明**

与上面相同，只是如果无法锁定总线，则返回-EAGAIN。

## int pcix\_get\_max\_mmrbc(struct pci\_dev \*dev)

获取PCI-X的最大设计内存读取字节计数

**参数**

struct pci\_dev \*dev

要查询的PCI设备

**说明**

返回最大设计内存读取字节计数（以字节为单位）或适当的错误值。

## int pcix\_get\_mmrbc(struct pci\_dev \*dev)

获取PCI-X的最大内存读取字节计数

**参数**

struct pci\_dev \*dev

要查询的PCI设备

**说明**

返回最大内存读取字节计数（以字节为单位）或适当的错误值。

## int pcix\_set\_mmrbc(struct pci\_dev \*dev, int mmrbc)

设置PCI-X的最大内存读取字节计数

**参数**

struct pci\_dev \*dev

要查询的PCI设备

int mmrbc

以字节为单位的最大内存读取计数，有效值为512、1024、2048、4096。

**说明**

如果可能，设置最大内存读取字节计数，某些桥接器存在错误，无法实现此功能。

## int pcie\_get\_readrq(struct pci\_dev \*dev)

获取PCI Express读请求大小

**参数**

struct pci\_dev \*dev

要查询的PCI设备

**说明**

返回最大内存读取请求（以字节为单位）或适当的错误值。

## int pcie\_set\_readrq(struct pci\_dev \*dev, int rq)

设置PCI Express最大内存读取请求

**参数**

struct pci\_dev \*dev

要查询的PCI设备

int rq

以字节为单位的最大内存读取计数，有效值为128、256、512、1024、2048、4096。

**说明**

如果可能，设置最大内存读取请求（以字节为单位）。

## int pcie\_get\_mps(struct pci\_dev \*dev)

获取PCI Express最大有效负载大小

**参数**

struct pci\_dev \*dev

要查询的PCI设备

**说明**

返回以字节为单位的最大有效负载大小

## int pcie\_set\_mps(struct pci\_dev \*dev, int mps)

设置PCI Express最大有效负载大小

**参数**

struct pci\_dev \*dev

要查询的PCI设备

int mps

最大有效负载大小（以字节为单位），有效值为128、256、512、1024、2048、4096。

**说明**

如果可能，设置最大有效负载大小。

## u32 pcie\_bandwidth\_available(struct pci\_dev \*dev, struct pci\_dev \*\*limiting\_dev, enum pci\_bus\_speed \*speed, enum pcie\_link\_width \*width)

确定PCIe设备和其带宽限制的最小链接设置

**参数**

struct pci\_dev \*dev

要查询的PCI设备

struct pci\_dev \*\*limiting\_dev

存储导致带宽限制的设备

enum pci\_bus\_speed \*speed

存储限制设备的速度

enum pcie\_link\_width \*width

存储限制设备的宽度

**说明**

向上遍历PCI设备链，并找到可用的最小带宽点。返回该点的可用带宽以及（如果提供限制\_dev、速度和宽度指针）该点的信息。返回的带宽以Mb/s为单位，即原始带宽的百万位/秒。

## enum pci\_bus\_speed pcie\_get\_speed\_cap(struct pci\_dev \*dev)

查询PCI设备的链接速度能力

**参数**

struct pci\_dev \*dev

要查询的PCI设备

**说明**

查询PCI设备速度能力。返回设备支持的最大链接速度。

## enum pcie\_link\_width pcie\_get\_width\_cap(struct pci\_dev \*dev)

查询PCI设备的链接宽度能力

**参数**

struct pci\_dev \*dev

要查询的PCI设备

**说明**

查询PCI设备宽度能力。返回设备支持的最大链接宽度。

## void pcie\_print\_link\_status(struct pci\_dev \*dev)

报告PCI设备的链接速度和宽度

**参数**

struct pci\_dev \*dev

要查询的PCI设备

**说明**

报告设备可用的带宽。

## int pci\_select\_bars(struct pci\_dev \*dev, unsigned long flags)

从资源类型制作BAR掩码

**参数**

struct pci\_dev \*dev

制作BAR掩码的PCI设备

unsigned long flags

要选择的资源类型掩码

**说明**

该辅助例程从资源类型中制作bar掩码。

## int pci\_add\_dynid(struct pci\_driver \*drv, unsigned int vendor, unsigned int device, unsigned int subvendor, unsigned int subdevice, unsigned int class, unsigned int class\_mask, unsigned long driver\_data)

向该驱动程序添加新的PCI设备ID并重新检测设备

**参数**

struct pci\_driver \*drv

目标PCI驱动程序

unsigned int vendor

PCI供应商ID

unsigned int device

PCI设备ID

unsigned int subvendor

PCI副厂商ID

unsigned int subdevice

PCI副设备ID

unsigned int class

PCI类

unsigned int class\_mask

PCI类掩码

unsigned long driver\_data

私有驱动程序数据

**说明**

向该驱动程序添加新的动态PCI设备ID，并导致该驱动程序重新检测所有设备。 在调用此函数之前，drv必须已经被注册。

**上下文**

执行GFP\_KERNEL分配。

**返回**

成功返回0，失败返回-errno。

## const struct pci\_device\_id \*pci\_match\_id(const struct pci\_device\_id \*ids, struct pci\_dev \*dev)

查看PCI设备是否与给定的pci\_id表匹配

**参数**

const struct pci\_device\_id \*ids

要搜索的PCI设备ID结构数组

struct pci\_dev \*dev

要匹配的PCI设备结构。

**说明**

由驱动程序用于检查是否有PCI设备在其支持设备列表中。 如果没有匹配项，则返回匹配的pci\_device\_id结构或NULL。

已弃用；请勿使用此功能，因为它将无法捕获驱动程序可能想要检查的任何动态ID。

## int \_\_pci\_register\_driver(struct pci\_driver \*drv, struct module \*owner, const char \*mod\_name)

注册新的pci驱动程序

**参数**

struct pci\_driver \*drv

要注册的驱动程序结构

struct module \*owner

drv的所有者模块

const char \*mod\_name

模块名称字符串

**说明**

将驱动程序结构添加到已注册驱动程序的列表中。 发生错误时返回负值，否则返回0。 如果没有发生错误，则即使在注册期间未声明任何设备，该驱动程序仍然保持注册状态。

## void pci\_unregister\_driver(struct pci\_driver \*drv)

注销PCI驱动程序

**参数**

struct pci\_driver \*drv

要注销的驱动程序结构

**说明**

从已注册的PCI驱动程序列表中删除驱动程序结构，并通过调用其remove()函数为它负责的每个设备进行清理，并将这些设备标记为没有驱动程序。

## struct pci\_driver \*pci\_dev\_driver(const struct pci\_dev \*dev)

获取设备的pci\_driver

**参数**

const struct pci\_dev \*dev

要查询的设备

**说明**

返回适当的pci\_driver结构或NULL，如果没有为设备注册驱动程序。

## struct pci\_dev \*pci\_dev\_get(struct pci\_dev \*dev)

增加PCI设备结构的引用计数

**参数**

struct pci\_dev \*dev

被引用的设备

**说明**

每个对设备的活动引用都应该有一个引用计数。

PCI设备的驱动程序通常应该在其探测()方法中记录这些引用，并通过调用pci\_dev\_put()释放它们，在其断开()方法中。

返回增加了引用计数的设备指针。

## void pci\_dev\_put(struct pci\_dev \*dev)

释放PCI设备结构的使用

**参数**

struct pci\_dev \*dev

已断开的设备

**说明**

当设备的使用者完成对其的使用时必须调用。当设备的最后一个使用者调用此函数时，会释放设备的内存。

## void pci\_stop\_and\_remove\_bus\_device(struct pci\_dev \*dev)

删除PCI设备及其子级

**参数**

struct pci\_dev \*dev

要删除的设备

**说明**

从设备列表中删除PCI设备，通知驱动程序该设备已被删除。 我们还以深度优先的方式删除任何下级总线和子元素。

对于我们删除的每个设备，从设备列表中删除设备结构，删除/proc条目，并通知用户空间(/sbin/hotplug)。

## struct pci\_bus \*pci\_find\_bus(int domain, int busnr)

从给定的域和总线号查找PCI总线

**参数**

int domain

要搜索的PCI域的数目

int busnr

所需的PCI总线号

**说明**

给定PCI总线号和域号，该全局PCI总线列表中的所需PCI总线被定位。 如果找到总线，则返回其数据结构的指针。 如果找不到总线，则返回NULL。

## struct pci\_bus \*pci\_find\_next\_bus(const struct pci\_bus \*from)

开始或继续搜索PCI总线

**参数**

const struct pci\_bus \*from

找到的先前PCI总线，或NULL以开始新的搜索。

**说明**

遍历已知PCI总线的列表。 通过将NULL作为from参数来启动新搜索。 否则，如果from不为NULL，则从全局列表上的下一个设备开始搜索。

## struct pci\_dev \*pci\_get\_slot(struct pci\_bus \*bus, unsigned int devfn)

寻找给定 PCI 插槽的 PCI 设备

**参数**

struct pci\_bus \*bus

所需 PCI 设备所在的 PCI 总线

unsigned int devfn

编码为 PCI 插槽编号以及多功能设备中逻辑设备编号

**说明**

根据给定的 PCI 总线和插槽/功能号，从 PCI 设备列表中找到需要的 PCI 设备。如果找到设备，则增加其引用计数并返回其数据结构的指针。调用者必须通过调用 pci\_dev\_put() 来减少引用计数。如果没有找到设备，则返回 NULL。

## struct pci\_dev \*pci\_get\_domain\_bus\_and\_slot(int domain, unsigned int bus, unsigned int devfn)

寻找给定 PCI 域（段）、总线和插槽的 PCI 设备

**参数**

int domain

所需的 PCI 设备所在的 PCI 域/段

unsigned int bus

所需的 PCI 设备所在的 PCI 总线

unsigned int devfn

编码为 PCI 插槽编号以及多功能设备中逻辑设备编号

**说明**

根据给定的 PCI 域、总线和插槽/功能号，从 PCI 设备列表中找到需要的 PCI 设备。如果找到设备，则增加其引用计数并返回其数据结构的指针。调用者必须通过调用 pci\_dev\_put() 来减少引用计数。如果没有找到设备，则返回 NULL。

## struct pci\_dev \*pci\_get\_subsys(unsigned int vendor, unsigned int device, unsigned int ss\_vendor, unsigned int ss\_device, struct pci\_dev \*from)

通过卖方/子卖方/设备/子设备 ID 开始或继续搜索 PCI 设备

**参数**

unsigned int vendor

匹配的 PCI 卖方 ID，或 PCI\_ANY\_ID 匹配所有卖方 ID

unsigned int device

匹配的 PCI 设备 ID，或 PCI\_ANY\_ID 匹配所有设备 ID

unsigned int ss\_vendor

匹配的 PCI 子系统卖方 ID，或 PCI\_ANY\_ID 匹配所有子系统卖方 ID

unsigned int ss\_device

匹配的 PCI 子系统设备 ID，或 PCI\_ANY\_ID 匹配所有子系统设备 ID

struct pci\_dev \*from

先前已在搜索中找到的 PCI 设备，或对于新搜索为 NULL。

**说明**

遍历已知的 PCI 设备列表。如果找到具有匹配的销售方、设备、ss\_vendor 和 ss\_device 的 PCI 设备，则增加对该设备的引用计数并返回其设备结构的指针。否则，返回 NULL。通过将 NULL 作为 from 参数传递，初始化新的搜索。否则，如果 from 不为 NULL，则搜索继续从全局列表中的下一个设备开始。如果 from 不为 NULL，则始终将其引用计数减少。

## struct pci\_dev \*pci\_get\_device(unsigned int vendor, unsigned int device, struct pci\_dev \*from)

通过卖方/设备 ID 开始或继续搜索 PCI 设备

**参数**

unsigned int vendor

匹配的 PCI 卖方 ID，或 PCI\_ANY\_ID 匹配所有卖方 ID

unsigned int device

匹配的 PCI 设备 ID，或 PCI\_ANY\_ID 匹配所有设备 ID

struct pci\_dev \*from

先前已在搜索中找到的 PCI 设备，或对于新搜索为 NULL。

**说明**

遍历已知的 PCI 设备列表。如果找到带有匹配的销售方和设备的 PCI 设备，则增加对该设备的引用计数并返回其设备结构的指针。否则，返回 NULL。通过将 NULL 作为 from 参数传递，初始化新的搜索。否则，如果 from 不为 NULL，则搜索继续从全局列表中的下一个设备开始。如果 from 不为 NULL，则始终将其引用计数减少。

## struct pci\_dev \*pci\_get\_class(unsigned int class, struct pci\_dev \*from)

通过类开始或继续搜索 PCI 设备

**参数**

unsigned int class

搜索具有此类别指定的 PCI 设备

struct pci\_dev \*from

先前已在搜索中找到的 PCI 设备，或对于新搜索为 NULL。

说明

遍历已知的 PCI 设备列表。如果找到具有匹配类别的 PCI 设备，则增加对该设备的引用计数并返回其设备结构的指针。否则，返回 NULL。通过将 NULL 作为 from 参数传递，初始化新的搜索。否则，如果 from 不为 NULL，则搜索继续从全局列表中的下一个设备开始。如果 from 不为 NULL，则始终将其引用计数减少。

## int pci\_dev\_present(const struct pci\_device\_id \*ids)

如果存在与设备列表匹配的设备，则返回 1，否则返回 0。

**参数**

const struct pci\_device\_id \*ids

一个指向 null 终止的 struct pci\_device\_id 结构列表的指针，它说明了调用者想要查找的 PCI 设备的类型。

**说明**

显而易见的事实您没有任何设备的引用可以由此函数找到，因此，如果该设备在此函数完成后从系统中移除，该值将过时。使用此函数以查找通常内置于系统中的设备，或者有关另一个设备在特定时刻是否存在的一般提示。

## void pci\_msi\_mask\_irq（struct irq\_data \* data）

用于屏蔽PCI / MSI中断的通用IRQ芯片回调

**参数**

struct irq\_data \* data

指向与该中断相关联的irqdata的指针

## void pci\_msi\_unmask\_irq（struct irq\_data \* data）

用于取消屏蔽PCI / MSI中断的通用IRQ芯片回调

**参数**

struct irq\_data \* data

指向与该中断相关联的irqdata的指针

## int pci\_msi\_vec\_count（struct pci\_dev \* dev）

返回设备可以发送的MSI向量数

**参数**

struct pci\_dev \* dev

要报告的设备

**说明**

此函数通过多消息能力寄存器返回设备请求的MSI向量数。如果该设备无法发送MSI中断，则返回负的errno。否则，调用成功并返回2^n（最大为2^5（32）），根据MSI规范。

## int pci\_msix\_vec\_count（struct pci\_dev \* dev）

返回设备的MSI-X表条目数

**参数**

struct pci\_dev \* dev

指向MSI-X设备函数的pci\_dev数据结构的指针

此函数返回设备的MSI-X表条目数，因此设备能够发送的MSI-X向量数。如果该设备无法发送MSI-X中断，则返回负值errno。

## int pci\_msi\_enabled（void）

是否启用MSI？

**参数**

void

没有参数

**说明**

如果命令行选项pci = nomsi没有禁用MSI，则返回true。

## int pci\_enable\_msix\_range（struct pci\_dev \* dev，struct msix\_entry \* entries，int minvec，int maxvec）

配置设备的MSI-X能力结构

**参数**

struct pci\_dev \* dev

指向MSI-X设备函数的pci\_dev数据结构的指针

struct msix\_entry \* entries

指向MSI-X条目数组的指针

int minvec

请求的最小MSI-X IRQ数

int maxvec

请求的最大MSI-X IRQ数

**说明**

在其软件驱动程序调用以请求在其硬件设备函数上启用MSI-X模式时，使用最大可能的中断数在minvec和maxvec之间的范围内设置设备函数的MSI-X能力结构。如果发生错误，它将返回负的errno。如果成功，则返回分配的中断实际数并指示分配新的分配的MSI-X中断的MSI-X能力结构的成功配置。

## int pci\_alloc\_irq\_vectors\_affinity（struct pci\_dev \* dev，unsigned int min\_vecs，unsigned int max\_vecs，unsigned int flags，struct irq\_affinity \* affd）

为设备分配多个IRQ

**参数**

struct pci\_dev \* dev

要操作的PCI设备

unsigned int min\_vecs

所需向量的最小数量（必须> = 1）

unsigned int max\_vecs

向量的最大（期望）数量

unsigned int flags

分配的标志或缺陷

struct irq\_affinity \* affd

关于亲和力要求的可选说明

**说明**

为dev分配最多max\_vecs个中断向量，如果可用则使用MSI-X或MSI向量，并在无法使用时退回到单个传统向量。成功时返回分配的向量数（可能小于max\_vecs），如果出现错误，则返回负错误代码。如果dev的中断向量少于min\_vecs，则函数将失败并带有-ENOSPC。

要获取用于传递给request\_irq（）的向量使用的Linux IRQ号，请使用pci\_irq\_vector（）助手。

## void pci\_free\_irq\_vectors（struct pci\_dev \* dev）

为设备释放先前分配的IRQ

**参数**

struct pci\_dev \* dev

要操作的PCI设备

**说明**

在pci\_alloc\_irq\_vectors（）中释放分配和启用。

## int pci\_irq\_vector（struct pci\_dev \* dev，unsigned int nr）

返回设备向量的Linux IRQ编号

**参数**

struct pci\_dev \* dev

要操作的PCI设备

unsigned int nr

中断向量索引（从0开始）

## const struct cpumask \* pci\_irq\_get\_affinity（struct pci\_dev \* dev，int nr）

返回特定MSI向量的亲和性

**参数**

struct pci\_dev \* dev

要操作的PCI设备

int nr

设备相关的中断向量索引（从0开始）。

## int pci\_irq\_get\_node（struct pci\_dev \* pdev，int vec）

返回特定MSI向量的NUMA节点

**参数**

struct pci\_dev \* pdev

要操作的PCI设备

int vec

设备相关中断向量索引（从0开始）。

## struct irq\_domain \*pci\_msi\_create\_irq\_domain(struct fwnode\_handle \*fwnode, struct msi\_domain\_info \*info, struct irq\_domain \*parent)

创建MSI中断域

**参数**

struct fwnode\_handle \*fwnode

中断控制器的可选fwnode

struct msi\_domain\_info \*info

MSI域信息

struct irq\_domain \*parent

父irq域

**说明**

更新域和芯片操作，并创建MSI中断域。

**返回**

一个域指针或失败时为空。

## int pci\_bus\_alloc\_resource(struct pci\_bus \*bus, struct resource \*res, resource\_size\_t size, resource\_size\_t align, resource\_size\_t min, unsigned long type\_mask, resource\_size\_t (\*alignf)(void \*, const struct resource \*, resource\_size\_t, resource\_size\_t), void \*alignf\_data)

从父总线中分配资源

**参数**

struct pci\_bus \*bus

PCI总线

struct resource \*res

要分配的资源

resource\_size\_t size

要分配的资源大小

resource\_size\_t align

要分配的资源对齐方式

resource\_size\_t min

分配的最小/proc/iomem地址

unsigned long type\_mask

IORESOURCE\_\*类型标志

resource\_size\_t (\*alignf)(void \*, const struct resource \*, resource\_size\_t, resource\_size\_t)

资源对齐函数

void \*alignf\_data

资源对齐函数的数据参数

**说明**

给定设备所在的PCI总线、大小、最小地址、对齐和类型，尝试为特定设备资源寻找可接受的资源分配。

## void pci\_bus\_add\_device(struct pci\_dev \*dev)

为单个设备启动驱动程序

**参数**

struct pci\_dev \*dev

要添加的设备

**说明**

这将添加sysfs条目并启动设备驱动程序

## void pci\_bus\_add\_devices(const struct pci\_bus \*bus)

启动PCI设备的驱动程序

**参数**

const struct pci\_bus \*bus

检查新设备的总线

**说明**

启动PCI设备的驱动程序并添加一些sysfs条目。

## struct pci\_ops \*pci\_bus\_set\_ops(struct pci\_bus \*bus, struct pci\_ops \*ops)

设置pci总线的原始操作

**参数**

struct pci\_bus \*bus

pci总线结构

struct pci\_ops \*ops

新的原始操作

**说明**

返回以前的原始操作

## void pci\_cfg\_access\_lock(struct pci\_dev \*dev)

锁定PCI配置读/写

**参数**

struct pci\_dev \*dev

pci设备结构

**说明**

当访问被锁定时，任何用户空间读取或写入配置空间和并发锁请求将睡眠，直到再次允许通过pci\_cfg\_access\_unlock()访问。

## bool pci\_cfg\_access\_trylock(struct pci\_dev \*dev)

尝试锁定PCI配置读写

**参数**

struct pci\_dev \*dev

pci设备结构

**说明**

与pci\_cfg\_access\_lock相同，但如果访问已被锁定，则返回0，否则返回1。此函数可从原子上下文中使用。

## void pci\_cfg\_access\_unlock(struct pci\_dev \*dev)

解锁PCI配置读/写

**参数**

struct pci\_dev \*dev

pci设备结构

**说明**

此函数允许PCI配置访问继续。

## enum pci\_lost\_interrupt\_reason pci\_lost\_interrupt(struct pci\_dev \*pdev)

报告丢失的PCI中断

**参数**

struct pci\_dev \*pdev

中断丢失的设备

**说明**

此程序的主要功能是以用户可以识别的标准方式报告丢失的中断（而不是责怪驱动程序）。

**返回**

修复它的建议（虽然驱动程序不必对此采取行动）。

## int pci\_request\_irq(struct pci\_dev \*dev, unsigned int nr, irq\_handler\_t handler, irq\_handler\_t thread\_fn, void \*dev\_id, const char \*fmt, ...)

为PCI设备分配中断线

**参数**

struct pci\_dev \*dev

要操作的PCI设备

unsigned int nr

设备相对中断向量索引（从0开始）。

irq\_handler\_t handler

当发生IRQ时要调用的函数。线程中断的主处理程序。如果为NULL且thread\_fn !=NULL，将安装默认的主处理程序。

irq\_handler\_t thread\_fn

从IRQ处理线程调用的函数，如果为NULL，则不创建IRQ线程

void \*dev\_id

传回处理程序函数的cookie

const char \*fmt

命名处理程序的类似于Printf的格式字符串

...

可变参数

**说明**

此调用分配中断资源并启用中断线和IRQ处理。从调用此函数开始，处理程序和thread\_fn可能会被调用。所有使用此函数请求的中断均可能被共享。

dev\_id不能为空且必须是全局唯一的。

## void pci\_free\_irq(struct pci\_dev \*dev, unsigned int nr, void \*dev\_id)

释放使用pci\_request\_irq分配的中断

**参数**

struct pci\_dev \*dev

要操作的PCI设备

unsigned int nr

设备相对中断向量索引（从0开始）。

void \*dev\_id

要释放的设备标识

**说明**

删除中断处理程序。处理程序被删除，如果中断线不再由任何驱动程序使用，则被禁用。调用者必须确保在调用此函数之前在设备上禁用中断。该函数在IRQ执行此IRQ的所有执行中也不会返回。

不得从中断上下文中调用此函数。

## bool pcie\_relaxed\_ordering\_enabled(struct pci\_dev \*dev)

探测PCIe宽松排序使能

**参数**

struct pci\_dev \*dev

要查询的PCI设备

**说明**

如果设备启用了宽松排序属性，则返回true。

## int pci\_scan\_slot(struct pci\_bus \*bus, int devfn)

扫描PCI总线上的插槽以查找设备

**参数**

struct pci\_bus \*bus

要扫描的PCI总线

int devfn

要扫描的插槽号（必须为零函数）

**说明**

在指定的PCI总线上扫描PCI插槽以查找设备，并将发现的设备添加到总线->设备列表中。新设备将不会有设置is\_added。

返回发现的新设备数量。

## unsigned int pci\_scan\_child\_bus（struct pci\_bus \* bus）

扫描总线下的设备

**参数**

struct pci\_bus \* bus

要扫描设备的总线

**说明**

扫描总线下的设备，包括从属总线。返回包括所有发现设备的新从属编号。

## unsigned int pci\_rescan\_bus（struct pci\_bus \* bus）

扫描PCI总线以查找设备

**参数**

struct pci\_bus \* bus

要扫描的PCI总线

**说明**

扫描PCI总线和子总线以查找新设备，添加并启用它们。

返回发现的从属总线的最大数量。

## struct pci\_slot \* pci\_create\_slot（struct pci\_bus \* parent，int slot\_nr，const char \* name，struct hotplug\_slot \* hotplug）

创建物理PCI插槽或增加引用计数

**参数**

struct pci\_bus \* parent

父桥的pci\_bus

int slot\_nr

PCI\_SLOT（pci\_dev -> devfn）或占位符-1

const char \* name

以/sys/bus/pci/slots/<name>呈现给用户的可见字符串

struct hotplug\_slot \* hotplug

设置为调用方是热插拔驱动程序，否则为NULL

**说明**

PCI插槽具有地址、速度、宽度等一流属性，使用struct pci\_slot来管理它们。此接口将返回一个新的struct pci\_slot给调用者，或者如果pci\_slot已经存在，则其引用计数将被增加。

插槽由pci\_bus、slot\_nr元组唯一标识。

已知存在固件存在问题的平台将同一个名称分配给多个插槽。通过代表调用方为插槽重命名来解决这些错误的平台。如果固件将名称N分配给多个插槽

第一个插槽被分配为N

第二个插槽被分配为N-1

第三个插槽分配为N-2等等。

占位符插槽在大多数情况下，pci\_bus、slot\_nr将足以唯一标识一个插槽。有一个引人注目的例外-pSeries（rpaphp），在其中无法确定slot\_nr，直到实际将设备插入插槽中。在这种情况下，调用者可以为slot\_nr传递-1。

当调用方传递slot\_nr == -1时，强制施加以下语义首先，我们不再检查现有的struct pci\_slot，因为可能有许多slot\_nr为-1的插槽。语义的另一个变化是用户可见的，在sysfs中呈现的“地址”参数仅由ddddbb元组组成，其中dddd是struct pci\_bus的PCI域号，bb是总线号。换句话说，“占位符”插槽的devfn将不会显示。

## void pci\_destroy\_slot（struct pci\_slot \* slot）

减少物理PCI插槽的引用计数

**参数**

struct pci\_slot \* slot

要减少的struct pci\_slot

**说明**

struct pci\_slot进行了引用计数，因此销毁它们非常容易；我们只需在其kobj上调用kobject\_put，让我们的释放方法来完成其余工作。

## void pci\_hp\_create\_module\_link（struct pci\_slot \* pci\_slot）

创建到热插拔驱动程序模块的符号链接

**参数**

struct pci\_slot \* pci\_slot

struct pci\_slot

**说明**

用于pci\_hotplug\_core.c的辅助函数，用于创建到热插拔驱动程序模块的符号链接。

## void pci\_hp\_remove\_module\_link（struct pci\_slot \* pci\_slot）

删除与热插拔驱动程序模块的符号链接。

**参数**

struct pci\_slot \* pci\_slot

struct pci\_slot

**说明**

用于pci\_hotplug\_core.c的辅助函数，用于删除与热插拔驱动程序模块的符号链接。

## int pci\_enable\_rom（struct pci\_dev \* pdev）

启用PCI设备的ROM解码

**参数**

struct pci\_dev \* pdev

要启用的PCI设备

**说明**

在pdev上启用ROM解码。这仅涉及打开PCI ROM BAR的最后一位。请注意，某些卡可能会在ROM和其他资源之间共享地址解码器，因此启用它可能会禁用对MMIO寄存器或其他卡存储器的访问。

## void pci\_disable\_rom（struct pci\_dev \* pdev）

禁用PCI设备的ROM解码

**参数**

struct pci\_dev \* pdev

要禁用的PCI设备

**说明**

通过关闭ROM BAR中的最后一位来禁用PCI设备上的ROM解码。

## void \_\_iomem \* pci\_map\_rom（struct pci\_dev \* pdev，size\_t \* size）

将PCI ROM映射到内核空间

**参数**

struct pci\_dev \* pdev

指向pci设备struct 的指针

size\_t \* size

指向覆盖ROM的PCI窗口大小的指针

**返回**

指向ROM映像的内核虚拟指针

**说明**

将PCI ROM映射到内核空间。如果ROM是引导视频ROM，则将返回阴影BIOS副本而不是实际ROM。

## void pci\_unmap\_rom（struct pci\_dev \* pdev，void \_\_iomem \* rom）

从内核空间中取消映射ROM

**参数**

struct pci\_dev \* pdev

指向pci设备struct 的指针

void \_\_iomem \* rom

以前映射的虚拟地址

**说明**

删除以前映射的ROM的映射

## void \_\_iomem \* pci\_platform\_rom（struct pci\_dev \* pdev，size\_t \* size）

提供平台提供的任何ROM映像的指针

**参数**

struct pci\_dev \* pdev

指针到 PCI 设备struct

大小为 size\_t \* 的指针

指针以接收 ROM 上 PCI 窗口的大小

## int pci\_enable\_sriov(struct pci\_dev \*dev, int nr\_virtfn)

启用 SR-IOV 能力

**参数**

struct pci\_dev \*dev

PCI 设备

int nr\_virtfn

启用的虚拟功能数量

**说明**

成功返回 0，失败返回负数。

## void pci\_disable\_sriov(struct pci\_dev \*dev)

禁用 SR-IOV 能力

**参数**

struct pci\_dev \*dev

PCI 设备

## int pci\_num\_vf(struct pci\_dev \*dev)

返回与 PF 设备关联的 VF 数

**参数**

struct pci\_dev \*dev

PCI 设备

**说明**

返回 VF 的数量，如果未启用 SR-IOV，则返回 0。

## int pci\_vfs\_assigned(struct pci\_dev \*dev)

返回分配给客户端的 VF 数量

**参数**

struct pci\_dev \*dev

PCI 设备

**说明**

返回分配给客户端的 VF 数量。如果设备不是物理功能，则返回 0。

## int pci\_sriov\_set\_totalvfs(struct pci\_dev \*dev, u16 numvfs)

减少可用的 TotalVFs

**参数**

struct pci\_dev \*dev

PCI PF 设备

u16 numvfs

应使用的 TotalVFs 支持的数量

**说明**

应从 PF 驱动程序的探测例程中使用该设备的互斥锁调用。

如果 PF 是 SRIOV-capable 设备并且 numvfs 有效，则返回 0。如果不是 PF，则返回 -ENOSYS；如果 numvfs 无效，则返回 -EINVAL；如果 VFs 已启用，则返回 -EBUSY。

## int pci\_sriov\_get\_totalvfs(struct pci\_dev \*dev)

获取此设备支持的 TotalVFs 总数

**参数**

struct pci\_dev \*dev

PCI PF 设备

**说明**

对于支持 SRIOV 的 PCIe 设备，请返回 TotalVFs 的 PCIe SRIOV 能力值或 driver\_max\_VFs 的值（如果驱动程序将其减小）。否则返回 0。

## int pci\_sriov\_configure\_simple(struct pci\_dev \*dev, int nr\_virtfn)

配置 SR-IOV 的辅助程序

**参数**

struct pci\_dev \*dev

PCI 设备

int nr\_virtfn

要启用的虚拟功能数，0 以禁用

**说明**

启用或禁用 SR-IOV，对于不需要在启用 SR-IOV 之前进行任何 PF 设置的设备。返回在出现错误时为负数，在成功时分配的 VF 数量。

## ssize\_t pci\_read\_legacy\_io(struct file \*filp, struct kobject \*kobj, struct bin\_attribute \*bin\_attr, char \*buf, loff\_t off, size\_t count)

从遗留的 I/O 端口空间读取字节

**参数**

struct file \*filp

打开的 sysfs 文件

struct kobject \*kobj

对应于要从中读取的文件的 kobject

struct bin\_attribute \*bin\_attr

此文件的 struct bin\_attribute

char \*buf

用于存储结果的缓冲区

loff\_t off

偏移量到遗留的 I/O 端口空间

size\_t count

要读取的字节数

**说明**

使用架构特定的回调例程 (pci\_legacy\_read) 从遗留的 I/O 端口空间读取 1、2 或 4 字节。

## ssize\_t pci\_write\_legacy\_io(struct file \*filp, struct kobject \*kobj, struct bin\_attribute \*bin\_attr, char \*buf, loff\_t off, size\_t count)

将字节写入遗留的 I/O 端口空间

**参数**

struct file \*filp

打开的 sysfs 文件

struct kobject \*kobj

对应于要从中写入的文件的 kobject

struct bin\_attribute \*bin\_attr

此文件的 struct bin\_attribute

char \*buf

包含要写入的值的缓冲区

loff\_t off

偏移量到遗留的 I/O 端口空间

size\_t count

要写入的字节数

**说明**

使用架构特定的回调例程（pci\_legacy\_write）从遗留的 I/O 端口空间写入 1、2 或 4 字节。

## int pci\_mmap\_legacy\_mem(struct file \*filp, struct kobject \*kobj, struct bin\_attribute \*attr, struct vm\_area\_struct \*vma)

将遗留的 PCI 内存映射到用户内存空间

**参数**

struct file \*filp

打开的 sysfs 文件

struct kobject \*kobj

要映射的设备对应的 kobject

struct bin\_attribute \*attr

此文件的 struct bin\_attribute

struct vm\_area\_struct \*vma

传递给 mmap 的 struct vm\_area\_struct

**说明**

使用架构特定的回调 pci\_mmap\_legacy\_mem\_page\_range 将遗留内存空间（总线空间的第一个兆字节）映射到应用程序虚拟内存空间中。

## int pci\_mmap\_legacy\_io(struct file \*filp, struct kobject \*kobj, struct bin\_attribute \*attr, struct vm\_area\_struct \*vma)

将遗留的 PCI IO 映射到用户内存空间中

**参数**

struct file \*filp

打开的 sysfs 文件

struct kobject \*kobj

要映射的设备对应的 kobject

struct bin\_attribute \*attr

此文件的 struct bin\_attribute

struct vm\_area\_struct \*vma

传递给 mmap 的 struct vm\_area\_struct

**说明**

使用架构特定的回调 pci\_mmap\_legacy\_io\_page\_range 将遗留的 IO 空间（总线空间的第一个兆字节）映射到应用程序虚拟内存空间中。如果不支持该操作，则返回 -ENOSYS。

## void pci\_adjust\_legacy\_attr(struct pci\_bus \*b, enum pci\_mmap\_state mmap\_type)

调整遗留文件属性

**参数**

struct pci\_bus \*b

要在其下创建文件的总线

enum pci\_mmap\_state mmap\_type

I/O 端口或内存

**说明**

存根实现。如果必要，可以被架构覆盖。

## void pci\_create\_legacy\_files(struct pci\_bus \*b)

创建遗留的 I/O 端口和内存文件

**参数**

struct pci\_bus \*b

要在其下创建文件的总线

**说明**

某些平台允许按总线访问遗留 I/O 端口和 ISA 内存空间。此例程创建文件并将其绑定到 pci-sysfs.c 中的相关读取、写入和 mmap 文件。

在错误展开时，不要将错误传播给调用者，因为即使没有这些文件，设置PCI总线也可以正常运作。

## int pci\_mmap\_resource(struct kobject \*kobj, struct bin\_attribute \*attr, struct vm\_area\_struct \*vma, int write\_combine)

将PCI资源映射到用户内存空间中。

**参数**

struct kobject \*kobj

用于映射的kobject

struct bin\_attribute \*attr

要映射的文件的bin\_attribute

struct vm\_area\_struct \*vma

传递到mmap的vm\_area\_struct

int write\_combine

1表示写组合映射

**说明**

使用常规PCI映射例程将PCI资源映射到用户空间。

## void pci\_remove\_resource\_files(struct pci\_dev \*pdev)

清理资源文件

**参数**

struct pci\_dev \*pdev

要清理的设备

**说明**

如果我们为pdev创建了资源文件，请将它们从sysfs中删除并释放它们的资源。

## int pci\_create\_resource\_files(struct pci\_dev \*pdev)

在sysfs中为设备创建资源文件

**参数**

struct pci\_dev \*pdev

要创建资源文件的设备

**说明**

遍历pdev中的资源，为每个可用资源创建文件。

## ssize\_t pci\_write\_rom(struct file \*filp, struct kobject \*kobj, struct bin\_attribute \*bin\_attr, char \*buf, loff\_t off, size\_t count)

用于启用对PCI ROM显示的访问

**参数**

struct file \*filp

sysfs文件

struct kobject \*kobj

内核对象句柄

struct bin\_attribute \*bin\_attr

此文件的bin\_attribute

char \*buf

用户输入

loff\_t off

文件偏移

size\_t count

输入中的字节数量

**说明**

写入除0以外的任何内容都会启用它

## ssize\_t pci\_read\_rom(struct file \*filp, struct kobject \*kobj, struct bin\_attribute \*bin\_attr, char \*buf, loff\_t off, size\_t count)

读取PCI ROM

**参数**

struct file \*filp

sysfs文件

struct kobject \*kobj

内核对象句柄

struct bin\_attribute \*bin\_attr

此文件的bin\_attribute

char \*buf

要将从ROM中读取的数据放置的位置

loff\_t off

文件偏移

size\_t count

要读取的字节数

**说明**

从与kobj对应的PCI设备中的ROM开始，将count字节从off开始放入buf中。

## void pci\_remove\_sysfs\_dev\_files(struct pci\_dev \*pdev)

清理PCI特定的sysfs文件

**参数**

struct pci\_dev \*pdev

我们应该释放它的设备条目

**说明**

当pdev从sysfs中删除时清除。

# PCI 热插拔支持库

## int \_\_pci\_hp\_register(struct hotplug\_slot \*slot, struct pci\_bus \*bus, int devnr, const char \*name, struct module \*owner, const char \*mod\_name)

使用PCI热插拔子系统注册热插插槽

**参数**

struct hotplug\_slot \*slot

要注册的struct hotplug\_slot的指针

struct pci\_bus \*bus

此插槽所在的总线

int devnr

设备号

const char \*name

在kobject core中注册的名称

struct module \*owner

调用者模块所有者

const char \*mod\_name

调用者模块名称

**说明**

准备一个热插拔槽供内核使用，同时立即将其发布到用户空间。驱动程序可以通过调用pci\_hp\_initialize()和pci\_hp\_add()单独执行这两个步骤。如果成功返回0，否则返回其他。

## int \_\_pci\_hp\_initialize(struct hotplug\_slot \*slot,struct pci\_bus \*bus,int devnr,const char \*name,struct module \*owner,const char \*mod\_name)

为内核使用准备热插槽

**参数**

struct hotplug\_slot \*slot

要初始化的struct hotplug\_slot的指针

struct pci\_bus \*bus

此插槽所在的总线

int devnr

插槽号

const char \*name

在kobject core中注册的名称

struct module \*owner

调用者模块所有者

const char \*mod\_name

调用者模块名称

**说明**

为热插驱动程序分配和填充PCI插槽。调用此后，驱动程序可以调用hotplug\_slot\_name（）以获取插槽的唯一名称。驱动程序必须准备好从此时开始处理->reset\_slot回调。成功返回0，否则返回负int。

## int pci\_hp\_add(struct hotplug\_slot \*slot)

将热插槽发布到用户空间

**参数**

struct hotplug\_slot \*slot

要发布的struct hotplug\_slot的指针

**说明**

使热插插槽的sysfs接口可用，并通过发送uevent将其添加到用户空间。此时，热插驱动程序必须准备好处理所有struct hotplug\_slot\_ops回调。成功返回0，否则返回负int。

## void pci\_hp\_deregister(struct hotplug\_slot \*slot)

从PCI热插拔子系统注销一个热插槽

**参数**

struct hotplug\_slot \*slot

要注销的struct hotplug\_slot的指针

**说明**

该插槽必须先使用调用pci\_hp\_register（）注册到pci热插拔子系统中。如果成功，返回0，否则返回其他。

## void pci\_hp\_del(struct hotplug\_slot \*slot)

从用户空间中删除热插槽

**参数**

struct hotplug\_slot \*slot

要取消发布的struct hotplug\_slot的指针

**说明**

删除热插槽的sysfs接口。成功返回0，否则返回负int。

## void pci\_hp\_destroy(struct hotplug\_slot \*slot)

从内核中删除热插槽

**参数**

struct hotplug\_slot \*slot

要销毁的struct hotplug\_slot的指针

**说明**

销毁热插驱动程序使用的PCI插槽。调用此后，驱动程序不再能够调用hotplug\_slot\_name（）以获取插槽的唯一名称。从此时开始，驱动程序不再需要处理->reset\_slot回调。

成功返回0，错误返回负整数。

## int pci\_hp\_change\_slot\_info(struct hotplug\_slot \* slot, struct hotplug\_slot\_info \* info)

在核心中更改插槽的信息结构

**参数**

struct hotplug\_slot \* slot

指向信息已更改的插槽的指针

struct hotplug\_slot\_info \* info

指向要复制到插槽信息结构的信息的指针

**说明**

插槽必须先用调用 pci\_hp\_register() 的pci热插拔子系统注册。

如果成功则返回0，否则返回其他任何错误。

# 串行外设接口 (SPI)

SPI是“串行外设接口”，在嵌入式系统中被广泛使用，因为它是一个简单而高效的接口基本上是一个多路复用的移位寄存器。它的三个信号线保持一个时钟（SCK，通常在1-20 MHz范围内），一个“主出从入”（MOSI）数据线和一个“主入从出”（MISO）数据线。SPI是一个全双工协议；对于每个从MOSI线移出的比特（每个时钟一个），另一个比特在MISO线上移入。这些比特在从系统内存到达和离开时被组合成不同大小的单词。通常，一个额外的片选线是低电位（nCS）；通常每个外设使用四个信号，有时还会有一个中断。

这里列出的SPI总线设施提供了一个通用接口来声明SPI总线和设备，根据标准的Linux驱动程序模型来管理它们，并执行输入/输出操作。目前，仅支持“主”端接口，其中Linux与SPI外围设备通信，而不实现这样的外围设备本身。 （支持实现SPI从设备的接口必然看起来不同。）

编程接口围绕两种驱动程序和两种设备类型构建。一个“控制器驱动程序”抽象了控制器硬件，它可以很简单，比如一组GPIO引脚，也可以很复杂，比如连接到SPI移位寄存器另一侧的双FIFO和双DMA引擎（最大化吞吐量）。这种驱动程序在它们所在的任何总线（通常是平台总线）和SPI之间建立桥梁，并将其设备的SPI端暴露为一个struct spi\_master。 SPI设备是该主设备的子级，由struct spi\_board\_info说明符制成，这些说明符通常由特定于板的初始化代码提供。struct spi\_driver称为“协议驱动程序”，并使用正常的驱动程序模型调用绑定到spi\_device。

I / O模型是一组排队的消息。协议驱动程序提交一个或多个struct spi\_message对象，这些对象被异步处理和完成。（然而，有同步包装器。）消息是从一个或多个struct spi\_transfer对象构建的，每个对象都包装了一个全双工SPI传输。需要各种协议调整选项，因为不同的芯片对使用与SPI传输的位有非常不同的策略。

## struct spi\_statistics

spi传输的统计信息

**定义**

struct spi\_statistics {

spinlock\_t lock;

unsigned long messages;

unsigned long transfers;

unsigned long errors;

unsigned long timedout;

unsigned long spi\_sync;

unsigned long spi\_sync\_immediate;

unsigned long spi\_async;

unsigned long long bytes;

unsigned long long bytes\_rx;

unsigned long long bytes\_tx;

#define SPI\_STATISTICS\_HISTO\_SIZE 17;

unsigned long transfer\_bytes\_histo[SPI\_STATISTICS\_HISTO\_SIZE];

unsigned long transfers\_split\_maxsize;

};

**成员**

lock

锁保护这个结构

messages

处理的 spi 消息数

transfers

处理的 spi\_transfers 数

errors

spi\_transfer 期间的错误数

timedout

spi\_transfer 期间的超时次数

spi\_sync

使用 spi\_sync 的次数

spi\_sync\_immediate

spi\_sync 在调用上下文中立即执行的次数，无需排队和调度

spi\_async

使用 spi\_async 的次数

bytes

传输到/从设备传输的字节数

bytes\_rx

从设备接收到的字节数

bytes\_tx

发送到设备的字节数

transfer\_bytes\_histo

传输字节直方图

transfers\_split\_maxsize

由于 maxsize 限制而被拆分的传输数

## struct spi\_device

SPI从设备的控制器端代理

**定义**

struct spi\_device {

struct device dev;

struct spi\_controller \*controller;

struct spi\_controller \*master;

u32 max\_speed\_hz;

u8 chip\_select;

u8 bits\_per\_word;

u16 mode;

#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 ;

int irq;

void \*controller\_state;

void \*controller\_data;

char modalias[SPI\_NAME\_SIZE];

int cs\_gpio;

struct spi\_statistics statistics;

};

**成员**

dev

设备的驱动程序模型表示。

controller

与设备一起使用的 SPI 控制器。

master

控制器的副本，用于向后兼容。

max\_speed\_hz

该芯片使用的最大时钟速率（在该板上）；可能会被设备的驱动程序更改。spi\_transfer.speed\_hz 可以为每次传输覆盖它。

chip\_select

Chipselect，区分controller处理的芯片。

bits\_per\_word

数据传输涉及一个或多个字；像 8 位或 12 位这样的字长很常见。内存中的字大小是两个字节的幂（例如，20 位样本使用 32 位）。这可能会被设备的驱动程序更改，或者保留默认值 (0)，表示协议字是八位字节。spi\_transfer.bits\_per\_word 可以为每次传输覆盖它。

mode

spi 模式定义了数据如何按时钟输出和输入。这可能会被设备的驱动程序更改。芯片选择模式的“低电平有效”默认值可以被覆盖（通过指定 SPI\_CS\_HIGH），传输中每个字的“MSB 优先”默认值也可以被覆盖（通过指定 SPI\_LSB\_FIRST）。

irq

request\_irq()负数，或传递给从该设备接收中断的数字。

controller\_state

控制器的运行时状态

controller\_data

控制器的特定于板的定义，例如 FIFO 初始化参数；来自 board\_info.controller\_data

modalias

与此设备一起使用的驱动程序的名称，或该名称的别名。这出现在驱动程序冷插拔的 sysfs“modalias”属性中，以及用于热插拔的 uevents 中

cs\_gpio

芯片选择线的 gpio 编号（可选，不使用 GPIO 线时为 -ENOENT）

statistics

spi\_device 的统计信息

**说明**

spi\_device用于在SPI从设备（通常是离散芯片）和CPU内存之间交换数据。

在dev中，platform\_data用于保存有关此设备的信息，该信息对于设备的协议驱动程序具有意义，但对于其控制器而言则无意义。 一个示例可能是具有稍微不同功能的芯片变体的标识符; 另一个可能是有关此特定板如何连接芯片引脚的信息。

## struct spi\_driver

主机端协议驱动程序

**定义**

struct spi\_driver {

const struct spi\_device\_id \*id\_table;

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

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

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

struct device\_driver driver;

};

**成员**

id\_table

此驱动程序支持的SPI设备列表

probe

将此驱动程序绑定到spi设备。 驱动程序可以验证设备是否实际存在，并可能需要配置特性（例如bits\_per\_word）这是在系统设置期间的初始配置不需要的。

remove

从spi设备中取消绑定此驱动程序。

shutdown

用于系统状态转换（例如电源关闭/停止和kexec）的标准关机回调

drive

SPI设备驱动程序应该初始化此结构的名称和所有者字段。

**说明**

这代表一种设备驱动程序，该驱动程序使用SPI消息与SPI链接的另一端的硬件进行交互。 它被称为“协议”驱动程序，因为它通过消息而不是直接与SPI硬件通信（这是底层SPI控制器驱动程序传递这些消息所做的事情）。 这些协议在驱动程序支持的设备的规范中定义。

通常，那些设备协议表示驱动程序支持的最低级别接口，它也支持上层接口。 此类上层包括像MTD，网络，MMC，RTC，文件系统字符设备节点和硬件监视等框架。

## void spi\_unregister\_driver（struct spi\_driver \* sdrv）

spi\_register\_driver的反向作用

**参数**

struct spi\_driver \* sdrv

要注销的驱动程序

**上下文**

可以睡觉

## module\_spi\_driver

module\_spi\_driver（\_\_spi\_driver）

用于注册SPI驱动程序的辅助宏

**参数**

\_\_spi\_driver

**说明**

用于不在模块init / exit中执行任何特殊操作的SPI驱动程序的辅助宏。 这消除了大量模板代码。 每个模块只能使用此宏一次，并调用它将替换module\_init（）和module\_exit（）。

## struct spi\_controller

与SPI主控制器或从控制器的接口

**定义**

struct spi\_controller {

struct device dev;

struct list\_head list;

s16 bus\_num;

u16 num\_chipselect;

u16 dma\_alignment;

u16 mode\_bits;

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));

u32 min\_speed\_hz;

u32 max\_speed\_hz;

u16 flags;

#define SPI\_CONTROLLER\_HALF\_DUPLEX BIT(0) ;

#define SPI\_CONTROLLER\_NO\_RX BIT(1) ;

#define SPI\_CONTROLLER\_NO\_TX BIT(2) ;

#define SPI\_CONTROLLER\_MUST\_RX BIT(3) ;

#define SPI\_CONTROLLER\_MUST\_TX BIT(4) ;

#define SPI\_MASTER\_GPIO\_SS BIT(5) ;

bool slave;

size\_t (\*max\_transfer\_size)(struct spi\_device \*spi);

size\_t (\*max\_message\_size)(struct spi\_device \*spi);

struct mutex io\_mutex;

spinlock\_t bus\_lock\_spinlock;

struct mutex bus\_lock\_mutex;

bool bus\_lock\_flag;

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

int (\*transfer)(struct spi\_device \*spi, struct spi\_message \*mesg);

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

bool (\*can\_dma)(struct spi\_controller \*ctlr,struct spi\_device \*spi, struct spi\_transfer \*xfer);

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\_controller \*ctlr);

int (\*transfer\_one\_message)(struct spi\_controller \*ctlr, struct spi\_message \*mesg);

int (\*unprepare\_transfer\_hardware)(struct spi\_controller \*ctlr);

int (\*prepare\_message)(struct spi\_controller \*ctlr, struct spi\_message \*message);

int (\*unprepare\_message)(struct spi\_controller \*ctlr, struct spi\_message \*message);

int (\*slave\_abort)(struct spi\_controller \*ctlr);

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

int (\*transfer\_one)(struct spi\_controller \*ctlr, struct spi\_device \*spi, struct spi\_transfer \*transfer);

void (\*handle\_err)(struct spi\_controller \*ctlr, struct spi\_message \*message);

const struct spi\_controller\_mem\_ops \*mem\_ops;

int \*cs\_gpios;

struct spi\_statistics statistics;

struct dma\_chan \*dma\_tx;

struct dma\_chan \*dma\_rx;

void \*dummy\_rx;

void \*dummy\_tx;

int (\*fw\_translate\_cs)(struct spi\_controller \*ctlr, unsigned cs);

};

**成员**

dev

该驱动程序的设备接口

list

链接到全局 spi\_controller 列表

bus\_num

给定 SPI 控制器的板特定（通常是 SOC 特定）标识符。

num\_chipselect

chipselects 用于区分各个 SPI 从设备，并从 0 到 num\_chipselects 进行编号。每个从机都有一个片选信号，但通常不是每个片选都连接到一个从机。

dma\_alignment

SPI 控制器对 DMA 缓冲区对齐的约束。

mode\_bits

此控制器驱动程序理解的标志

bits\_per\_word\_mask

指示驱动程序支持哪些 bits\_per\_word 值的掩码。位 n 表示支持 bits\_per\_word n+1。如果设置，SPI 内核将拒绝任何具有不受支持的 bits\_per\_word 的传输。如果未设置，则此值将被忽略，并且由各个驱动程序执行任何验证。

min\_speed\_hz

支持的最低传输速度

max\_speed\_hz

支持的最高传输速度

flags

与此驱动程序相关的其他限制

slave

表明这是一个 SPI 从控制器

max\_transfer\_size

返回 a 的最大传输大小的函数[spi\_device](https://www.kernel.org/doc/html/v4.19/driver-api/spi.html" \l "c.spi_device" \o "spi_device)；可能是，因此将使用NULL默认值。SIZE\_MAX

max\_message\_size

返回 a 的最大消息大小的函数[spi\_device](https://www.kernel.org/doc/html/v4.19/driver-api/spi.html" \l "c.spi_device" \o "spi_device)；可能是，因此将使用NULL默认值。SIZE\_MAX

io\_mutex

物理总线访问互斥锁

bus\_lock\_spinlock

用于 SPI 总线锁定的自旋锁

bus\_lock\_mutex

用于排除多个呼叫者的互斥体

bus\_lock\_flag

表示SPI总线被锁定为独占使用

setup

更新设备的 SPI 控制器使用的设备模式和时钟记录；协议代码可以调用这个。如果请求无法识别或不支持的模式，则此操作必须失败。调用它总是安全的，除非传输在其设置被修改的设备上挂起。

transfer

向控制器的传输队列添加一条消息。

cleanup

释放特定于控制器的状态

can\_dma

判断这个控制器是否支持 DMA

queued

此控制器是否提供内部消息队列

kworker

消息泵的线程结构

kworker\_task

指向消息泵 kworker 线程的任务指针

pump\_messages

用于将工作安排到消息泵的工作结构

queue\_lock

自旋锁以同步对消息队列的访问

queue

消息队列

cur\_msg

当前飞行消息

idling

设备进入空闲状态

busy

消息泵正忙

running

消息泵正在运行

rt

此队列是否设置为作为实时任务运行

auto\_runtime\_pm

内核应确保在准备硬件时保留运行时 PM 引用，使用 spidev 的父设备

cur\_msg\_prepared

spi\_prepare\_message 被调用用于当前正在运行的消息

cur\_msg\_mapped

消息已映射到 DMA

xfer\_completion

核心使用transfer\_one\_message()

max\_dma\_len

设备的 DMA 传输的最大长度。

prepare\_transfer\_hardware

一条消息很快就会从队列中到达，因此子系统通过发出此调用请求驱动程序准备传输硬件

transfer\_one\_message

子系统调用驱动程序来传输单个消息，同时对同时到达的传输进行排队。当驱动程序完成此消息时，它必须调用 [spi\_finalize\_current\_message()](https://www.kernel.org/doc/html/v4.19/driver-api/spi.html" \l "c.spi_finalize_current_message" \o "spi_finalize_current_message)以便子系统可以发出下一条消息

unprepare\_transfer\_hardware

当前队列中没有更多消息，因此子系统通知驱动程序它可以通过发出此调用来放松硬件

prepare\_message

设置控制器以传输单个消息，例如进行 DMA 映射。从线程上下文中调用。

unprepare\_message

撤消 所做的任何工作prepare\_message()。

slave\_abort

中止 SPI 从控制器上正在进行的传输请求

set\_cs

设置片选线的逻辑电平。可以从中断上下文中调用。

transfer\_one

传输单个 spi\_transfer。- 如果传输完成则返回 0， - 如果传输仍在进行中则返回 1。什么时候

驱动程序完成了它必须调用的此传输，[spi\_finalize\_current\_transfer()](https://www.kernel.org/doc/html/v4.19/driver-api/spi.html" \l "c.spi_finalize_current_transfer" \o "spi_finalize_current_transfer)以便子系统可以发出下一个传输。注意：transfer\_one 和 transfer\_one\_message 是互斥的；当两者都设置时，通用子系统不会调用您的 transfer\_one 回调。

handle\_err

子系统调用驱动程序来处理在 的通用实现中发生的错误transfer\_one\_message()。

mem\_ops

用于与 SPI 内存交互的优化/专用操作。该字段是可选的，只有在控制器具有对类似内存的操作的本机支持时才应实现。

cs\_gpios

用作芯片选择线的 GPIO 阵列；每个 CS 编号一个。对于不是 GPIO 的 CS 线（由 SPI 控制器本身驱动），任何单独的值都可以是 -ENOENT。

statistics

spi\_controller 的统计信息

dma\_tx

DMA传输通道

dma\_rx

DMA接收通道

dummy\_rx

全双工设备的虚拟接收缓冲区

dummy\_tx

全双工设备的虚拟传输缓冲区

fw\_translate\_cs

如果引导固件使用 Linux 期望的不同编号方案，则可以使用此可选挂钩在两者之间进行转换。

**说明**

每个SPI控制器都可以与一个或多个spi\_device子项通信。它们组成一个小型总线，共享MOSI、MISO和SCK信号，但不共享片选信号。每个设备可以配置为使用不同的时钟速率，因为这些共享信号仅在选择芯片时才会被使用。

SPI控制器的驱动程序通过spi\_message事务队列管理对这些设备的访问，将数据在CPU内存和SPI从设备之间进行复制。对于每个排队的消息，在事务完成后调用该消息的完成函数。

## spi\_resstruct

SPI资源管理结构

**定义**

struct spi\_res {

struct list\_head entry;

spi\_res\_release\_t release;

unsigned long long data[];

};

**成员**

entry

列表条目

release

在释放此资源之前调用的释放代码

data

为特定用例分配的额外数据

**说明**

这是基于devres的想法，但专注于spi\_message处理期间的生命周期管理

## spi\_transferstruct

读/写缓冲区对

**定义**

struct spi\_transfer {

const void \*tx\_buf;

void \*rx\_buf;

unsigned len;

dma\_addr\_t tx\_dma;

dma\_addr\_t rx\_dma;

struct sg\_table tx\_sg;

struct sg\_table rx\_sg;

unsigned dummy\_data:1;

unsigned cs\_off:1;

unsigned cs\_change:1;

unsigned tx\_nbits:3;

unsigned rx\_nbits:3;

#define SPI\_NBITS\_SINGLE 0x01 ;

#define SPI\_NBITS\_DUAL 0x02 ;

#define SPI\_NBITS\_QUAD 0x04 ;

u8 bits\_per\_word;

u16 delay\_usecs;

u32 speed\_hz;

struct list\_head transfer\_list;

};

**成员**

tx\_buf

要写入的数据（dma-safe内存）或NULL

rx\_buf

要读取的数据（dma-safe内存）或NULL

len

rx和tx缓冲区的大小（以字节为单位）

tx\_dma

tx\_buf的DMA地址，如果spi\_message.is\_dma\_mapped

rx\_dma

rx\_buf的DMA地址，如果spi\_message.is\_dma\_mapped

tx\_sg

传输的散列表，目前不用于客户端使用

rx\_sg

接收的散列表，目前不用于客户端使用

dummy\_data

表示传输是虚拟字节传输。

cs\_off

执行带有关闭片选的传输。

cs\_change

影响此传输完成后的片选。

tx\_nbits

用于编写的位数。如果为0，则使用默认值（SPI\_NBITS\_SINGLE）。

rx\_nbits

用于读取的位数。如果为0，则使用默认值（SPI\_NBITS\_SINGLE）。

bits\_per\_word

为此传输选择一个bits\_per\_word，而不是设备默认值。如果为0，则使用默认值（从spi\_device）。

delay\_usecs

在此传输之后延迟微秒数，然后（可选）更改片选状态，然后启动下一个传输或完成此spi\_message。

speed\_hz

为此传输选择一个速度，而不是设备默认值。如果为0，则使用默认值（从spi\_device）。

transfer\_list

传输通过spi\_message.transfers进行排序。

**说明**

SPI传输始终写入与读取相同数量的字节。协议驱动程序应始终提供rx\_buf和/或tx\_buf。在某些情况下，它们可能还希望为正在传输的数据提供DMA地址；当底层驱动程序使用dma时，可以减少开销。

如果传输缓冲区为null，则会在填充rx\_buf时移位零。如果接收缓冲区为null，则移位进来的数据将被丢弃。仅移位“len”字节（或进来）。尝试移位部分字是错误的。（例如，通过使用大小为16或20位的字移位3个字节；前者使用每个字节2个字节，后者使用每个字节4个字节。）

内存中的数据值始终处于本机CPU字节顺序中，从线传递的字节顺序（大端字节序，除了SPI\_LSB\_FIRST）进行转换。因此，例如当bits\_per\_word为16时，缓冲区的长度为2N字节（len = 2N），并且在CPU字节顺序中保持N个16位字。

当SPI传输的字大小不是8位的2的次幂时，这些内存字包括额外的位。协议驱动程序始终将内存字视为右对齐，因此未定义的（rx）或未使用的（tx）位始终是最高位。

所有SPI传输都始于相关芯片选择。通常它会在消息中的最后传输之前保持选中。驱动程序可以使用cs\_change来影响芯片选择信号。

如果传输不是消息中的最后一个，则使用此标志使芯片选择在消息中间短暂地变为非活动状态。以这种方式切换芯片选择可能需要终止芯片命令，使单个spi\_message执行所有的芯片事务组。

当传输是消息中的最后一个时，芯片可能会保持选中状态，直到下一个传输。在没有阻止消息发送到其他设备的多设备SPI总线上，这只是一个性能提示；开始发送到另一个设备的消息会取消选择该设备。但在其他情况下，这可以用于确保正确性。一些设备需要将协议事务构建为一系列spi\_message提交，其中一个消息的内容由前一个消息的结果确定，整个事务在芯片选择变为非活动状态时结束。

当SPI在1x，2x或4x中传输时，可以通过tx\_nbits和rx\_nbits从设备获取传输信息。在双向传输中，这两个应同时设置。用户可以使用SPI\_NBITS\_SINGLE（1x）SPI\_NBITS\_DUAL（2x）和SPI\_NBITS\_QUAD（4x）设置传输模式，以支持这三种传输。

提交spi\_message（及其spi\_transfers）到较低层的代码负责管理其内存。零初始化您没有明确设置的每个字段，以隔离未来的API更新。在提交消息及其传输之后，直到其完成回调再将其忽略。

## struct spi\_message

一个多段SPI事务

**定义**

struct spi\_message {

struct list\_head transfers;

struct spi\_device \*spi;

unsigned is\_dma\_mapped:1;

void (\*complete)(void \*context);

void \*context;

unsigned frame\_length;

unsigned actual\_length;

int status;

struct list\_head queue;

void \*state;

struct list\_head resources;

bool prepared;

};

**成员**

transfers

此事务中传输段的列表

spi

事务排队的SPI设备

is\_dma\_mapped

如果为true，则调用方为每个传输缓冲区提供了dma和cpu虚拟地址

complete

用于报告事务完成情况的回调

context

调用complete（）时的参数

frame\_length

消息中的总字节数

actual\_length

在所有成功段中传输的总字节数

status

成功为零，其他为负的errno

queue

由当前拥有消息的驱动程序使用

state

由当前拥有消息的驱动程序使用

resources

当处理spi消息时用于资源管理

prepared

为此消息调用了spi\_prepare\_message

**说明**

spi\_message用于执行数据传输的原子序列，每个传输由struct spi\_transfer表示。在以下意义上，序列是“原子”的在该序列完成之前，没有其他spi\_message可以使用该SPI总线。在某些系统上，许多这样的序列可以作为单个编程DMA传输执行。在所有系统上，这些消息都是排队的，并且可能在向其他设备的交易之后完成。发送到给定spi\_device的消息始终按FIFO顺序执行。

提交spi\_message （及其spi\_transfers）到较低层的代码负责管理其内存。零初始化您没有明确设置的每个字段，以隔离未来的API更新。在提交消息及其传输之后，直到其完成回调再将其忽略。

## void spi\_message\_init\_with\_transfers(struct spi\_message \*m, struct spi\_transfer \*xfers, unsigned int num\_xfers)

初始化spi\_message并附加传输

**参数**

struct spi\_message \*m

要初始化的spi\_message

struct spi\_transfer \*xfers

spi传输的数组

unsigned int num\_xfers

xfer数组中的项目数

**说明**

此函数初始化给定的spi\_message并将给定数组中的每个spi\_transfer添加到消息中。

## struct spi\_replaced\_transfers

说明spi\_transfer替换的结构，以便可以将其还原

**定义**

struct spi\_replaced\_transfers {

spi\_replaced\_release\_t release;

void \*extradata;

struct list\_head replaced\_transfers;

struct list\_head \*replaced\_after;

size\_t inserted;

struct spi\_transfer inserted\_transfers[];

};

**成员**

release

在释放此结构之前执行一些额外的发布代码

extradata

如果请求或 NULL，指向一些额外数据的指针

replaced\_transfers

已被替换且需要恢复的传输

replaced\_after

重新插入replaced\_transfers之后的传输

inserted

插入的传输数

inserted\_transfers

数组大小的 spi\_transfers 数组inserted，已经替换了 replaced\_transfers

**注意**

如果请求了一些额外的分配，则extradata将指向inserted\_transfers [inserted]，因此对spi\_transfers的对齐将相同

## int spi\_sync\_transfer(struct spi\_device \*spi, struct spi\_transfer \*xfers, unsigned int num\_xfers)

同步SPI数据传输

**参数**

struct spi\_device \*spi

与之交换数据的设备

struct spi\_transfer \*xfers

spi转移的数组

unsigned int num\_xfers

xfer数组中的项目数

**上下文**

可以休眠

**说明**

执行给定spi\_transfer数组的同步SPI数据传输。

有关更具体的语义，请参见spi\_sync（）。

**返回**

成功为零，否则为负错误代码。

## int spi\_write(struct spi\_device \*spi, const void \*buf, size\_t len)

SPI同步写

**参数**

struct spi\_device \*spi

将向其写入数据的设备

const void \*buf

数据缓冲区

size\_t len

数据缓冲区大小

**上下文**

可以休眠

**说明**

此函数写入缓冲区buf。仅从可睡眠上下文调用。

**返回**

成功为零，否则为负错误代码。

## int spi\_read(struct spi\_device \*spi, void \*buf, size\_t len)

SPI同步读取

**参数**

struct spi\_device \*spi

将从其读取数据的设备

void \*buf

数据缓冲区

size\_t len

数据缓冲区大小

**上下文**

可以休眠

**说明**

此函数读取缓冲区buf。仅从可睡眠上下文调用。

**返回**

成功为零，否则为负错误代码。

## ssize\_t spi\_w8r8(struct spi\_device \*spi, u8 cmd)

SPI同步8位写入后接8位读取

**参数**

struct spi\_device \*spi

将要交换数据的设备

u8 cmd

在读取数据之前要写的命令

**上下文**

可睡眠

**说明**

只能从可睡眠的上下文中调用。

**返回**

设备返回的（无符号的）8位数字，否则为负错误代码。

## ssize\_t spi\_w8r16(struct spi\_device \*spi, u8 cmd)

SPI同步8位写入，后跟16位读取

**参数**

struct spi\_device \*spi

将要交换数据的设备

u8 cmd

在读取数据之前要写的命令

**上下文**

可睡眠

**说明**

该数字按线序返回，至少有时是大端序。

只能从可睡眠的上下文中调用。

**返回**

设备返回的（无符号的）16位数字，否则为负错误代码。

## ssize\_t spi\_w8r16be(struct spi\_device \*spi, u8 cmd)

SPI同步8位写入，后跟16位大端序读取

**参数**

struct spi\_device \*spi

将要交换数据的设备

u8 cmd

在读取数据之前要写的命令

**上下文**

可睡眠

**说明**

这个函数类似于spi\_w8r16，但它将把读取的16位数据字从大端序转换为本机字节序。

只能从可睡眠的上下文中调用。

**返回**

设备在CPU字节序中返回的（无符号的）16位数字，否则为负错误代码。

## struct spi\_board\_info

特定板的SPI设备模板

**定义**

struct spi\_board\_info {

char modalias[SPI\_NAME\_SIZE];

const void \*platform\_data;

const struct property\_entry \*properties;

void \*controller\_data;

int irq;

u32 max\_speed\_hz;

u16 bus\_num;

u16 chip\_select;

u16 mode;

};

**成员**

modalias

初始化spi\_device.modalias; 用于标识驱动程序。

platform\_data

初始化spi\_device.platform\_data; 存储在其中的特定数据是特定于驱动程序的。

properties

设备的附加属性。

controller\_data

初始化spi\_device.controller\_data; 一些控制器需要有关硬件设置的提示，例如DMA。

irq

初始化spi\_device.irq; 取决于板线接法。

max\_speed\_hz

初始化spi\_device.max\_speed\_hz; 基于芯片数据表和特定于板的信号质量问题的限制。

bus\_num

标识spi\_controller的父级spi\_device; 未被spi\_new\_device()使用，否则取决于板线接法。

chip\_select

初始化spi\_device.chip\_select; 取决于板线接法。

mode

初始化spi\_device.mode; 基于芯片数据表，板线接法（一些设备支持3WIRE和标准模式），以及芯片选择路径中可能存在的反相器的存在。

**说明**

当向设备树添加新的SPI设备时，这些结构充当部分设备模板。它们保存了驱动程序无法确定的信息。探测（例如默认的传输字长）可以建立的信息不包括在这里。

这些结构在两个地方使用。它们的主要作用是存储在板特定的设备说明符表中，在板初始化早期声明，然后用于在该控制器的驱动程序初始化后填充该控制器的设备树。次要（非典型）角色是作为参数传递给spi\_new\_device()调用，该调用在这些控制器驱动程序在某些动态板配置模型中活跃后发生。

## int spi\_register\_board\_info(struct spi\_board\_info const \*info, unsigned n)

为给定的板注册SPI设备

**参数**

struct spi\_board\_info const \*info

芯片说明符数组

unsigned n

提供的说明符数

**上下文**

可以睡眠

**说明**

特定于板的早期初始化代码在此调用（可能在arch\_initcall期间）。任何设备节点都是在相关的父SPI控制器（bus\_num）定义之后创建的。我们将这些设备的数据表保留下来，以便重新加载控制器驱动程序不会使Linux忘记这些硬定线设备。

其他代码也可以调用此代码，例如特定的附加板可以通过其扩展连接器提供SPI设备，因此初始化该板的代码自然会声明其SPI设备。

传递的板信息可以安全地为\_\_initdata ... 但要小心任何嵌入式指针（platform\_data等），它们将按原样复制。

**返回**

成功返回零，否则为负错误代码。

## int \_\_spi\_register\_driver(struct module \*owner, struct spi\_driver \*sdrv)

注册SPI驱动程序

**参数**

struct module \*owner

要注册的驱动程序的所有者模块

struct spi\_driver \*sdrv

要注册的驱动程序

**上下文**

可以睡眠

**返回**

成功返回零，否则为负错误代码。

## struct spi\_device \*spi\_alloc\_device(struct spi\_controller \*ctlr)

分配新的SPI设备

**参数**

struct spi\_controller \*ctlr

连接到该设备的控制器

**上下文**

可以睡眠

**说明**

允许驱动程序分配和初始化spi\_device，而不立即注册它。这使驱动程序能够在调用spi\_add\_device()之前直接填充spi\_device的设备参数。

调用者有责任在返回的spi\_device结构上调用spi\_add\_device()将其添加到SPI控制器中。如果调用者需要放弃spi\_device而不添加它，则应在其上调用spi\_dev\_put()。

**返回**

指向新设备的指针，或者NULL。

## int spi\_add\_device(struct spi\_device \*spi)

使用spi\_alloc\_device分配的spi\_device添加到spi总线上的伴随函数。

**参数**

struct spi\_device \*spi

要注册的spi\_device

**说明**

使用spi\_alloc\_device分配的设备可以通过该函数添加到spi总线上。

**返回**

成功返回0；失败返回负的错误代码

## struct spi\_device \*spi\_new\_device(struct spi\_controller \*ctlr, struct spi\_board\_info \*chip)

实例化一个新的SPI设备

**参数**

struct spi\_controller \*ctlr

连接设备的控制器

struct spi\_board\_info \*chip

说明SPI设备

**上下文**

可以休眠

**说明**

在典型的主板上，这是纯粹的内部操作；在板子初始化创建硬连线设备后就不再需要它了。一些开发平台可能无法使用spi\_register\_board\_info，因此将其导出，以便例如基于USB或并口的适配器驱动程序可以添加设备（驱动程序可以在机外了解到设备）。

**返回**

新设备或NULL。

## void spi\_unregister\_device(struct spi\_device \*spi)

注销单个SPI设备

**参数**

struct spi\_device \*spi

要注销的spi\_device

**说明**

开始让传递的SPI设备消失。通常这将由spi\_unregister\_controller()处理。

## void spi\_finalize\_current\_transfer(struct spi\_controller \*ctlr)

报告传输完成

**参数**

struct spi\_controller \*ctlr

报告完成的控制器

**说明**

由使用核心transfer\_one\_message()实现的SPI驱动程序调用，以通知它当前中断驱动的传输已经完成，下一个传输可能会被安排。

## struct spi\_message \*spi\_get\_next\_queued\_message(struct spi\_controller \*ctlr)

由驱动程序调用，检查是否有排队的消息

**参数**

struct spi\_controller \*ctlr

要检查的控制器是否有排队的消息

**说明**

如果队列中还有更多的消息，则从此调用返回下一个消息。

**返回**

队列中的下一条消息，否则如果队列为空则返回NULL。

## void spi\_finalize\_current\_message(struct spi\_controller \*ctlr)

当前的消息已经完成

**参数**

struct spi\_controller \*ctlr

返回消息的控制器

**说明**

由驱动程序调用，通知核心队列的前端消息已完成，可以从队列中删除。

## int spi\_slave\_abort(struct spi\_device \*spi)

中止SPI从站控制器上正在进行的传输请求

**参数**

struct spi\_device \*spi

用于当前传输的设备

## struct spi\_controller \*\_\_spi\_alloc\_controller(struct device \*dev, unsigned int size, bool slave)

分配一个SPI主控制器或从控制器

**参数**

struct device \*dev

控制器，可能使用平台总线

unsigned int size

要分配多少个零私有数据驱动程序;此内存的指针在返回的设备的driver\_data字段中，可通过spi\_controller\_get\_devdata()访问；该内存是缓存行对齐的；授予DMA访问其私有数据部分的驱动程序需要使用ALIGN(size, dma\_get\_cache\_alignment())将其四舍五入。

bool slave

指示是否要分配SPI主（false）或SPI从（true）控制器的标志

**上下文**

可以休眠

**说明**

SPI控制器驱动程序可以直接接触芯片寄存器，因此仅使用该调用。这是它们分配spi\_controller结构的方式，在调用spi\_register\_controller()之前。

必须从可以休眠的上下文中调用此函数。

调用者负责在调用spi\_register\_controller()之前分配总线号并初始化控制器方法；并在添加设备时（添加设备后出现错误）调用spi\_controller\_put()以防止内存泄漏。

**返回**

成功时是SPI控制器结构，否则为NULL。

## struct spi\_controller \*\_\_devm\_spi\_alloc\_controller(struct device \*dev, unsigned int size, bool slave)

资源管理\_\_spi\_alloc\_controller()

**参数**

struct device \*dev

SPI控制器的物理设备

unsigned int size

要分配多少个零私有数据的驱动程序

bool slave

是否分配SPI主（false）或SPI从（true）控制器

**上下文**

可以休眠

**说明**

分配SPI控制器，并在从其驱动程序解绑定dev时自动释放对它的引用。因此，驱动程序不必调用spi\_controller\_put()。

此函数的参数与\_\_spi\_alloc\_controller()相同。

**返回**

成功时是SPI控制器结构，否则为NULL。

## int spi\_register\_controller(struct spi\_controller \*ctlr)

注册SPI主控制器或从控制器

**参数**

struct spi\_controller \*ctlr

初始化主控制器，最初来自spi\_alloc\_master()或spi\_alloc\_slave()

**上下文**

可以休眠

**说明**

SPI控制器使用某些非SPI总线（例如平台总线）连接到其驱动程序。该代码中的probe()的最后阶段包括调用spi\_register\_controller()以连接到这个SPI总线胶水。

SPI控制器使用特定于板子的（通常是SOC特定的）总线编号，SPI设备的板子特定寻址将这些编号与片选编号组合。由于SPI不直接支持动态设备识别，板需要配置表指出哪个芯片在哪个地址。

必须从可以休眠的上下文中调用此函数。它返回成功返回零，否则是负的错误代码（释放控制器的引用计数）。成功返回后，调用者负责调用spi\_unregister\_controller()。

**返回**

成功返回零，否则返回负错误代码。

## int devm\_spi\_register\_controller（struct device \* dev，struct spi\_controller \* ctlr）

注册所管理的 SPI 主控制器或从控制器

**参数**

struct device \* dev

管理 SPI 控制器的设备

struct spi\_controller \* ctlr

初始化的控制器，最初来自 spi\_alloc\_master()或 spi\_alloc\_slave()

环境

可以休眠

**说明**

使用 spi\_register\_controller() 注册 SPI 设备，该设备将自动取消注册并释放。

**返回**

成功返回零，否则返回负错误代码。

## void spi\_unregister\_controller（struct spi\_controller \* ctlr）

取消注册 SPI 主控制器或从控制器

**参数**

struct spi\_controller \* ctlr

被注销的控制器

环境

可以休眠

**说明**

此调用仅由 SPI 控制器驱动程序使用，它们是唯一直接触摸芯片寄存器的驱动程序。

必须从可以休眠的上下文中调用此函数。

请注意，此函数还会释放控制器的引用。

## struct spi\_controller \* spi\_busnum\_to\_master（u16 bus\_num）

查找与 bus\_num 相关联的主机

**参数**

u16 bus\_num

主的总线编号

环境

可以休眠

**说明**

可以使用此调用与在架构初始化时间之后注册的设备一起使用。它返回指向相关 spi\_controller 的 refcounted 指针（调用者必须释放该指针），或者如果没有注册此类主控制器，则返回 NULL。

**返回**

成功返回 SPI 主结构，否则返回 NULL。

## void \* spi\_res\_alloc（struct spi\_device \* spi，spi\_res\_release\_t release，size\_t size，gfp\_t gfp）

在 spi\_transfer\_one 处理期间进行生命周期管理的 SPI 资源分配

**参数**

struct spi\_device \* spi

我们为其分配内存的 SPI 设备

spi\_res\_release\_t release

用于此资源的释放代码

size\_t size

要分配和返回的大小

gfp\_t gfp

GFP 分配标志

**返回**

分配数据的指针

将来可以将此功能增强为从 spi\_device 或 spi\_controller 的内存池分配，以避免重复分配。

## void spi\_res\_free（void \* res）

释放 spi 资源

**参数**

void \* res

指向资源自定义数据的指针

## void spi\_res\_add（struct spi\_message \* message，void \* res）

将 spi\_res 添加到 spi\_message

**参数**

struct spi\_message \* message

spi 消息

void \* res

spi\_resource

## void spi\_res\_release（struct spi\_controller \* ctlr，struct spi\_message \* message）

释放此消息所有 spi 资源

**参数**

struct spi\_controller \* ctlr

the spi\_controller

struct spi\_message \* message

the spi\_message

## struct spi\_replaced\_transfers \* spi\_replace\_transfers（struct spi\_message \* msg，struct spi\_transfer \* xfer\_first，size\_t remove，size\_t insert，spi\_replaced\_release\_t release，size\_t extradatasize，gfp\_t gfp）

用数个传输替换传输，并使用 spi\_message.resources 注册更改

**参数**

struct spi\_message \* msg

我们正在处理的 spi\_message

struct spi\_transfer \* xfer\_first

我们要替换的第一个 spi\_transfer

size\_t remove

要删除的传输数量

size\_t insert

我们要插入的传输数量

spi\_replaced\_release\_t release

在某些情况下需要的额外释放代码

size\_t extradatasize

要分配的额外数据（具有 struct spi\_transfer 的对齐保证）

gfp\_t gfp

gfp 标志

**返回**

指向 spi\_replaced\_transfers 的指针，

错误的话为 PTR\_ERR(...)。

## int spi\_split\_transfers\_maxsize（struct spi\_controller \* ctlr，struct spi\_message \* msg，size\_t maxsize，gfp\_t gfp）

当单个传输超过某个大小时，将 spi 传输拆分为多个传输

**参数**

struct spi\_controller \* ctlr

此传输的 spi\_controller

struct spi\_message \* msg

要转换的 spi\_message

size\_t maxsize

应用此时的最大值

gfp\_t gfp

GFP 分配标志

**返回**

转换状态

## int spi\_setup（struct spi\_device \* spi）

设置 SPI 模式和时钟速率

**参数**

struct spi\_device \* spi

正在修改其设置的设备

环境

可以休眠，并且没有请求排队到该设备上

**说明**

如果设备不能使用默认值，则 SPI 协议驱动程序可能需要更新传输模式。它们可能还需要从初始值更新时钟速率或字大小。此函数更改这些设置，并必须从可以休眠的上下文中调用。除了 SPI\_CS\_HIGH，该设置立即生效，更改将在下一次选择设备并传输数据到或从设备时生效。当此函数返回时，spi 设备将被取消选择。

请注意，如果协议驱动程序指定底层控制器或其驱动程序不支持的选项，则此调用将失败。例如，并非所有硬件都支持使用九位字的线传输、LSB-first 线编码或 active-high 芯片选择。

**返回**

成功返回零，否则返回负错误代码。

## int spi\_async（struct spi\_device \* spi，struct spi\_message \* message）

异步 SPI 传输

**参数**

struct spi\_device \* spi

要交换数据的设备

struct spi\_message \* message

说明数据传输，包括完成回调

环境

任何（可以阻塞中断等）

**说明**

可以在 irq 和其他无法休眠的上下文以及可以休眠的任务上下文中使用此调用。

完成回调在不能休眠的上下文中调用。在该调用之前，message->status 的值未定义。当调用回调时，message->status 保持零（表示完全成功）或负错误代码。在该回调返回之后，发出传输请求的驱动程序可以释放相关内存；它不再由任何 SPI 核或控制器驱动程序代码使用。

请注意，虽然所有发送到spi\_device的消息都按照FIFO顺序处理，但是消息可能以其他顺序发送到不同的设备。例如，某个设备可能具有更高的优先级或具有各种“硬”访问时间要求。

在传输过程中检测到任何故障时，将中止整个消息的处理，并取消选择设备。在返回关联的消息完成回调之前，不会处理排队到该设备的任何其他spi\_message。 （此规则同样适用于所有同步传输调用，这些调用是核心异步原语的包装器。）

成功返回零，否则返回负错误代码。

## int spi\_async\_locked（struct spi\_device \* spi，struct spi\_message \* message）

具有排他总线使用权的spi\_async版本

**参数**

struct spi\_device \* spi

将要交换数据的设备

struct spi\_message \* message

说明数据传输，包括完成回调

**上下文**

任何（可能会阻塞irq等等）

**说明**

此调用可在无法睡眠的中断请求和其他上下文中使用，也可在可睡眠的任务上下文中使用。

在回调之前，消息-> status的值未定义。发出回调时，消息-> status为零（表示完全成功）或负错误代码。在回调返回后，发出传输请求的驱动程序可以释放关联的内存；它不再被任何SPI核心或控制器驱动程序代码使用。

请注意，虽然所有发送到spi\_device的消息都按照FIFO顺序处理，但是消息可能以其他顺序发送到不同的设备。例如，某个设备可能具有更高的优先级或具有各种“硬”访问时间要求。

在传输过程中检测到任何故障时，将中止整个消息的处理，并取消选择设备。在返回关联的消息完成回调之前，不会处理排队到该设备的任何其他spi\_message。 （此规则同样适用于所有同步传输调用，这些调用是核心异步原语的包装器。）

成功返回零，否则返回负错误代码。

## int spi\_sync（struct spi\_device \* spi，struct spi\_message \* message）

阻塞/同步SPI数据传输

**参数**

struct spi\_device \* spi

将要交换数据的设备

struct spi\_message \* message

说明数据传输

**上下文**

可以休眠

**说明**

此调用只能从可以休眠的上下文中使用。睡眠是不可中断的，没有超时。低开销控制器驱动程序可以直接DMA到和离开消息缓冲区。

请注意，SPI设备的芯片选择在消息期间处于活动状态，然后通常在消息之间禁用。某些经常使用的设备的驱动程序可能希望最小化选择芯片的成本，通过保留对芯片的选择来预期下一条消息将发送到同一芯片。 （这可能增加功耗。）

另外，调用者保证与消息相关联的内存不会在此调用返回之前释放。

成功返回零，否则返回负错误代码。

## int spi\_sync\_locked（struct spi\_device \* spi，struct spi\_message \* message）

具有排他总线使用权的spi\_sync版本

**参数**

struct spi\_device \* spi

将要交换数据的设备

struct spi\_message \* message

说明数据传输

**上下文**

可以休眠

**说明**

此调用只能从可以休眠的上下文中使用。睡眠是不可中断的，没有超时。低开销控制器驱动程序可以直接DMA到和离开消息缓冲区。

驱动程序需要独占访问SPI总线时，应使用此调用。它必须在spi\_bus\_lock调用之前。当独占访问结束时，必须通过spi\_bus\_unlock调用释放SPI总线。在保持SPI总线锁定时，必须使用spi\_sync\_locked和spi\_async\_locked调用进行数据传输。

成功返回零，否则返回负错误代码。

## int spi\_bus\_lock（struct spi\_controller \* ctlr）

获得排他性SPI总线使用权的锁定

**参数**

struct spi\_controller \* ctlr

应锁定以实现独占总线访问权限的SPI总线主

**上下文**

可以休眠

**说明**

此调用只能从可以休眠的上下文中使用。睡眠是不可中断的，没有超时。

驱动程序需要独占访问SPI总线时，应使用此调用。在结束独占访问时，必须通过spi\_bus\_unlock调用释放SPI总线。在保持SPI总线锁定时，必须使用spi\_sync\_locked和spi\_async\_locked调用进行数据传输。

总是返回零。

## int spi\_bus\_unlock（struct spi\_controller \* ctlr）

释放排他性SPI总线使用权限的锁定

**参数**

struct spi\_controller \* ctlr

已锁定以实现独占总线访问权限的SPI总线主

**上下文**

可以休眠

**说明**

此调用只能从可以休眠的上下文中使用。睡眠是不可中断的，没有超时。

该调用释放先前由spi\_bus\_lock调用获取的SPI总线锁定。

**返回**

始终为零。

## int spi\_write\_then\_read（struct spi\_device \* spi，const void \* txbuf，unsigned n\_tx，void \* rxbuf，unsigned n\_rx）

SPI同步写入后读取

**参数**

struct spi\_device \* spi

将与之交换数据的设备

const void \* txbuf

要写入的数据（不必为dma-safe）

unsigned n\_tx

txbuf的大小，以字节为单位

void \* rxbuf

要读取数据的缓冲区（不必为dma-safe）

unsigned n\_rx

rxbuf的大小，以字节为单位

**上下文**

可以睡眠

**说明**

这执行半双工MicroWire样式的与设备的交易，发送txbuf，然后读取rxbuf。成功返回为零，否则为负的errno状态码。该调用只能从可能休眠的上下文中使用。

这个例程的参数总是使用小缓冲区复制。对性能敏感或批量传输代码应改用具有dma-safe缓冲区的spi\_ {async，sync}（）调用。

**返回**

成功返回零，否则为负错误代码。

# I2C和SMBus子系统

I2C（或无花哨的排版，“I2C”）是“互联 IC”总线的简称，是一种简单的总线协议，适用于低数据速率通信。由于它还是一个经授权的商标，因此一些供应商使用另一个名称（例如“两线接口”，TWI）来表示相同的总线。I2C只需要两个信号（SCL用于时钟，SDA用于数据），节省了电路板的实际空间，并最小化了信号质量问题。大多数I2C设备使用七位地址，并且总线速度高达400 kHz；有一个高速扩展（3.4 MHz）尚未广泛使用。I2C是多主机总线；使用开放式驱动器信号来仲裁主机之间，以及进行握手并从较慢的客户端同步时钟。

Linux I2C编程接口支持总线交互的主侧和从侧。编程接口围绕两种驱动程序和两种设备结构。一个I2C“适配器驱动程序”抽象控制器硬件；它绑定到物理设备（可能是PCI设备或platform\_device）并公开表示其管理的每个I2C总线段的结构i2c\_adapter。在每个I2C总线段上将有由struct i2c\_client表示的I2C设备。这些设备将绑定到一个符合标准Linux驱动程序模型的结构i2c\_driver。有各种I2C协议操作的函数；在本写作时，所有这些函数都只能从任务上下文中使用。

系统管理总线（SMBus）是一个类似的协议。大多数SMBus系统也符合I2C。SMBus的电气约束更严格，并且它标准化了特定的协议消息和习惯用法。支持I2C的控制器也可以支持大多数SMBus操作，但SMBus控制器不支持I2C控制器提供的所有协议选项。有各种SMBus协议操作的函数，可以使用I2C基元或通过向不支持那些I2C操作的i2c\_adapter设备发出SMBus命令来执行这些操作。

## int i2c\_master\_recv（const struct i2c\_client \* client，char \* buf，int count）

以主接收模式发出单个I2C消息

**参数**

const struct i2c\_client \* client

从设备的句柄

char \* buf

从从设备读取的数据的存储位置

int count

要读取的字节数，必须小于64k，因为msg.len是u16

**说明**

返回负errno，否则读取的字节数。

## int i2c\_master\_recv\_dmasafe（const struct i2c\_client \* client，char \* buf，int count）

使用DMA安全缓冲区在主接收模式下发出单个I2C消息

**参数**

const struct i2c\_client \* client

从设备的句柄

char \* buf

从从设备读取的数据的存储位置，必须针对DMA安全

int count

要读取的字节数，必须小于64k，因为msg.len是u16

**说明**

返回负errno，否则读取的字节数。

## int i2c\_master\_send（const struct i2c\_client \* client，const char \* buf，int count）

在主传输模式下发出单个I2C消息

**参数**

const struct i2c\_client \* client

从设备的句柄

const char \* buf

将写入从设备的数据

int count

要写入的字节数，必须小于64k，因为msg.len是u16

**说明**

返回负errno，否则写入的字节数。

## int i2c\_master\_send\_dmasafe（const struct i2c\_client \* client，const char \* buf，int count）

使用DMA安全缓冲区在主传输模式下发出单个I2C消息

**参数**

const struct i2c\_client \* client

从设备的句柄

const char \* buf

将写入从设备的数据，必须安全使用DMA

int count

要写入的字节数，必须小于64k，因为msg.len是u16

**说明**

返回负 errno，否则返回写入的字节数。

## struct i2c\_device\_identity

i2c 客户设备识别

**定义**

struct i2c\_device\_identity {

u16 manufacturer\_id;

#define I2C\_DEVICE\_ID\_NXP\_SEMICONDUCTORS 0;

#define I2C\_DEVICE\_ID\_NXP\_SEMICONDUCTORS\_1 1;

#define I2C\_DEVICE\_ID\_NXP\_SEMICONDUCTORS\_2 2;

#define I2C\_DEVICE\_ID\_NXP\_SEMICONDUCTORS\_3 3;

#define I2C\_DEVICE\_ID\_RAMTRON\_INTERNATIONAL 4;

#define I2C\_DEVICE\_ID\_ANALOG\_DEVICES 5;

#define I2C\_DEVICE\_ID\_STMICROELECTRONICS 6;

#define I2C\_DEVICE\_ID\_ON\_SEMICONDUCTOR 7;

#define I2C\_DEVICE\_ID\_SPRINTEK\_CORPORATION 8;

#define I2C\_DEVICE\_ID\_ESPROS\_PHOTONICS\_AG 9;

#define I2C\_DEVICE\_ID\_FUJITSU\_SEMICONDUCTOR 10;

#define I2C\_DEVICE\_ID\_FLIR 11;

#define I2C\_DEVICE\_ID\_O2MICRO 12;

#define I2C\_DEVICE\_ID\_ATMEL 13;

#define I2C\_DEVICE\_ID\_NONE 0xffff;

u16 part\_id;

u8 die\_revision;

};

**成员**

manufacturer\_id

0 - 4095，由 NXP 维护的数据库

part\_id

0 - 511，根据制造商而定

die\_revision

0 - 7，根据制造商而定

## struct i2c\_driver

代表一个 I2C 设备驱动程序

**定义**

struct i2c\_driver {

unsigned int class;

int (\*probe)(struct i2c\_client \*client, const struct i2c\_device\_id \*id);

void (\*remove)(struct i2c\_client \*client);

int (\*probe\_new)(struct i2c\_client \*client);

void (\*shutdown)(struct i2c\_client \*client);

void (\*alert)(struct i2c\_client \*client, enum i2c\_alert\_protocol protocol, unsigned int data);

int (\*command)(struct i2c\_client \*client, unsigned int cmd, void \*arg);

struct device\_driver driver;

const struct i2c\_device\_id \*id\_table;

int (\*detect)(struct i2c\_client \*client, struct i2c\_board\_info \*info);

const unsigned short \*address\_list;

struct list\_head clients;

bool disable\_i2c\_core\_irq\_mapping;

};

**成员**

class

我们要实例化什么样的 i2c 设备（用于检测）

probe

设备绑定回调 - 即将废弃

remove

设备取消绑定回调

probe\_new

用于设备绑定的新回调

shutdown

设备关闭回调

alert

警报回调，例如 SMBus 警报协议

command

总线广播信号回调（可选）

driver

设备驱动模型驱动程序

id\_table

由此驱动程序支持的 I2C 设备列表

detect

设备检测回调

address\_list

要探测的 I2C 地址（用于检测）

clients

我们创建的已检测客户端的列表（仅用于 i2c-core）

disable\_i2c\_core\_irq\_mapping

告诉 i2c-core 不要进行 IRQ 映射

**说明**

driver.owner 字段应设置为此驱动程序的模块所有者。driver.name 字段应设置为此驱动程序的名称。

对于自动设备检测，必须定义 detect 和 address\_list。否则，只能创建通过模块参数强制执行的设备。如果检测函数在成功检测时填充至少 i2c\_board\_info 结构的 name 字段并可能还填充 flags 字段。

如果 detect 丢失，则驱动程序仍将适用于enum设备。只是不支持找到的设备。这是针对许多无法可靠检测的 I2C/SMBus 设备以及实际上总是可以enum的设备所期望的。

传递到检测回调的 i2c\_client 结构并不是真正的 i2c\_client。它初始化足够，因此您可以在其上调用 i2c\_smbus\_read\_byte\_data 等函数。不要对它执行其他操作。特别是，不允许在其上调用 dev\_dbg 等调试函数。

## struct i2c\_client

代表一个 I2C 从设备

**定义**

struct i2c\_client {

unsigned short flags;

unsigned short addr;

char name[I2C\_NAME\_SIZE];

struct i2c\_adapter \*adapter;

struct device dev;

int irq;

struct list\_head detected;

#if IS\_ENABLED(CONFIG\_I2C\_SLAVE);

i2c\_slave\_cb\_t slave\_cb;

#endif;

};

**成员**

flags

请参阅 I2C\_CLIENT\_\* 获取可能的标志

addr

在连接到父适配器的 I2C 总线上使用的地址。

name

指示设备类型，通常是足够通用的芯片名称，隐藏第二来源和兼容修订版本。

adapter

管理托管此 I2C 设备的总线段

dev

从设备的驱动程序模型设备节点。

irq

指示此设备生成的 IRQ（如果有）

detected

在 i2c\_driver.clients 列表或 i2c-core 的 userspace\_devices 列表中的成员

slave\_cb

当使用适配器的 I2C 从机模式时的回调。适配器调用它将从机事件传递给从机驱动程序。

**说明**

一个 i2c\_client 标识连接到 I2C 总线上的单个设备（即芯片）。Linux 显露的行为由管理设备的驱动程序定义。

## struct i2c\_board\_info

设备创建的模板

**定义**

struct i2c\_board\_info {

char type[I2C\_NAME\_SIZE];

unsigned short flags;

unsigned short addr;

const char \*dev\_name;

void \*platform\_data;

struct device\_node \*of\_node;

struct fwnode\_handle \*fwnode;

const struct property\_entry \*properties;

const struct resource \*resources;

unsigned int num\_resources;

int irq;

};

**成员**

type

芯片类型，用于初始化 i2c\_client.name

flags

用于初始化 i2c\_client.flags

addr

存储在 i2c\_client.addr 中

dev\_name

如果设置，将覆盖默认的 <busnr>-<addr> dev\_name

platform\_data

存储在 i2c\_client.dev.platform\_data 中

of\_node

指向 OpenFirmware 设备节点的指针

fwnode

由平台固件提供的设备节点

properties

设备的其他属性

resources

与设备关联的资源

num\_resources

资源数组中的资源数

irq

存储在 i2c\_client.irq 中

**说明**

实际上，I2C 不支持硬件探测，尽管控制器和设备可能能够使用 I2C\_SMBUS\_QUICK 来告知是否存在给定地址的设备。驱动程序通常需要比这更多的信息，例如芯片类型、配置、相关 IRQ 等等。

i2c\_board\_info 用于建立列出已连接 I2C 设备信息的表格，这些信息用于扩展驱动程序模型树。对于主板，使用 i2c\_register\_board\_info() 来静态地完成这一过程；总线号用来标识尚未可用的适配器。对于附加板，i2c\_new\_client\_device() 使用已知的适配器来动态地完成此过程。

## I2C\_BOARD\_INFO

I2C\_BOARD\_INFO (dev\_type, dev\_addr)

宏用于列出 I2C 设备及其地址

**参数**

dev\_type

标识设备类型

dev\_addr

总线上设备的地址。

**说明**

此宏初始化struct i2c\_board\_info 的必要字段，声明了特定板上所提供的内容。可选字段（比如关联的 irq 或设备特定的平台数据）使用传统语法提供。

## struct i2c\_algorithm

代表 I2C 传输方法

**定义**

struct i2c\_algorithm {

int (\*master\_xfer)(struct i2c\_adapter \*adap, struct i2c\_msg \*msgs, int num);

int (\*smbus\_xfer) (struct i2c\_adapter \*adap, u16 addr,unsigned short flags, char read\_write, u8 command, int size, union i2c\_smbus\_data \*data);

u32 (\*functionality) (struct i2c\_adapter \*);

#if IS\_ENABLED(CONFIG\_I2C\_SLAVE);

int (\*reg\_slave)(struct i2c\_client \*client);

int (\*unreg\_slave)(struct i2c\_client \*client);

#endif;

};

**成员**

master\_xfer

使用 msgs 数组定义的给定 I2C 适配器向给定 I2C 适配器传输的一组 i2c 事务，有 num 条可用于传输。

smbus\_xfer

向给定的 I2C 适配器发出 smbus 事务。如果此项不存在，则总线层将尝试将 SMBus 调用转换为 I2C 传输。

functionality

返回 I2C\_FUNC\_\* 标志对应的此算法/适配器组支持的标志。

reg\_slave

注册给定客户机到该适配器的 I2C 从设备模式

unreg\_slave

从该适配器的 I2C 从设备模式中注销给定的客户机

**说明**

以下struct 适用于想要实现新的总线驱动程序的人i2c\_algorithm 是一类硬件解决方案的接口，它们可以使用相同的总线算法进行寻址，即使用比如模拟，并行总线控制器等。

master\_xfer{\_atomic} 字段返回的代码应指示传输期间发生的错误代码类型，如在内核文档文件 I2C/SMBUS 错误码中所述。否则，应返回执行的消息数。

## struct i2c\_lock\_operations

代表 I2C 锁操作

**定义**

struct i2c\_lock\_operations {

void (\*lock\_bus)(struct i2c\_adapter \*adapter, unsigned int flags);

int (\*trylock\_bus)(struct i2c\_adapter \*adapter, unsigned int flags);

void (\*unlock\_bus)(struct i2c\_adapter \*adapter, unsigned int flags);

};

**成员**

lock\_bus

获得对 I2C 总线段的独占访问

trylock\_bus

尝试获取对 I2C 总线段的独占访问

unlock\_bus

释放对 I2C 总线段的独占访问

**说明**

主要操作由 i2c\_lock\_bus 和 i2c\_unlock\_bus 包装。

## struct i2c\_timings

I2C 时序信息

**定义**

struct i2c\_timings {

u32 bus\_freq\_hz;

u32 scl\_rise\_ns;

u32 scl\_fall\_ns;

u32 scl\_int\_delay\_ns;

u32 sda\_fall\_ns;

u32 sda\_hold\_ns;

};

**成员**

bus\_freq\_hz

总线频率，单位 Hz

scl\_rise\_ns

SCL 信号上升所需时间，单位 ns；I2C 规范中的 tr

scl\_fall\_ns

SCL 信号下降所需时间，单位 ns；I2C 规范中的 tf

scl\_int\_delay\_ns

为了设置 SCL，IP 核需要的额外时间，单位 ns

sda\_fall\_ns

SDA 信号下降所需时间，单位 ns；I2C 规范中的 tf

sda\_hold\_ns

IP 核需要保持 SDA 的额外时间，单位 ns

**说明**

### struct i2c\_bus\_recovery\_info

I2C 总线恢复信息

**定义**

struct i2c\_bus\_recovery\_info {

int (\*recover\_bus)(struct i2c\_adapter \*adap);

int (\*get\_scl)(struct i2c\_adapter \*adap);

void (\*set\_scl)(struct i2c\_adapter \*adap, int val);

int (\*get\_sda)(struct i2c\_adapter \*adap);

void (\*set\_sda)(struct i2c\_adapter \*adap, int val);

int (\*get\_bus\_free)(struct i2c\_adapter \*adap);

void (\*prepare\_recovery)(struct i2c\_adapter \*adap);

void (\*unprepare\_recovery)(struct i2c\_adapter \*adap);

struct gpio\_desc \*scl\_gpiod;

struct gpio\_desc \*sda\_gpiod;

};

**成员**

recover\_bus

恢复例程。可以使用驱动程序的 recover\_bus() 例程，也可以使用 i2c\_generic\_scl\_recovery()。

get\_scl

获取当前 SCL 线的值。用于通用 SCL 恢复时强制执行。对于通用 GPIO 恢复，内部填充。

set\_scl

设置/清除 SCL 线。用于通用 SCL 恢复时强制执行。对于通用 GPIO 恢复，内部填充。

get\_sda

获取当前 SDA 线的值。对于通用 SCL 恢复，此项或 set\_sda() 必均须强制执行。对于通用 GPIO 恢复，如果 sda\_gpio 是有效的 GPIO，则在内部填充。

set\_sda

设置/清除 SDA 线。对于通用 SCL 恢复，此项或 get\_sda() 必均须强制执行。对于通用 GPIO 恢复，如果 sda\_gpio 是有效的 GPIO，则在内部填充。

get\_bus\_free

返回 IP 核从 I2C 中复杂一些，需要更多内部逻辑判断而不是仅读取 SDA 时所看到的总线空闲状态。是可选的。

prepare\_recovery

此函数将在启动恢复之前调用。平台可以在此处为 SDA/SCL 线路或其他他们想要的东西配置 padmux。

unprepare\_recovery

此函数将在恢复完成后调用。平台可以在此处为 SDA/SCL 线路或其他他们想要的东西配置 padmux。

scl\_gpiod

SCL 线的 gpiod。仅用于 GPIO 恢复。

sda\_gpiod

SDA线的gpiod。仅适用于GPIO恢复。

## struct i2c\_adapter\_quirks

说明i2c适配器的缺陷

**定义**

struct i2c\_adapter\_quirks {

u64 flags;

int max\_num\_msgs;

u16 max\_write\_len;

u16 max\_read\_len;

u16 max\_comb\_1st\_msg\_len;

u16 max\_comb\_2nd\_msg\_len;

};

**成员**

flags

可能的标志请参见I2C\_AQ\_ \*，并请阅读以下内容

max\_num\_msgs

每个传输的最大消息数

max\_write\_len

写入消息的最大长度

max\_read\_len

读取消息的最大长度

max\_comb\_1st\_msg\_len

组合消息中第一个msg的最大长度

max\_comb\_2nd\_msg\_len

组合消息中第二个msg的最大长度

**说明**

关于组合消息的说明一些I2C控制器只能在传输中发送一条消息，再加上称为组合消息或写入-读取的东西。这通常是一个小写消息，后跟一个读取消息，几乎足以访问基于寄存器的设备，如EEPROM。有一个标志来支持该模式。它意味着max\_num\_msg = 2，并且使用max\_comb\_\*\_len进行长度检查，因为组合消息模式通常具有自己的限制。由于硬件实现，一些控制器实际上可以执行写入-然后-任何其他变体。为了支持它，写-然后-读已分解为诸如先写后读之类的较小位，可以根据需要进行组合。

## void i2c\_lock\_bus(struct i2c\_adapter \*adapter, unsigned int flags)

获取对I2C总线段的独占访问权限

**参数**

struct i2c\_adapter \*adapter

目标I2C总线段

unsigned int flags

I2C\_LOCK\_ROOT\_ADAPTER锁定根I2C适配器，I2C\_LOCK\_SEGMENT仅锁定适配器树中的此分支

## int i2c\_trylock\_bus(struct i2c\_adapter \*adapter, unsigned int flags)

尝试获取对I2C总线段的独占访问权限

**参数**

struct i2c\_adapter \*adapter

目标I2C总线段

unsigned int flags

I2C\_LOCK\_ROOT\_ADAPTER尝试锁定根I2C适配器，I2C\_LOCK\_SEGMENT尝试仅锁定适配器树中的此分支

**返回**

如果锁定了I2C总线段，则为true；否则为false

## void i2c\_unlock\_bus(struct i2c\_adapter \*adapter, unsigned int flags)

释放对I2C总线段的独占访问权

**参数**

struct i2c\_adapter \*adapter

目标I2C总线段

unsigned int flags

I2C\_LOCK\_ROOT\_ADAPTER解锁根I2C适配器，I2C\_LOCK\_SEGMENT仅解锁适配器树中的此分支

## bool i2c\_check\_quirks(struct i2c\_adapter \*adap, u64 quirks)

用于检查i2c适配器中的怪癖标志的功能

**参数**

struct i2c\_adapter \*adap

i2c适配器

u64 quirks

怪癖标志

**返回**

如果适配器具有所有指定的异常标志，则为true；如果没有，则为false

## module\_i2c\_driver

module\_i2c\_driver(\_\_i2c\_driver)

用于注册模块化I2C驱动程序的助手宏

**参数**

\_\_i2c\_driver

i2c\_driver结构

**说明**

用于I2C驱动程序中不做任何特殊事情的助手宏。这消除了大量模板。每个模块只能使用此宏一次，并且调用它将替换module\_init（）和module\_exit（）

## builtin\_i2c\_driver

builtin\_i2c\_driver(\_\_i2c\_driver)

用于注册内置I2C驱动程序的助手宏

**参数**

\_\_i2c\_driver

i2c\_driver结构

**说明**

用于在其初始化中不做任何特殊事情的I2C驱动程序的助手宏。这消除了大量模板。每个驱动程序只能使用此宏一次，并且调用它将替换device\_initcall（）。

## int i2c\_register\_board\_info(int busnum, struct i2c\_board\_info const \*info, unsigned len)

静态声明I2C设备

**参数**

int busnum

标识这些设备所属的总线

struct i2c\_board\_info const \*info

i2c设备说明符的向量

unsigned len

向量中有多少说明符；可以为零以保留指定的总线号。

**说明**

使用Linux I2C驱动程序堆栈的系统可以在初始化时声明板信息表。这应该在与arch\_initcall（）时间相近的特定于板的init代码中完成，或者在注册任何I2C适配器驱动程序之前完成。例如，主板init代码可以定义多个设备，每个堆栈中的板子卡的init代码也可以定义多个设备。

I2C设备将在相关总线的适配器注册后创建。在那之后，使用标准驱动程序模型工具将“新样式”I2C驱动程序绑定到设备上。使用此例程声明的任何设备的总线号均不可用于动态分配。

传递的板信息可以安全地\_\_initdata，但请注意嵌入式指针（用于platform\_data、函数等），因为它们不会被复制。

## struct i2c\_client \*i2c\_verify\_client(struct device \*dev)

返回参数作为i2c\_client，或为NULL

**参数**

struct device \*dev

设备，可能来自某个驱动程序模型迭代器

**说明**

当遍历驱动程序模型树时，可能使用驱动程序模型迭代器（如device\_for\_each\_child（））找到的节点不能假设太多。使用此函数可以避免由错误地将某些非I2C设备视为i2c\_client而引起的oops。

## struct i2c\_client \*i2c\_new\_client\_device(struct i2c\_adapter \*adap, struct i2c\_board\_info const \*info)

实例化i2c设备

**参数**

struct i2c\_adapter \*adap

管理设备的适配器

struct i2c\_board\_info const \*info

说明一个I2C设备；忽略bus\_num

创建一个i2c设备。绑定通过驱动程序模型的probe()/remove()方法处理。当我们从此函数返回时，驱动程序可以绑定到此设备，或者在以后的任何时刻（例如，可能会热插拔装载驱动程序模块）。此调用不适用于主板初始化逻辑，后者通常在arch\_initcall()运行之前很久就会运行，此时可能还不存在任何i2c\_adapter。

这将返回新的i2c客户端，其中可保存i2c\_unregister\_device()的后续使用；或者返回ERR\_PTR来说明错误。

## void i2c\_unregister\_device(struct i2c\_client \*client)

反向效果为i2c\_new\_\*\_device()

**参数**

struct i2c\_client \*client

从i2c\_new\_\*\_device()返回的值

**上下文**

可以睡眠

## struct i2c\_client \*i2c\_new\_dummy(struct i2c\_adapter \*adapter, u16 address)

返回绑定到虚拟驱动程序的新i2c设备

**参数**

struct i2c\_adapter \*adapter

管理设备的适配器

u16 address

要使用的七位地址

**上下文**

可以睡眠

**说明**

此返回绑定到“虚拟”驱动程序的i2c客户端，用于占用多个地址的设备。此类芯片的示例包括各种EEPROM（如24c04和24c08型号）。

这些虚拟设备有两个主要用途。首先，大多数I2C和SMBus调用，除了i2c\_transfer()需要客户端句柄；虚拟设备将是该句柄。其次，这可以防止将指定地址绑定到其他驱动程序。

这将返回新的i2c客户端，其中可保存i2c\_unregister\_device()的后续使用；或者返回ERR\_PTR来说明错误。

## struct [i2c\_client](https://www.kernel.org/doc/html/v4.19/driver-api/i2c.html" \l "c.i2c_client" \o "i2c_client) \* i2c\_new\_secondary\_device(struct [i2c\_client](https://www.kernel.org/doc/html/v4.19/driver-api/i2c.html" \l "c.i2c_client" \o "i2c_client) \* client, const char \* name, u16 default\_addr)

帮助程序以获取实例化的次要地址并创建相关设备

**参数**

struct i2c\_client \*client

用于主设备的句柄

const char \* name

用于指定要获取的次要地址的句柄

u16 default\_addr

如果没有指定次要地址，则用作回退

**上下文**

可以睡眠

**说明**

I2C客户端可以由多个绑在一起的I2C从设备组成。然后，I2C客户端驱动程序将绑定到主I2C从设备，并需要创建I2C虚拟客户端来与所有其他从设备进行通信。

此函数将创建并返回一个I2C虚拟客户端，其I2C地址从基于给定从设备名称的平台固件中检索。如果固件未指定地址，则使用default\_addr。

在基于DT的平台上，该地址是从“reg-names”值与从设备名称匹配的“reg”属性条目单元格中检索的。

这将返回新的i2c客户端，其中可保存i2c\_unregister\_device()的后续使用；或者返回NULL表示错误。

## struct i2c\_adapter \*i2c\_verify\_adapter(struct device \*dev)

返回参数为i2c\_adapter或NULL

**参数**

struct device \*dev

设备，可能来自某些驱动程序模型迭代器

**说明**

当遍历驱动程序模型树时，可能会使用驱动程序模型迭代器，例如device\_for\_each\_child()，您不能假设您发现的节点非常多。使用此函数可避免由于错误地将某些非I2C设备视为i2c\_adapter而导致崩溃。

## int i2c\_handle\_smbus\_host\_notify(struct i2c\_adapter \*adap, unsigned short addr)

转发Host Notify事件到正确的I2C客户端。

**参数**

struct i2c\_adapter \* adap

适配器

unsigned short addr

通知设备的I2C地址

**上下文**

可以睡眠

**说明**

帮助函数从I2C总线驱动程序的中断处理程序中调用。它将安排Host Notify IRQ。

## int i2c\_add\_adapter(struct i2c\_adapter \*adapter)

声明i2c适配器，使用动态总线号

**参数**

struct i2c\_adapter \*adapter

要添加的适配器

**上下文**

可以睡眠

**说明**

此例程用于在总线号不重要或总线号由dt别名指定时声明I2C适配器。动态添加到USB链路或PCI插件卡的I2C适配器是其总线号不重要的例子。

当它返回零时，新的总线号已分配并存储在adap->nr中，并且指定的适配器变为可用于客户端。否则，返回负的errno值。

## int i2c\_add\_numbered\_adapter(struct i2c\_adapter \*adap)

声明i2c适配器，使用静态总线号

**参数**

struct i2c\_adapter \*adap

要注册的适配器（已初始化adap->nr）

**上下文**

可以睡眠

**说明**

当总线号很重要时，使用此例程声明I2C适配器。例如，对于来自系统级芯片的I2C适配器，或者内置于系统主板中的适配器，并且使用i2c\_board\_info来正确配置I2C设备。

如果请求的总线号设置为-1，则此函数将表现得与i2c\_add\_adapter相同，并将动态分配总线号。

如果没有为此总线预先声明任何设备，请确保在任何动态分配的设备之前注册适配器。否则，所需的总线ID可能不可用。

当返回为零时，指定的适配器就可以提供给在adap->nr中提供的总线编号使用的客户端。此外，使用i2c\_register\_board\_info()注册预声明的I2C设备表并扫描，创建适当的驱动模型设备节点。否则，返回负的errno值。

## void i2c\_del\_adapter(struct i2c\_adapter \*adap)

注销I2C适配器

**参数**

struct i2c\_adapter \*adap

被注销的适配器

**上下文**

可能休眠

**说明**

这将注销之前由i2c\_add\_adapter或i2c\_add\_numbered\_adapter注册的I2C适配器。

## void i2c\_parse\_fw\_timings(struct device \*dev, struct i2c\_timings \*t, bool use\_defaults)

从固件中获取I2C相关的时序参数

**参数**

struct device \*dev

用于扫描I2C时序属性的设备

struct i2c\_timings \*t

被填充的i2c\_timingsstruct 值

bool use\_defaults

是否使用从I2C规范派生出来的良好缺省值，否则不更新

**说明**

扫描设备以获取用于说明信号时序参数的通用I2C属性，并使用结果填充给定的struct 。如果未找到某个属性并且use\_defaults为true，则假定最大时序，该时序基于I2C规范。如果不使用use\_defaults，则结果与之前相同，因此驱动程序可以在调用此助手之前应用其自己的缺省值。后者主要用于避免现有驱动程序的返回，这些驱动程序想要转换到这个函数。新驱动程序几乎总是要使用缺省值。

## void i2c\_del\_driver(struct i2c\_driver \*driver)

注销I2C驱动程序

**参数**

struct i2c\_driver \*driver

被注销的驱动程序

**上下文**

可能休眠

## struct i2c\_client \*i2c\_use\_client(struct i2c\_client \*client)

增加i2c客户端结构的引用计数

**参数**

struct i2c\_client \*client

被引用的客户端

**说明**

每个对客户端的活动引用都应是引用计数的。驱动程序模型自动完成这个过程作为驱动程序绑定的一部分，所以大多数驱动程序不需要显式完成这个操作它们一直持有一个引用，直到从设备上解除绑定。

返回一个指向增加引用计数的客户端的指针。

## void i2c\_release\_client(struct i2c\_client \*client)

释放对i2c客户端结构的引用

**参数**

struct i2c\_client \*client

不再引用的客户端

**说明**

当客户端的使用者完成对它的使用时必须调用它。

## int \_\_i2c\_transfer(struct i2c\_adapter \*adap, struct i2c\_msg \*msgs, int num)

未锁定的i2c\_transfer

**参数**

struct i2c\_adapter \*adap

I2C总线句柄

struct i2c\_msg \*msgs

在终止操作之前执行一个或多个消息；每个消息都以一个START开头。

int num

要执行的消息数。

**说明**

返回负的errno，否则返回执行的消息数。

在调用此函数时必须持有适配器锁定。不进行调试日志记录。不会检查adap->algo->master\_xfer的存在。

## int i2c\_transfer(struct i2c\_adapter \*adap, struct i2c\_msg \*msgs, int num)

执行单个或组合的I2C消息

**参数**

struct i2c\_adapter \*adap

I2C总线句柄

struct i2c\_msg \*msgs

在终止操作之前执行一个或多个消息；每个消息都以一个START开头。

int num

要执行的消息数。

**说明**

返回负的errno，否则返回执行的消息数。

请注意，不要求每个消息都发送到相同的从地址，尽管这是最常见的模型。

## int i2c\_transfer\_buffer\_flags(const struct i2c\_client \*client, char \*buf, int count, u16 flags)

发出单个I2C消息，将数据传输到/从缓冲区

**参数**

const struct i2c\_client \*client

从设备句柄

char \*buf

数据存储的位置

int count

要传输的字节数，必须小于64k，因为msg.len是u16

u16 flags

消息使用的标记，例如读取的I2C\_M\_RD

**说明**

返回负的errno，否则返回传输的字节数。

## int i2c\_get\_device\_id(const struct i2c\_client \*client, struct i2c\_device\_identity \*id)

获取设备的制造商，部件ID和芯片修订版本

**参数**

const struct i2c\_client \*client

要查询的设备

struct i2c\_device\_identity \*id

查询的信息

**说明**

失败时**返回**负的errno，成功时**返回**零。

u8 \*i2c\_get\_dma\_safe\_msg\_buf(struct i2c\_msg \*msg, unsigned int threshold)

获取给定i2c\_msg的DMA安全缓冲区

**参数**

struct i2c\_msg \*msg

要检查的消息

unsigned int threshold

对于使用DMA有意义的最小字节数。至少应为1。

**返回**

如果未获得DMA安全缓冲区，则返回NULL。使用PIO的msg->buf。

或一个有效的指针，可用于DMA。使用后，通过调用i2c\_put\_dma\_safe\_msg\_buf()释放它。

**说明**

此函数仅在进程上下文中调用！

## void i2c\_put\_dma\_safe\_msg\_buf(u8 \*buf, struct i2c\_msg \*msg, bool xferred)

释放DMA安全缓冲区并与i2c\_msg同步

**参数**

u8 \*buf

从i2c\_get\_dma\_safe\_msg\_buf()获得的缓冲区。可能为空。

struct i2c\_msg \*msg

缓冲区对应的消息

bool xferred

消息是否被传输

## s32 i2c\_smbus\_read\_byte (const struct i2c\_client \*client)

SMBus“接收字节”规范

**参数**

const struct i2c\_client \*client

从设备的句柄

**说明**

这执行SMBus“接收字节”协议，返回负的errno，否则从设备接收到的字节。

## s32 i2c\_smbus\_write\_byte（const struct i2c\_client \*client，u8 value）

SMBus“发送字节”规范

**参数**

const struct i2c\_client \*client

从设备的句柄

u8 value

要发送的字节

**说明**

这执行SMBus“发送字节”协议，返回负的errno，否则成功返回零。

## s32 i2c\_smbus\_read\_byte\_data（const struct i2c\_client \*client，u8 command）

SMBus“读取字节”规范

**参数**

const struct i2c\_client \*client

从设备的句柄

u8 command

从设备解释的字节

**说明**

这执行SMBus“读取字节”协议，返回负的errno，否则从设备接收到的数据字节。

## s32 i2c\_smbus\_write\_byte\_data（const struct i2c\_client \*client，u8 command，u8 value）

SMBus“写字节”规范

**参数**

const struct i2c\_client \*client

从设备的句柄

u8 command

从设备解释的字节

u8 value

正在写入的字节

**说明**

这执行SMBus“写字节”协议，返回负的errno，否则成功返回零。

## s32 i2c\_smbus\_read\_word\_data（const struct i2c\_client \*client，u8 command）

SMBus“读取字”规范

**参数**

const struct i2c\_client \*client

从设备的句柄

u8 command

从设备解释的字节

**说明**

这执行SMBus“读取字”协议，返回负的errno，否则从设备接收到的16位无符号“字”。

## s32 i2c\_smbus\_write\_word\_data（const struct i2c\_client \*client，u8 command，u16 value）

SMBus“写字”规范

**参数**

const struct i2c\_client \*client

从设备的句柄

u8 command

从设备解释的字节

u16 value

正在写入的16位“字”

**说明**

这执行SMBus“写字”协议，返回负的errno，否则成功返回零。

## s32 i2c\_smbus\_read\_block\_data（const struct i2c\_client \*client，u8 command，u8 \*values）

SMBus“块读取”规范

**参数**

const struct i2c\_client \*client

从设备的句柄

u8 command

从设备解释的字节

u8 \*values

数据将被读入的字节数组；足够大以容纳从设备返回的数据。 SMBus最多允许32个字节。

**说明**

这执行SMBus“块读取”协议，返回负的errno，否则从从设备读取的数据字节数。

请注意，使用此函数需要客户端的适配器支持I2C\_FUNC\_SMBUS\_READ\_BLOCK\_DATA功能。并非所有适配器驱动程序都支持此功能；其通过I2C消息传递的仿真依赖于一个特定的机制（I2C\_M\_RECV\_LEN），可能没有实现。

## s32 i2c\_smbus\_write\_block\_data（const struct i2c\_client \*client，u8 command，u8 length，const u8 \*values）

SMBus“块写入”规范

**参数**

const struct i2c\_client \*client

从设备的句柄

u8 command

从设备解释的字节

u8 length

数据块的大小； SMBus最多允许32个字节

const u8 \*values

将被写入的字节数组。

**说明**

这执行SMBus“块写入”协议，返回负的errno，否则成功返回零。

## s32 i2c\_smbus\_xfer（struct i2c\_adapter \*adapter，u16 addr，unsigned short flags，char read\_write，u8 command，int protocol，union i2c\_smbus\_data \*data）

执行SMBus协议操作

**参数**

struct i2c\_adapter \*adapter

I2C总线的句柄

u16 addr

该总线上的SMBus从设备地址

unsigned short flags

I2C\_CLIENT\_ \*标志（通常为零或I2C\_CLIENT\_PEC）

char read\_write

I2C\_SMBUS\_READ或I2C\_SMBUS\_WRITE

u8 command

从设备解释的字节，用于使用这些字节的协议

int protocol

要执行的SMBus协议操作，例如I2C\_SMBUS\_PROC\_CALL

union i2c\_smbus\_data \*data

要读取或写入的数据

**说明**

这执行SMBus协议操作，并返回负的errno代码，否则成功返回零。

## s32 i2c\_smbus\_read\_i2c\_block\_data\_or\_emulated（const struct i2c\_client \*client，u8 command，u8 length，u8 \*values）

读取块或模拟

**参数**

const struct i2c\_client \*client

从设备的句柄

u8 command

从设备解释的字节

u8 length

数据块的大小；SMBus允许最多I2C\_SMBUS\_BLOCK\_MAX个字节

u8 \*values

数据将被读入的字节数组；足够大以容纳从设备返回的数据。 SMBus允许最多I2C\_SMBUS\_BLOCK\_MAX个字节。

**说明**

如果适配器支持，这执行SMBus“块读取”协议。如果不支持块读取，则根据可用性使用字或字节读取协议进行模拟。

使用此函数访问的I2C从设备的地址必须映射到线性区域，以便块读取具有与字节读取相同的效果。在使用此函数之前，必须再次检查I2C从设备是否支持用字节传输交换块传输。

## struct i2c\_client \* i2c\_setup\_smbus\_alert(struct i2c\_adapter \* adapter, struct i2c\_smbus\_alert\_setup \* setup)

设置SMBus警报支持

**参数**

struct i2c\_adapter \* adapter

目标适配器

struct i2c\_smbus\_alert\_setup \* setup

SMBus警报处理程序的设置数据

**上下文**

可以睡觉

**说明**

设置处理给定I2C总线段上的SMBus警报协议。可以通过我们的IRQ处理程序或适配器来处理（从其处理程序，定期轮询或其他方式）。请注意，如果管理IRQ，我们必须知道它是电平触发还是边沿触发，以便将其正确地传递给工作队列。如果触发警报似乎使系统发生死锁，则应该说它是电平触发。这将返回ara客户端，应将其保存以供稍后使用i2c\_handle\_smbus\_alert（）和最终i2c\_unregister\_device（）；或ERRPTR以指示错误。

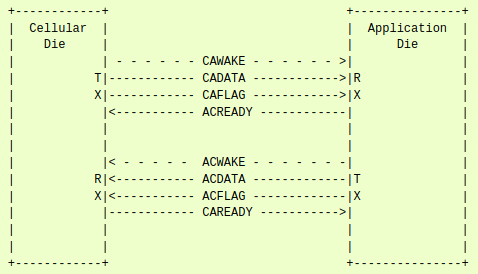
# 高速同步串行接口 (HSI)

## 介绍

高速同步接口（HSI）是一种全双工、低延迟的协议，针对应用处理器和基带芯片之间的芯片级互连进行了优化。它由MIPI联盟在2003年规定，自那以后，已由多个供应商实现。

HSI接口支持多通道（通常为8个）的全双工通信，并能够达到高达200 Mbit/s的速度。串行协议

使用两个信号DATA和FLAG作为组合数据和时钟信号，以及一个额外的READY信号进行流控制。额外的WAKE信号可以用于从待机模式中唤醒芯片。信号通常带有AC的前缀，用于从应用芯片到蜂窝芯片的信号，以及CA用于沿另一方向传输的信号。



## 在Linux中的HSI子系统

在Linux内核中，hsi子系统应用于HSI设备。hsi子系统包含hsi控制器的驱动程序，包括对多端口控制器的支持，并提供用于使用HSI端口的通用API。它还包含HSI客户端驱动程序，这些客户端驱动程序利用通用API来实现在HSI接口上使用的协议。这些客户端驱动程序可以使用任意数量的通道。

## hsi-char设备

每个端口都会自动注册一个名为hsi\_char的通用客户端驱动程序，为用户空间提供表示HSI端口的字符设备。它可以用于通过用户空间进行HSI通信。用户空间可以使用以下ioctl命令配置hsi\_char设备

HSC\_RESET

刷新HSI端口

HSC\_SET\_PM

启用或禁用客户端。

HSC\_SEND\_BREAK

发送中断

HSC\_SET\_RX

设置RX配置

HSC\_GET\_RX

获取RX配置

HSC\_SET\_TX

设置TX配置

HSC\_GET\_TX

获取TX配置

## 内核HSI API

### struct hsi\_channel

hsi客户端使用的通道资源

**定义**

struct hsi\_channel {

unsigned int id;

const char \*name;

};

**成员**

id

通道号

name

通道名称

### struct hsi\_config

RX / TX HSI模块的配置

**定义**

struct hsi\_config {

unsigned int mode;

struct hsi\_channel \*channels;

unsigned int num\_channels;

unsigned int num\_hw\_channels;

unsigned int speed;

union {

unsigned int flow;

unsigned int arb\_mode;

};

};

**成员**

mode

位传输模式（流或帧）

channels

客户端使用的通道资源

num\_channels

通道资源数

num\_hw\_channels

传输器配置的通道数[1..16]

speed

最大位传输速度（Kbit/s）

{unnamed\_union}

匿名联合体

flow

RX流类型（同步或流水线）

arb\_mode

TX帧的仲裁模式（轮流，优先级）

### struct hsi\_board\_info

HSI客户端板信息

**定义**

struct hsi\_board\_info {

const char \*name;

unsigned int hsi\_id;

unsigned int port;

struct hsi\_config tx\_cfg;

struct hsi\_config rx\_cfg;

void \*platform\_data;

struct dev\_archdata \*archdata;

};

**成员**

name

HSI设备名称

hsi\_id

客户端所在的HSI控制器ID

port

客户端所在控制器中的端口号

tx\_cfg

HSI TX配置

rx\_cfg

HSI RX配置

platform\_data

平台相关数据

archdata

体系结构相关的设备数据

### struct hsi\_client

附加到HSI端口的HSI客户端

**定义**

struct hsi\_client {

struct device device;

struct hsi\_config tx\_cfg;

struct hsi\_config rx\_cfg;

};

**成员**

device

设备的驱动程序模型表示

tx\_cfg

HSI TX配置

rx\_cfg

HSI RX配置

### struct hsi\_client\_driver

与HSI客户端关联的驱动程序

**定义**

struct hsi\_client\_driver {

struct device\_driver driver;

};

**成员**

driver

驱动程序的驱动程序模型表示

### struct hsi\_msg

HSI消息说明符

**定义**

struct hsi\_msg {

struct list\_head link;

struct hsi\_client \*cl;

struct sg\_table sgt;

void \*context;

void (\*complete)(struct hsi\_msg \*msg);

void (\*destructor)(struct hsi\_msg \*msg);

int status;

unsigned int actual\_len;

unsigned int channel;

unsigned int ttype:1;

unsigned int break\_frame:1;

};

**成员**

link

由当前**说明**符所有者自由使用

cl

发出传输的HSI设备客户端

sgt

散列表数组的头

**context**

与传输相关联的客户端**上下文**数据

complete

传输完成回调

destructor

在刷新时释放资源的销毁者

status

传输完成时的状态

actual\_len

完成时传输的实际数据长度

channel

TX / RX消息的通道

ttype

传输类型（如果设置了TX，则为TX，否则为RX）

break\_frame

如果为true，则HSI将发送/接收断帧。在请求中，数据缓冲区将被忽略。

### struct hsi\_port

HSI端口设备

**定义**

struct hsi\_port {

struct device device;

struct hsi\_config tx\_cfg;

struct hsi\_config rx\_cfg;

unsigned int num;

unsigned int shared:1;

int claimed;

struct mutex lock;

int (\*async)(struct hsi\_msg \*msg);

int (\*setup)(struct hsi\_client \*cl);

int (\*flush)(struct hsi\_client \*cl);

int (\*start\_tx)(struct hsi\_client \*cl);

int (\*stop\_tx)(struct hsi\_client \*cl);

int (\*release)(struct hsi\_client \*cl);

struct blocking\_notifier\_head n\_head;

};

**成员**

device

设备的驱动模型表示

tx\_cfg

当前的TX路径配置

rx\_cfg

当前的RX路径配置

num

端口号

shared

当端口可以被不同的客户共享时设置

claimed

声明端口的客户的引用计数

lock

串行化端口声明

async

异步传输回调

setup

设置HSI客户端配置的回调

flush

清理硬件状态并销毁所有未决传输的回调

start\_tx

通知客户端要TX数据的回调

stop\_tx

通知客户端不再想TX数据的回调

release

通知客户端不再使用该端口的回调

n\_head

用于向客户端发出端口事件的通知器链。

### struct hsi\_controller

HSI控制器设备

**定义**

struct hsi\_controller {

struct device device;

struct module \*owner;

unsigned int id;

unsigned int num\_ports;

struct hsi\_port \*\*port;

};

**成员**

device

设备的驱动模型表示

owner

指向拥有该控制器的模块的指针

id

HSI控制器ID

num\_ports

HSI控制器中的端口数

port

HSI端口数组

### unsigned int hsi\_id(struct hsi\_client \*cl)

获取与客户端关联的HSI控制器ID

**参数**

struct hsi\_client \*cl

HSI客户端的指针

**说明**

返回客户端连接的控制器ID

### unsigned int hsi\_port\_id(struct hsi\_client \*cl)

获取客户端连接的端口号

**参数**

struct hsi\_client \*cl

HSI客户端的指针

**说明**

返回与客户端关联的端口号

### int hsi\_setup(struct hsi\_client \*cl)

配置客户端的端口

**参数**

struct hsi\_client \*cl

HSI客户端的指针

**说明**

在共享端口时，客户端应该依靠单个客户端安装或为所有客户端提供相同的安装。

返回失败的-errno，成功的0

### int hsi\_flush(struct hsi\_client \*cl)

刷新客户端端口上的所有挂起事务

**参数**

struct hsi\_client \*cl

HSI客户端的指针

**说明**

此函数将销毁端口中的所有挂起hsi\_msg，并重置HW端口，使其准备好从干净状态接收和传输。

返回失败的-errno，成功的0

### int hsi\_async\_read(struct hsi\_client \*cl, struct hsi\_msg \*msg)

提交一个读取传输

**参数**

struct hsi\_client \*cl

HSI客户端的指针

struct hsi\_msg \*msg

传输的HSI消息说明符

**说明**

返回失败的-errno，成功的0

### int hsi\_async\_write(struct hsi\_client \*cl, struct hsi\_msg \*msg)

提交写入传输

**参数**

struct hsi\_client \*cl

HSI客户端的指针

struct hsi\_msg \*msg

传输的HSI消息说明符

**说明**

返回失败的-errno，成功的0

### int hsi\_start\_tx(struct hsi\_client \*cl)

通知端口客户端想要启动TX

**参数**

struct hsi\_client \*cl

HSI客户端的指针

**说明**

返回失败的-errno，成功的0

### int hsi\_stop\_tx(struct hsi\_client \*cl)

通知端口客户端不再想要传输

**参数**

struct hsi\_client \*cl

HSI客户端的指针

**说明**

返回失败的-errno，成功的0

### void hsi\_port\_unregister\_clients(struct hsi\_port \*port)

注销HSI端口

**参数**

struct hsi\_port \*port

要注销的HSI端口

### void hsi\_unregister\_controller(struct hsi\_controller \*hsi)

注销HSI控制器

**参数**

struct hsi\_controller \*hsi

要注册的HSI控制器

### int hsi\_register\_controller(struct hsi\_controller \*hsi)

注册HSI控制器及其端口

**参数**

struct hsi\_controller \*hsi

要注册的HSI控制器

**说明**

返回失败的-errno，成功的0。

### int hsi\_register\_client\_driver(struct hsi\_client\_driver \*drv)

将HSI客户端注册到HSI总线

**参数**

struct hsi\_client\_driver \*drv

要注册的HSI客户端驱动程序

**说明**

返回失败的-errno，成功的0。

### void hsi\_put\_controller(struct hsi\_controller \*hsi)

释放HSI控制器

**参数**

struct hsi\_controller \*hsi

要释放的HSI控制器的指针

**说明**

只有当HSI控制器驱动程序需要在成功调用hsi\_register\_controller之前释放其分配的hsi\_controller结构时，才应使用此函数。 其他使用是不允许的。

### struct hsi\_controller \*hsi\_alloc\_controller(unsigned int n\_ports, gfp\_t flags)

分配HSI控制器及其端口

**参数**

unsigned int n\_ports

HSI控制器上的端口数

gfp\_t flags

内核分配标志

**说明**

失败返回NULL或成功返回指向hsi\_controller的指针。

### void hsi\_free\_msg(struct hsi\_msg \*msg)

释放HSI消息

**参数**

struct hsi\_msg \*msg

要释放的HSI消息的指针

**说明**

客户端负责释放散列列表所指向的缓冲区。

### struct hsi\_msg \*hsi\_alloc\_msg(unsigned int nents, gfp\_t flags)

分配HSI消息

**参数**

unsigned int nents

内存条目数量

gfp\_t flags

内核分配标志

**说明**

nents可以为0。这主要针对读取传输。在这种情况下，HSI驱动程序会在读取数据时调用完整的回调函数，但不会消耗数据。

失败时返回NULL，成功时返回指向hsi\_msg的指针。

### int hsi\_async(struct hsi\_client \*cl, struct hsi\_msg \*msg)

向控制器提交HSI传输

**参数**

struct hsi\_client \*cl

发送传输的HSI客户端

struct hsi\_msg \*msg

传递给控制器的HSI传输

**说明**

HSI消息必须在先设定通道，ttype，complete和destructor字段。如果nents> 0，则客户端还必须初始化scatterlists，以指向要写入或从中读取的缓冲区。

HSI控制器依赖于其客户端预分配的缓冲区，而不是自行分配缓冲区。

一旦HSI消息传输完成，HSI控制器将使用HSI消息更新状态和actual\_len字段调用完整的回调函数。在从hsi\_async返回之前，可以调用完整的回调函数。

失败时返回-errno，成功时返回0

### int hsi\_claim\_port(struct hsi\_client \*cl, unsigned int share)

声明HSI客户端的端口

**参数**

struct hsi\_client \*cl

要声明其端口的HSI客户端

unsigned int share

指示客户端是否希望共享端口的标志。

**说明**

失败时返回-errno，成功时返回0。

### void hsi\_release\_port(struct hsi\_client \*cl)

释放HSI客户端的端口

**参数**

struct hsi\_client \*cl

先前声明其端口的HSI客户端

### int hsi\_register\_port\_event(struct hsi\_client \*cl, void (\*handler)(struct hsi\_client\*, unsigned long))

注册客户端以接收端口事件

**参数**

struct hsi\_client \*cl

希望接收端口事件的HSI客户端

void (\*handler)(struct hsi\_client \*, unsigned long)

事件处理程序回调

**说明**

客户端应注册回调以能够从端口接收事件。注册应在声明端口后进行。处理程序可以在中断上下文中调用。

发生错误时返回-errno，成功时返回0。

### int hsi\_unregister\_port\_event(struct hsi\_client \*cl)

停止为客户端接收端口事件

**参数**

struct hsi\_client \*cl

希望停止接收端口事件的HSI客户端

**说明**

客户端应在释放关联端口之前调用此函数。

发生错误时返回-errno，成功时返回0。

### int hsi\_event(struct hsi\_port \*port, unsigned long event)

通知客户端有关端口事件

**参数**

struct hsi\_port \*port

事件发生的端口

unsigned long event

事件类型

**说明**

客户端不应关心唤醒线行为。但是，由于HSI硬件协议中存在竞争条件，客户端需要被通知唤醒线变化，以便它们可以实现解决方法。

事件HSI\_EVENT\_START\_RX-传入唤醒线高HSI\_EVENT\_STOP\_RX-传入唤醒线下

发生错误时返回-errno，成功时返回0。

### int hsi\_get\_channel\_id\_by\_name(struct hsi\_client \*cl, char \*name)

通过通道名称获取通道ID

**参数**

struct hsi\_client \*cl

使用通道的HSI客户端

char \*name

通道所知的名称

**说明**

客户端可以调用此函数以获取类似于按名称请求IRQ或GPIO的HSI通道ID。此函数假定RX和TX使用相同的通道配置。

发生错误时返回-errno或通道ID成功。

# 错误检测和纠正（EDAC）设备

## 在EDAC子系统中使用的主要概念

有些事情并不明显，例如插座、\*插座集、银行、行、芯片选择行、通道等等，需要注意。

这些是许多术语，它们不总是意味着人们认为的意思（难以置信！）。为了创建一个讨论的公共基础，将建立术语及其定义。

内存设备

内存条上的单个DRAM芯片。这些设备通常每个输出4和8位（x4，x8）。在并行组合多个设备时，它们提供了内存控制器所需的位数：通常为72位，以提供64位+ 8位的ECC数据。

内存条

并行聚合多个内存设备的印刷电路板。通常情况下，在发生过多错误的情况下，它是可替换单元（FRU）并被替换。在大多数情况下，它也被称为DIMM（双列内存模块）。

内存插座

主板上的物理连接器，可接受单个内存条。在几个数据表中也被称为“插槽”。

通道

内存控制器通道，负责与一组DIMM通信。每个通道具有其自己独立的控制（命令）和数据总线，并可以独立使用或与其他通道分组使用。

分支

它通常是完全缓冲的 DIMM 内存控制器中的最高层次。通常，它包含两个通道。同一分支中的两个通道可以在单模式或锁步模式下使用。启用锁步时，高速缓存行会加倍，但通常会带来一些性能损失。此外，在出现错误时通常不可能仅指向一个内存条，因为纠错码是使用两个 DIMM 而不是一个计算的。因此，它能够比单模式更多地纠正错误。

单通道

内存控制器访问的数据仅包含在一个 DIMM 中。例如，如果数据宽度为 64 位，数据将使用一个 64 位并行访问流向 CPU。通常与 SDR、DDR、DDR2 和 DDR3 存储器一起使用。FB-DIMM 和 RAMBUS 使用不同的通道概念，因此该概念在这里不适用。

双通道

内存控制器访问的数据大小被交错到两个 DIMM 中，同时访问。例如，如果 DIMM 宽度为 64 位（带 ECC 为 72 位），则数据将使用 128 位并行访问流向 CPU。

芯片选择行

这是用于选择要访问的 DRAM 等级的 DRAM 信号的名称。单通道的常见芯片选择行为 64 位，双通道为 128 位。它可能无法被内存控制器看到，因为一些 DIMM 类型具有可以隐藏从内存控制器直接访问它的存储器缓冲区。

单排内存条

单个内存条有 1 个芯片选择行。主板通常将两个芯片选择引脚驱动到内存条。单个排名的内存条只会占用其中一个行。另一个不会使用。

双排内存条

双排内存条有两个芯片选择行，可以访问不同的内存设备组。两行不能同时访问。

双面内存条

已弃用的术语，见双排内存条。

双面内存条有两个芯片选择行，可以访问不同的内存设备组。两行不能同时访问。“双面”与在内存条的两侧安装的存储器设备无关。

插座集合

所需的所有内存条，以进行单个内存访问或由一个芯片选择行跨越的所有内存条。单个插座集合有两个芯片选择行，如果使用双面内存条，则这些内存条将占用这些芯片选择行。

银行

此术语应避免使用，因为需要区分芯片选择行和插座集合时并不清楚。

## 内存控制器

大多数 EDAC 核心都专注于进行内存控制器错误检测。edac\_mc\_alloc()。它使用内部的 struct mem\_ctl\_info 来说明内存控制器，该结构是EDAC驱动程序的一个不透明结构。只有 EDAC 核心被允许触摸它。

### enum dev\_type

说明所使用内存的 DRAM 芯片类型

**常量**

DEV\_UNKNOWN

无法确定，或 MC 不支持检测

DEV\_X1

1 位数据

DEV\_X2

2 位数据

DEV\_X4

4 位数据

DEV\_X8

8 位数据

DEV\_X16

16 位数据

DEV\_X32

32 位数据

DEV\_X64

64 位数据

**说明**

典型值为 x4 和 x8。

enum hw\_event\_mc\_err\_type

检测到的错误类型

**常量**

HW\_EVENT\_ERR\_CORRECTED

已纠正错误 - 表示检测到了一种经过 ECC 纠正的错误

HW\_EVENT\_ERR\_UNCORRECTED

未纠正错误 - 表示一种不能通过 ECC 纠正但不致命的错误（可能在未使用的存储区域上，或者内存控制器可以从错误中恢复，例如，通过重试操作）。

HW\_EVENT\_ERR\_DEFERRED

延迟错误 - 表示一种不紧急的无法纠正的错误处理。这可能是由于硬件数据污染，其中系统可继续运行，直到污染数据被消耗。也可以采取预防措施，例如，离线页面等。

HW\_EVENT\_ERR\_FATAL

致命错误 - 无法纠正的错误。

HW\_EVENT\_ERR\_INFO

信息 - CPER 规范定义了第四种错误类型：信息记录。

### enum mem\_type

内存类型。更详细的参考，请参阅 http://en.wikipedia.org/wiki/DRAM

**常量**

MEM\_EMPTY

空的 csrow

MEM\_RESERVED

保留的 csrow 类型

MEM\_UNKNOWN

未知的 csrow 类型

MEM\_FPM

FPM - 快速分页模式，用于1995年及以前的系统。

MEM\_EDO

EDO - 扩展数据输出，用于1998年及以前的系统。

MEM\_BEDO

BEDO - 带状扩展数据输出，是 EDO 的一种变体。

MEM\_SDR

SDR - 单数据速率 SDRAM http://en.wikipedia.org/wiki/Synchronous\_dynamic\_random-access\_memory，它们使用 3 个芯片选择引脚：引脚 0 和 2 用于排名 0；引脚 1 和 3 用于排名 1，如果内存是双排名的。

MEM\_RDR

已注册的SDR SDRAM

MEM\_DDR

双倍速率SDRAM http://en.wikipedia.org/wiki/DDR\_SDRAM

MEM\_RDDR

注册的双倍数据率SDRAM，这是DDR存储器的一种变体。注册存储器内置缓冲器，隐藏了部分存储器详细信息以对存储器控制器进行掩盖。

MEM\_RMBS

Rambus DRAM，用于一些Pentium III/IV控制器。

MEM\_DDR2

DDR2 RAM，如JEDEC JESD79-2F所述。这些记忆体被标记为“PC2-”，而不是“PC”，以区分DDR。

MEM\_FB\_DDR2

完全缓冲DDR2，如JEDEC Std No. 205和JESD206所述。这些记忆体通过DIMM插槽访问，而不是通过芯片选择信号访问。

MEM\_RDDR2

注册DDR2 RAM，这是DDR2存储器的一种变体。

MEM\_XDR

Rambus XDR，它是原始RAMBUS记忆体的演变，创建来与DDR2竞争。没有在任何x86架构上使用，但cell\_edac PPC记忆体控制器使用它。

MEM\_DDR3

DDR3 RAM

MEM\_RDDR3

注册的DDR3 RAM，这是DDR3存储器的一种变体。

MEM\_LRDDR3

减载的DDR3内存。

MEM\_LPDDR3

低功耗DDR3存储器。

MEM\_DDR4

非缓冲DDR4 RAM

MEM\_RDDR4

已注册的DDR4 RAM，这是DDR4存储器的一种变体。

MEM\_LRDDR4

减载的DDR4内存。

MEM\_LPDDR4

低功耗DDR4存储器。

MEM\_DDR5

非缓冲DDR5 RAM

MEM\_RDDR5

已注册的DDR5 RAM

MEM\_LRDDR5

减载的DDR5内存。

MEM\_NVDIMM

非易失性RAM

MEM\_WIO2

宽I/O 2。

MEM\_HBM2

高带宽存储器Gen 2。

### enum edac\_type

错误检测和纠正功能和模式

**常量**

EDAC\_UNKNOWN

未知是否支持ECC

EDAC\_NONE

不支持ECC

EDAC\_RESERVED

保留的ECC类型

EDAC\_PARITY

检测奇偶校验错误

EDAC\_EC

错误检测-无纠正

EDAC\_SECDED

单错误纠正，双错误检测

EDAC\_S2ECD2ED

Chipkill x2设备-这些存在吗？

EDAC\_S4ECD4ED

Chipkill x4设备

EDAC\_S8ECD8ED

Chipkill x8设备

EDAC\_S16ECD16ED

Chipkill x16设备

### enum scrub\_type

擦拭能力

**常量**

SCRUB\_UNKNOWN

未知是否有擦拭器

SCRUB\_NONE

没有擦拭器

SCRUB\_SW\_PROG

SW渐进（顺序）擦拭

SCRUB\_SW\_SRC

仅软件擦拭错误

SCRUB\_SW\_PROG\_SRC

自错误开始的渐进式软件擦拭

SCRUB\_SW\_TUNABLE

软件擦拭频率可调

SCRUB\_HW\_PROG

HW渐进（顺序）擦拭

SCRUB\_HW\_SRC

仅硬件擦拭错误

SCRUB\_HW\_PROG\_SRC

自错误开始的渐进式硬件擦拭

SCRUB\_HW\_TUNABLE

硬件擦拭频率可调

### enum edac\_mc\_layer\_type

内存控制器层次结构层

**常量**

EDAC\_MC\_LAYER\_BRANCH

内存层以“branch”命名

EDAC\_MC\_LAYER\_CHANNEL

内存层以“channel”命名

EDAC\_MC\_LAYER\_SLOT

内存层以“slot”命名

EDAC\_MC\_LAYER\_CHIP\_SELECT

内存层以“chip select”命名

EDAC\_MC\_LAYER\_ALL\_MEM

内存布局未知。所有存储器被映射为单个存储区域。在从固件驱动程序检索错误时使用。

**说明**

这个enum由驱动程序用来告诉edac\_mc\_sysfs在说明内存条位置时应使用什么名称。

### struct edac\_mc\_layer

说明内存控制器层次结构

**定义**

struct edac\_mc\_layer {

enum edac\_mc\_layer\_type type;

unsigned size;

bool is\_virt\_csrow;

};

**成员**

type

图层类型

size

每层组件的数量。例如，如果通道层有两个通道，则大小= 2

is\_virt\_csrow

这一层是“csrow”的一部分，当启用旧的API兼容性模式时。否则，它是一个通道。

### EDAC\_DIMM\_OFF（层，nlayers，layer0，layer1，layer2）

宏负责获取指针数组中由[layer0，layer1，layer2]位置给定的元素的指针偏移量

**参数**

layers

edac\_mc\_layer结构数组，说明为每个层分配了多少个元素

nlayers

 在层数组中的层数

layer0

layer0位置

layer1

层1位置。 如果n\_layers <2，则未使用

layer2

层2位置。 如果n\_layers <3，则未使用

**说明**

对于1层，此宏返回“var [layer0] - var”；

对于2层，此宏类似于分配一个二维数组，并返回“var [layer0] [layer1] - var”；

对于3层，此宏类似于分配一个三维数组，并返回“var [layer0] [layer1] [layer2] - var”。

这里可以使用循环使其更通用，但是，由于只有3层，因此这会更快。

按设计，图层永远不可能是0或超过3。如果这种情况发生了，将返回一个空值，导致内存分配例程中的OOPS，指向开发人员他正在做一件错误的事情。

### EDAC\_DIMM\_PTR（layers，var，nlayers，layer0，layer1，layer2）

宏负责获取由[layer0，layer1，layer2]位置给定的元素的指针数组中的指针

**参数**

layers

edac\_mc\_layer结构数组，说明为每个层分配了多少个元素

var

我们要获取指针的变量名称（如mci->dimms）

nlayers

 在层数组中的层数

layer0

layer0位置

layer1

层1位置。 如果n\_layers <2，则未使用

layer2

层2位置。 如果n\_layers <3，则未使用

**说明**

对于1层，此宏返回“var [layer0]”；

对于2层，此宏类似于分配一个二维数组，并返回“var [layer0] [layer1]”；

对于3层，这个宏类似于分配三维数组并返回“var [layer0] [layer1] [layer2]”;

### struct rank\_info

包含了一个DIMM等级的信息。

**定义**

struct rank\_info {

int chan\_idx;

struct csrow\_info \*csrow;

struct dimm\_info \*dimm;

u32 ce\_count;

};

**成员**

chan\_idx：

等级所在通道号（通常为0或1）

csrow:

指向芯片选择行结构（父结构）的指针。等级的位置由（csrow->csrow\_idx, chan\_idx）向量给出。

dimm：

指向DIMM结构的指针，其中存储了DIMM标签信息。

ce\_count：

此等级的可校正错误数

**说明**

FIXME：目前，EDAC内核模型将假定每个等级一个DIMM。

这是一个糟糕的假设，但它使得此补丁更容易。此系列的后续补丁将解决此问题。

### struct edac\_raw\_error\_desc

原始错误报告结构

**定义**

struct edac\_raw\_error\_desc {

char location[LOCATION\_SIZE];

char label[(EDAC\_MC\_LABEL\_LEN + 1 + sizeof(OTHER\_LABEL)) \* EDAC\_MAX\_LABELS];

long grain;

u16 error\_count;

int top\_layer;

int mid\_layer;

int low\_layer;

unsigned long page\_frame\_number;

unsigned long offset\_in\_page;

unsigned long syndrome;

const char \*msg;

const char \*other\_detail;

bool enable\_per\_layer\_report;

};

**成员**

location

错误位置

label

受影响的标签DIMM(s)

grain

错误报告的最小颗粒度，以字节为单位

error\_count

相同类型错误的数量

top\_layer

错误的顶层 (layer[0])

mid\_layer

错误的中间层（layer[1]）

low\_layer

错误的低层（层[2]）

page\_frame\_number

发生错误的页面

offset\_in\_page

页偏移量

syndrome

错误的综合症（如果未知或综合症不适用则为 0）

msg

错误信息

other\_detail

有关错误的其他特定于驱动程序的详细信息

enable\_per\_layer\_report

如果为假，错误会影响所有层（通常是内存控制器错误）

### struct mem\_ctl\_info \*edac\_mc\_alloc(unsigned int mc\_num, unsigned int n\_layers, struct edac\_mc\_layer \*layers, unsigned int sz\_pvt)

分配并部分填充struct mem\_ctl\_info 。

**参数**

unsigned int mc\_num

内存控制器编号

unsigned int n\_layers

MC层次结构层数

struct edac\_mc\_layer \*layers

说明内存控制器如何看到每个层次结构

unsigned int sz\_pvt

所需的私人存储空间大小

**说明**

一切都是kmalloc为一个大块-更有效。只能在所有结构具有相同的生命周期时使用-否则您必须分配并初始化自己的结构。

使用edac\_mc\_free()释放由此函数分配的mc结构。

**注意**

驱动程序以不同的方式处理多个等级的内存在某些驱动程序中，一个多等级内存棒被映射为一个条目，而在其他驱动程序中，一个单个多等级内存棒将映射到多个条目中。目前，此功能将在此类情况下分配多个struct dimm\_info ，因为分组多个等级需要驱动程序更改。

**返回**

成功时，返回指向struct mem\_ctl\_info指针的指针；否则为空

### const char \*edac\_get\_owner(void)

返回EDAC MC的拥有者mod\_name

**参数**

void

没有参数

**返回**

当EDAC MC被拥有时，返回mod\_name字符串指针。否则为空。

### void edac\_mc\_free(struct mem\_ctl\_info \*mci)

释放以前分配的mci结构

**参数**

struct mem\_ctl\_info \*mci

指向struct mem\_ctl\_info结构的指针

### bool edac\_has\_mcs(void)

检查是否已分配任何MC。

**参数**

void

没有参数

**返回**

如果MC实例已成功注册，则为true。否则为false。

### struct mem\_ctl\_info \*edac\_mc\_find(int idx)

搜索索引为idx的mem\_ctl\_info结构。

**参数**

int idx

要寻找的索引

**说明**

如果找到，返回指向结构的指针。否则返回NULL。

### struct mem\_ctl\_info \*find\_mci\_by\_dev(struct device \*dev)

扫描控制器列表，寻找管理dev设备的控制器。

**参数**

struct device \*dev

指向与MCI相关的struct device的指针

**返回**

成功时返回指向struct mem\_ctl\_info的指针；否则为空。

### struct mem\_ctl\_info \*edac\_mc\_del\_mc(struct device \*dev)

删除与dev相关联的mci结构的sysfs条目，并将mci结构从全局列表中删除。

**参数**

struct device \*dev

指向表示要删除的mci结构的struct device。

**返回**

指向已删除的mci结构的指针，如果未找到设备则返回NULL。

### int edac\_mc\_find\_csrow\_by\_page(struct mem\_ctl\_info \*mci, unsigned long page)

辅助例程，用于确定包含内存页的csrow。

**参数**

struct mem\_ctl\_info \*mci

指向struct mem\_ctl\_info结构的指针

unsigned long page

要查找的内存页

**返回**

成功时，返回csrow。如果找不到，则为-1。

### void edac\_raw\_mc\_handle\_error(struct edac\_raw\_error\_desc \*e)

向用户空间报告内存事件，而不采取任何措施发现错误位置。

**参数**

struct edac\_raw\_error\_desc \*e

错误说明

**说明**

edac\_mc\_handle\_error（）的内部原始函数。只有在BIOS直接出现硬件错误的情况下才应直接调用它，例如在APEI GHES驱动程序的情况下。

### void edac\_mc\_handle\_error(const enum hw\_event\_mc\_err\_type type, struct mem\_ctl\_info \*mci, const u16 error\_count, const unsigned long page\_frame\_number, const unsigned long offset\_in\_page, const unsigned long syndrome, const int top\_layer, const int mid\_layer, const int low\_layer, const char \*msg, const char \*other\_detail)

向用户空间报告内存事件。

**参数**

const enum hw\_event\_mc\_err\_type type

错误类型的严重程度（CE/UE/致命）

struct mem\_ctl\_info \*mci

指向mem\_ctl\_info结构的指针

const u16 error\_count

相同类型的错误数量

const unsigned long page\_frame\_number

发生错误的内存页

const unsigned long offset\_in\_page

页面内错误的偏移量

const unsigned long syndrome

ECC综合征

const int top\_layer

内存层[0]的位置

const int mid\_layer

内存层[1]的位置

const int low\_layer

内存层[2]的位置

const char \*msg

对最终用户有意义的消息，解释事件

const char \*other\_detail

关于事件的技术细节，可能有助于硬件制造商和EDAC开发人员进行分析

## PCI控制器

EDAC子系统通过调用edac\_pci\_alloc\_ctl\_info()来处理PCI控制器。它将使用struct edac\_pci\_ctl\_info来说明PCI控制器。

### struct edac\_pci\_ctl\_info \*edac\_pci\_alloc\_ctl\_info(unsigned int sz\_pvt，const char \*edac\_pci\_name)

“edac\_pci”控制信息结构的alloc()函数。

**参数**

unsigned int sz\_pvt

edac\_pci\_ctl\_info结构中私有信息的大小

const char \*edac\_pci\_name

PCI设备名称

**说明**

芯片驱动程序将为每个要控制/注册到EDAC CORE的edac\_pci分配其中之一。

返回

指向成功的struct edac\_pci\_ctl\_info结构的指针；否则是NULL。

### void edac\_pci\_free\_ctl\_info(struct edac\_pci\_ctl\_info \*pci)

pci控制结构的最后一个操作。

**参数**

struct edac\_pci\_ctl\_info \*pci

指向struct edac\_pci\_ctl\_info的指针

**说明**

调用remove sysfs信息，这将注销该控制结构的kobj。当该kobj的引用计数降至零时，将调用其释放函数，然后kfree()该内存。

### int edac\_pci\_alloc\_index(void)

分配唯一的PCI索引号

**参数**

void

无参

**返回**

分配的索引号

### int edac\_pci\_add\_device(struct edac\_pci\_ctl\_info \*pci，int edac\_idx)

将“edac\_dev”结构插入edac\_pci全局列表中，并创建与edac\_pci结构相关的sysfs条目。

**参数**

struct edac\_pci\_ctl\_info \*pci

要添加到列表中的edac\_device结构的指针

int edac\_idx

分配给“edac\_pci”结构的唯一数值标识符。

**返回**

成功返回0，失败返回错误码

### struct edac\_pci\_ctl\_info \*edac\_pci\_del\_device(struct device \*dev)

**参数**

struct device \*dev

表示要移除的edac\_pci结构的“struct device”指针

**说明**

为指定的edac\_pci结构删除sysfs条目，然后从全局列表中删除edac\_pci结构。

**返回**

指向已移除的edac\_pci结构的指针，如果未找到设备，则为NULL

### struct edac\_pci\_ctl\_info \*edac\_pci\_create\_generic\_ctl(struct device \*dev，const char \*mod\_name)

**参数**

struct device \*dev

指向struct device的指针；

const char \*mod\_name

PCI设备的名称

**说明**

用于PCI奇偶性轮询设备的通用构造器某些系统具有多个PCI总线域。对于只有一个域的系统，此API将提供通用轮询器。

这个例程调用了edac\_pci\_alloc\_ctl\_info()，将使用默认值为通用设备分配控制信息。

**返回**

成功时指向struct edac\_pci\_ctl\_info的指针，失败时为NULL。

### void edac\_pci\_release\_generic\_ctl(struct edac\_pci\_ctl\_info \*pci)

**参数**

struct edac\_pci\_ctl\_info \*pci

指向struct edac\_pci\_ctl\_info的指针

**说明**

通用EDAC PCI轮询设备的释放功能

### int edac\_pci\_create\_sysfs(struct edac\_pci\_ctl\_info \*pci)

**参数**

struct edac\_pci\_ctl\_info \*pci

指向struct edac\_pci\_ctl\_info的指针

**说明**

为指定的EDAC PCI设备创建控件/属性

### void edac\_pci\_remove\_sysfs(struct edac\_pci\_ctl\_info \*pci)

**参数**

struct edac\_pci\_ctl\_info \*pci

指向struct edac\_pci\_ctl\_info的指针

**说明**

删除此EDAC PCI设备的控件和属性

## EDAC块

EDAC子系统还提供了一种通用机制，可以通过edac\_device\_alloc\_ctl\_info()函数报告硬件其他部分的错误。

结构edac\_dev\_sysfs\_block\_attribute，edac\_device\_block，edac\_device\_instance和edac\_device\_ctl\_info在sysfs中提供了一个通用或抽象的“edac\_device”表示。

这组结构和实现其API的代码提供了注册非标准内存或PCI之外的EDAC类型设备的机制，例如

CPU缓存（L1和L2）

DMA引擎

核心CPU交换机

织物交换机单元

PCIe接口控制器

其他可以监视错误的EDAC/ECC类型的设备

它允许使用两级层次结构。

例如，缓存可能由L1、L2和L3缓存级别组成。每个CPU核心都有自己的L1缓存，同时共享L2缓存，可能也共享L3缓存。在这种情况下，可以通过以下sysfs节点表示

/sys/devices/system/edac/..

pci/ <existing pci directory (if available)>

mc/ <existing memory device directory>

cpu/cpu0/.. <L1 and L2 block directory>

/L1-cache/ce\_count

/ue\_count

/L2-cache/ce\_count

/ue\_count

cpu/cpu1/.. <L1 and L2 block directory>

/L1-cache/ce\_count

/ue\_count

/L2-cache/ce\_count

/ue\_count

...

the L1 and L2 directories would be "edac\_device\_block's"

### int edac\_device\_add\_device(struct edac\_device\_ctl\_info \*edac\_dev)

将“edac\_dev”结构插入edac\_device全局列表中，并创建与edac\_device结构相关联的sysfs条目。

**参数**

struct edac\_device\_ctl\_info \*edac\_dev

要添加到列表‘edac\_device’结构的edac\_device结构的指针。

**返回**

成功返回0，失败返回错误代码。

### struct edac\_device\_ctl\_info \*edac\_device\_del\_device(struct device \*dev)

删除指定的edac\_device结构的sysfs条目，然后从全局列表中删除edac\_device结构。

**参数**

struct device \*dev

代表要删除的edac设备结构的struct device的指针。

**返回**

指向已删除的edac\_device结构的指针，如果未找到设备则返回NULL。

### void edac\_device\_handle\_ue(struct edac\_device\_ctl\_info \* edac\_dev, int inst\_nr, int block\_nr, const char \* msg)

**参数**

struct edac\_device\_ctl\_info \* edac\_dev

指向结构edac\_device\_ctl\_info的指针

int inst\_nr

UE错误发生的实例号

int block\_nr

UE错误发生的块号

const char \* msg

要打印的消息

**说明**

执行‘edac\_dev’ UE事件的通用输出和处理

### void edac\_device\_handle\_ce(struct edac\_device\_ctl\_info \* edac\_dev, int inst\_nr, int block\_nr, const char \* msg)

**参数**

struct edac\_device\_ctl\_info \* edac\_dev

指向结构edac\_device\_ctl\_info的指针

int inst\_nr

CE错误发生的实例号

int block\_nr

CE错误发生的块号

const char \* msg

要打印的消息

**说明**

执行‘edac\_dev’ CE事件的通用输出和处理

### int edac\_device\_alloc\_index(void)

分配一个唯一的设备索引号

**参数**

void

无参数

**返回**

分配的索引号

# SCSI 接口指南

## 介绍

### 协议与总线

曾经，小型计算机系统界面定义了并行I/O总线和数据协议，以连接各种外围设备（磁盘驱动器、磁带驱动器、调制解调器、打印机、扫描仪、光学驱动器、测试设备和医疗设备）与主机计算机。

虽然旧的并行（快速/宽带/超级）SCSI总线已经大量停用，但SCSI命令集被广泛用于通过许多不同的总线与设备通信。

SCSI协议是一个大端对等的基于数据包的协议。SCSI命令长度为6、10、12或16字节，通常后跟一个相关的数据负载。

SCSI命令可以在几乎任何类型的总线上传输，并且是附加到USB、SATA、SAS、光纤通道、FireWire和ATAPI设备的存储设备的默认协议。SCSI数据包也常在Infiniband、I2O、TCP/IP（iSCSI）和平行端口之间进行交换。

### Linux SCSI子系统的设计

SCSI子系统使用三层设计，分为上、中、下三层。涉及SCSI子系统的每个操作（例如从磁盘读取扇区）使用3个级别中的每个驱动程序：一个上层驱动程序、一个下层驱动程序和SCSI中层。

SCSI上层提供用户空间和内核之间的接口，以I/O和ioctl()的块和字符设备节点的形式。SCSI下层包含针对特定硬件设备的驱动程序。

介于两者之间的是SCSI中层，类似于类似IPv4堆栈的网络路由层。SCSI中层在上层的/dev节点与下层的相应设备之间路由基于数据包的数据协议。它管理命令队列，提供错误处理和电源管理功能，并响应ioctl()请求。

## SCSI上层

上层支持用户-内核接口，提供设备节点。

sd (SCSI磁盘)

sd (sd\_mod.o)

sr (SCSI 光盘)

sr (sr\_mod.o)

st (SCSI 磁带)

st (st.o)

sg (通用SCSI )

sg (sg.o)

ch (SCSI介质转换器)

ch (ch.c)

## SCSI中间层

### SCSI中间层实现

#### include/scsi/scsi\_device.h

**struct scsi\_vpd**

SCSI 重要产品数据

1）定义

struct scsi\_vpd {

struct rcu\_head rcu;

int len;

unsigned char data[];

};

2）成员

rcu

用于kfree\_rcu()。

len

数据长度（以字节为单位）。

data

VPD数据，如T10 SCSI标准文档中定义的那样。

**shost\_for\_each\_device**

shost\_for\_each\_device(sdev, shost)

在主机的所有设备之间进行迭代

1）参数

sdev

用作游标的struct scsi\_device

shost

要迭代的struct scsi\_host

2）说明

这个迭代器返回连接到shost的每个设备。该循环获取每个设备的引用，并在结束时释放它。如果您退出循环，必须调用scsi\_device\_put(sdev)。

**\_\_shost\_for\_each\_device**

\_\_shost\_for\_each\_device(sdev, shost)

在主机的所有设备之间进行迭代（未加锁）

1）参数

sdev

用作游标的struct scsi\_device

shost

要迭代的struct scsi\_host

2）说明

这个迭代器返回连接到shost的每个设备。它不获取scsi\_device的引用，因此必须通过shost->host\_lock保护整个循环。

3）注意事项

唯一使用它的原因是因为您需要在中断上下文中访问设备列表。否则，您真正想要使用shost\_for\_each\_device。

**int scsi\_device\_supports\_vpd(struct scsi\_device \*sdev)**

测试设备是否支持VPD页面

1）参数

struct scsi\_device \*sdev

要测试的struct scsi\_device

2）说明

如果设置了“try\_vpd\_pages”标志，则它优先。否则，如果SCSI级别至少为SPC-3且未设置“skip\_vpd\_pages”，则我们将假定支持VPD页面。

#### drivers/scsi/scsi.c

SCSI中层的主文件

**void scsi\_cmd\_get\_serial(struct Scsi\_Host \* host, struct scsi\_cmnd \* cmd)**

为命令分配序列号

1）参数

struct Scsi\_Host \* host

scsi主机

struct scsi\_cmnd \* cmd

要分配序列号的命令

2）说明

序列号用于标识请求以进行错误恢复和调试。由主机的Host\_Lock保护。

**int scsi\_change\_queue\_depth(struct scsi\_device \*sdev, int depth)**

更改设备队列深度

1）参数

struct scsi\_device \*sdev

需要更改队列深度的SCSI设备

int depth

允许排队到驱动程序的命令数

2）说明

设置设备队列深度并返回新值。

**int scsi\_track\_queue\_full(struct scsi\_device \*sdev, int depth)**

跟踪QUEUE\_FULL事件以调整队列深度

1）参数

struct scsi\_device \*sdev

要跟踪的SCSI设备

int depth

此设备上当前未完成的SCSI命令的数量，不包括返回为QUEUE\_FULL的命令。

2）说明

此函数将跟踪特定SCSI设备上连续的QUEUE\_FULL事件，以确定何时需要调整设备的队列深度。

输入时未保持任何锁定状态

3）返回

0-不需要更改，>0-将队列深度调整为此新深度，

-1-回到使用host->cmd\_per\_lun作为未标记的命令深度

4）注意事项

低级驱动程序可以随时调用此功能，我们会做“正确的事情”。我们是中断上下文安全的。

**int scsi\_get\_vpd\_page(struct scsi\_device \*sdev, u8 page, unsigned char \*buf, int buf\_len)**

从SCSI设备获取重要产品数据

1）参数

struct scsi\_device \*sdev

要查询的设备

u8 page

要返回的重要产品数据

unsigned char \*buf

存储VPD的位置

int buf\_len

VPD缓冲区区域中的字节数

2）说明

SCSI设备可以选择提供重要产品数据。 VPD的每个“页面”在适当的SCSI文档（例如SPC，SBC）中定义。如果设备支持此VPD页面，则此例程将使用该页中的数据填充buf并返回0。如果不支持VPD页面或无法检索其内容，则返回-EINVAL。

**int scsi\_report\_opcode(struct scsi\_device \*sdev, unsigned char \*buffer, unsigned int len, unsigned char opcode)**

找出是否支持给定的命令操作码

1）参数

struct scsi\_device \*sdev

要查询的scsi设备

unsigned char \*buffer

暂存区（必须至少长20个字节）

unsigned int len

缓冲区的长度

unsigned char opcode

要查找的命令操作码

2）说明

使用报告支持的操作码查找给定的操作码。如果RSOC失败则返回-EINVAL，如果命令操作码不受支持则返回0，如果设备声称支持命令则返回1。

**int scsi\_device\_get(struct scsi\_device \*sdev)**

获取对scsi\_device的附加引用

1）参数

struct scsi\_device \*sdev

要引用的设备

2）说明

获取scsi\_device的引用并增加底层LLDD模块的使用计数。在调用此函数时，您必须持有母Scsi\_Host的host\_lock或已经有引用。

如果设备已删除或取消，或者LLD模块正在卸载过程中，则此操作将失败。

释放对scsi\_device的引用

**void scsi\_device\_put(struct scsi\_device \*sdev)**

1）参数

struct scsi\_device \*sdev

要释放引用的设备

2）说明

释放对scsi\_device的引用并减少底层LLDD模块的使用计数。设备在最后一个用户消失后被释放。

**void starget\_for\_each\_device(struct scsi\_target \*starget, void \*data, void (\*fn)(struct scsi\_device\*, void\*))**

遍历目标的所有设备

1）参数

struct scsi\_target \*starget

要遍历其设备的目标。

void \*data

传递给每个函数调用的不透明参数。

void (\*fn)(struct scsi\_device \*, void \*)

要在每个设备上调用的函数

2）说明

这个函数遍历starget的每个设备。这些设备具有必须由scsi\_host\_put释放的引用。

**void \_\_starget\_for\_each\_device(struct scsi\_target \*starget, void \*data, void (\*fn)(struct scsi\_device\*, void\*))**

遍历目标的所有设备（解锁）

1）参数

struct scsi\_target \*starget

要遍历其设备的目标。

void \*data

回调fn()的参数。

void (\*fn)(struct scsi\_device \*, void \*)

在每个设备上调用的回调函数。

2）说明

这个函数遍历starget的每个设备。它不对scsi\_device进行引用，因此整个循环必须受shost->host\_lock保护。

3）注意事项

唯一需要驱动程序使用这个函数的原因是因为他们需要在irq上下文中访问设备列表。否则，你确实希望使用starget\_for\_each\_device。

**struct scsi\_device \*\_\_scsi\_device\_lookup\_by\_target(struct scsi\_target \*starget, u64 lun)**

根据目标查找设备（非锁定）

1）参数

struct scsi\_target \*starget

SCSI目标指针

u64 lun

SCSI逻辑单元号

2）说明

查找指定lun的scsi\_device。返回的scsi\_device没有额外引用。在调用此函数和任何访问返回的scsi\_device时，您必须持有主机的host\_lock。跳过处于SDEV\_DEL状态的scsi\_device。

3）注意事项

驱动程序应该使用这个函数的唯一原因是因为他们需要在irq上下文中访问设备列表。否则，你确实希望使用scsi\_device\_lookup\_by\_target。

**struct scsi\_device \*scsi\_device\_lookup\_by\_target(struct scsi\_target \*starget, u64 lun)**

根据目标查找设备

1）参数

struct scsi\_target \*starget

SCSI目标指针

u64 lun

SCSI逻辑单元号

2）说明

查找指定lun的scsi\_device。返回的scsi\_device有一个需要使用scsi\_device\_put释放的额外引用。

**struct scsi\_device \*\_\_scsi\_device\_lookup(struct Scsi\_Host \*shost, uint channel, uint id, u64 lun)**

根据主机查找设备（非锁定）

1）参数

struct Scsi\_Host \*shost

SCSI主机指针

uint channel

SCSI通道（如果只有一个通道，则为零）

uint id

SCSI目标号（物理单元号）

u64 lun

SCSI逻辑单元号

2）说明

查找指定主机的通道、id、lun的scsi\_device。返回的scsi\_device没有额外引用。在调用此函数和任何访问返回的scsi\_device时，您必须持有主机的host\_lock。

3）注意事项

驱动程序使用此函数的唯一原因是因为他们需要在irq上下文中访问设备列表。否则，你确实希望使用scsi\_device\_lookup。

**struct scsi\_device \*scsi\_device\_lookup(struct Scsi\_Host \*shost, uint channel, uint id, u64 lun**)

根据主机查找设备

1）参数

struct Scsi\_Host \*shost

SCSI主机指针

uint channel

SCSI通道（如果只有一个通道，则为零）

uint id

SCSI目标号（物理单元号）

u64 lun

SCSI逻辑单元号

2）说明

查找指定主机的通道、id、lun的scsi\_device。返回的scsi\_device有一个需要使用scsi\_device\_put释放的额外引用。

#### drivers/scsi/scsicam.c

用于HDIO\_GETGEO等的SCSI通用访问方法支持函数

**unsigned char \*scsi\_bios\_ptable(struct block\_device \*dev)**

从设备的第一个扇区读取PC分区表。

1）参数

struct block\_device \*dev

来自这个设备

2）说明

读取设备的第一个扇区，并从0x1BE偏移开始返回0x42字节。

3）返回

kmalloc（GFP\_KERNEL）内存中的分区表，或错误时为NULL。

**int scsicam\_bios\_param(struct block\_device \*bdev, sector\_t capacity, int \*ip)**

确定磁盘的缸/磁头/扇区。

1）参数

struct block\_device \*bdev

哪个设备

sector\_t capacity

磁盘的尺寸（以扇区为单位）

int \*ip

返回：ip[0]=磁头数，ip[1]=扇区数，ip[2]=柱面数

2）说明

确定用于SCSI-CAM系统中的驱动器的BIOS映射/几何，根据HDIO\_GETGEO ioctl()所需的结果将结果存储在ip中。

3）返回

失败时为-1，成功时为0。

**int scsi\_partsize(unsigned char \* buf, unsigned long capacity, unsigned int \* cyls, unsigned int \* hds, unsigned int \* secs)**

从PC分区表中解析缸/磁头/扇区

1）参数

unsigned char \* buf

分区表，见scsi\_bios\_ptable（）

unsigned long capacity

磁盘的尺寸（以扇区为单位）

unsigned int \* cyls

放置缸数

unsigned int \* hds

放置磁头数

unsigned int \* secs

放置扇区数

2）说明

确定用于创建分区表的BIOS映射/几何，并将结果存储在cyls，hds和secs中

3）返回

失败时为-1，成功时为0。

#### drivers/scsi/scsi\_error.c

常见SCSI错误/超时处理例程。

**void scsi\_schedule\_eh(struct Scsi\_Host \*shost)**

为SCSI主机安排EH处理

1）参数

struct Scsi\_Host \*shost

在其上调用错误处理的SCSI主机

2）说明

无scmd的安排SCSI EH。

**int scsi\_block\_when\_processing\_errors(struct scsi\_device \*sdev)**

防止命令排队。

1）参数

struct scsi\_device \*sdev

正在从中执行恢复的设备。

2）说明

我们阻止直到主机完成错误恢复，然后检查主机或设备是否离线。

返回：

dev被错误恢复取下时为0。 1表示可以继续。

**int scsi\_check\_sense(struct scsi\_cmnd \*scmd)**

检查SCSI cmd sense

1）参数

struct scsi\_cmnd \*scmd

要检查感觉的CMD。

2）说明

返回：

SUCCESS或FAILED或NEEDS\_RETRY或ADD\_TO\_MLQUEUE

3）注释

检测到延迟错误时，当前命令尚未执行，需要重试。

**void scsi\_eh\_prep\_cmnd(struct scsi\_cmnd \*scmd, struct scsi\_eh\_save \*ses, unsigned char \*cmnd, int cmnd\_size, unsigned sense\_bytes)**

将scsi命令信息保存为错误恢复的一部分

1）参数

struct scsi\_cmnd \*scmd

要劫持的SCSI命令结构

struct scsi\_eh\_save \*ses

保存还原信息的结构

unsigned char \*cmnd

要发送的CDB。如果不需要新命令，则可以为NULL

int cmnd\_size

cmnd的字节数（必须<= MAX\_COMMAND\_SIZE)

unsigned sense\_bytes

要复制的感知数据的大小。或0（如果！= 0，则忽略cmnd）

2）说明

此函数用于在错误恢复过程中重新执行之前保存SCSI命令信息。如果sense\_bytes为0，则发送的命令必须是不传输任何数据的命令。如果sense\_bytes！= 0，则忽略cmnd，并且此函数设置REQUEST\_SENSE命令和cmnd缓冲区以将sense\_bytes读入scmd - > sense\_buffer。

**void scsi\_eh\_restore\_cmnd(struct scsi\_cmnd \*scmd, struct scsi\_eh\_save \*ses)**

将scsi命令信息恢复为错误恢复的一部分

1）参数

struct scsi\_cmnd \*scmd

要恢复的SCSI命令结构

struct scsi\_eh\_save \*ses

与scsi\_eh\_prep\_cmnd的相应调用相关的保存信息

2）说明

撤消上述scsi\_eh\_prep\_cmnd()造成的任何破坏。

**void scsi\_eh\_finish\_cmd(struct scsi\_cmnd \*scmd, struct list\_head \*done\_q)**

处理eh完成的cmd。

1）参数

struct scsi\_cmnd \*scmd

eh已完成的原始SCSI命令。

struct list\_head \*done\_q

已处理命令的队列。

2）注释

我们不想在仍在处理错误时使用正常的命令完成 - 它可能导致其他命令排队，这会干扰我们正在做的事情。因此，我们真正想保留一个挂起命令的列表以进行最终完成，一旦我们准备离开错误处理，我们就可以真正地处理完成。

**int scsi\_eh\_get\_sense(struct list\_head \*work\_q, struct list\_head \*done\_q)**

获取设备感知数据。

1）参数

struct list\_head \*work\_q

要处理的命令队列。

struct list\_head \*done\_q

已处理命令的队列。

2）说明

查看是否需要请求感知信息。如果是，则立即获取它，以便我们更好地知道要做什么。

3）注释

不幸的是，这会导致一个令人不悦的副作用，即如果shost适配器不自动请求感知信息，我们最终会关闭它之前请求它。

现在，所有驱动程序都应在内部请求感知信息，因此，如果您进入此位置，则只能说是tough noogies。

XXX：长期来看，这段代码应该消失，但首先需要审核所有LLDD。

**void scsi\_eh\_ready\_devs(struct Scsi\_Host \*shost, struct list\_head \*work\_q, struct list\_head \*done\_q)**

检查设备准备就绪状态，如有必要进行恢复。

1）参数

struct Scsi\_Host \*shost

要恢复的主机。

struct list\_head \*work\_q

挂起命令的list\_head。

struct list\_head \*done\_q

已处理命令的list\_head。

**void scsi\_eh\_flush\_done\_q(struct list\_head \*done\_q)**

完成已处理的命令或重试它们。

1）参数

struct list\_head \*done\_q

已处理命令的list\_head。

**int scsi\_ioctl\_reset(struct scsi\_device \*dev，int \_\_user \*arg)**

参数

struct scsi\_device \* dev

要操作的scsi\_device

int \_\_user \*arg

重置类型（参见sg.h）

**bool scsi\_get\_sense\_info\_fld(const u8 \*sense\_buffer，int sb\_len，u64 \*info\_out)**

从感知数据（固定或说明符格式）中获取信息字段

1）参数

const u8 \*sense\_buffer

感知数据的字节数组

int sb\_len

sense\_buffer中有效字节的数量

u64 \*info\_out

指向64整数的指针，如果找到8或4字节信息字段，则将其放置在其中。

2）说明

返回：

如果找到信息字段，则返回true，如果未找到，则返回false。

#### drivers/scsi/scsi\_devinfo.c

管理scsi\_dev\_info\_list，该列表跟踪已列入黑名单和白名单的设备。

**int scsi\_dev\_info\_list\_add(int compatible，char \*vendor，char \*model，char \*strflags，blist\_flags\_t flags)**

添加一个dev\_info列表条目。

1）参数

int compatible

如果为true，则将短字符串终止为null。否则，进行空格填充。

char \*vendor

供应商字符串

char \*model

型号（产品）字符串

char \*strflags

整数字符串

blist\_flags\_t flags

如果strflags为NULL，则使用此标志值

2）说明

为供应商、型号、strflags或标志创建并添加一个dev\_info条目。如果兼容，则添加到列表的尾部，不进行空格填充，并设置devinfo->compatible。使用兼容性为1和clfags NULL添加scsi\_static\_device\_list条目。

3）返回

0 OK，-错误的失败。

**struct scsi\_dev\_info\_list \*scsi\_dev\_info\_list\_find(const char \*vendor，const char \*model，enum scsi\_devinfo\_key key)**

查找匹配的dev\_info列表条目。

1）参数

const char \*vendor

完整的供应商字符串

const char \*model

完整的型号（产品）字符串

enum scsi\_devinfo\_key key

指定要使用的列表

2）说明

在由key指定的列表中查找与供应商、型号匹配的第一个dev\_info条目。

3）返回

匹配条目的指针，或失败的ERR\_PTR。

**int scsi\_dev\_info\_list\_add\_str(char \*dev\_list)**

解析dev\_list并添加到scsi\_dev\_info\_list。

1）参数

char \*dev\_list

要添加的设备标志字符串

2）说明

解析dev\_list并添加条目到scsi\_dev\_info\_list。dev\_list的格式为“vendor:product:flag，vendor:product:flag”。通过strsep修改dev\_list。可用于命令行添加，proc或sysfs接口。

3）返回

0表示OK，-error表示失败。

**blist\_flags\_t scsi\_get\_device\_flags（struct scsi\_device \* sdev，const unsigned char \* vendor，const unsigned char \*model）**

从动态设备列表中获取设备特定标志。

1）参数

struct scsi\_device \* sdev

获取标志的scsi\_device

const unsigned char \* vendor

厂商名称

const unsigned char \* model

型号名称

2）说明

搜索全局scsi\_dev\_info\_list（由列表零指定）以查找与供应商和型号匹配的条目，如果找到，则返回匹配的标志值，否则返回主机或全局默认设置。在扫描时调用。

**void scsi\_exit\_devinfo（void）**

删除/ proc / scsi / device\_info和scsi\_dev\_info\_list

1）参数

void

没有参数

**int scsi\_init\_devinfo(void)**

设置动态设备列表。

1）参数

void

没有参数

2）说明

从scsi\_dev\_flags添加命令行条目，然后将scsi\_static\_device\_list条目添加到scsi设备信息列表中。

#### drivers / scsi / scsi\_ioctl.c

处理SCSI设备的ioctl（）调用。

**int scsi\_ioctl（struct scsi\_device \* sdev，fmode\_t mode，int cmd，void \_\_user \* arg）**

将ioctl分派到scsi设备

1）参数

struct scsi\_device \* sdev

接收ioctl的scsi设备

fmode\_t mode

用于打开块/字符设备的模式

int cmd

是哪个ioctl

void \_\_user \* arg

与ioctl相关联的数据

2）说明

scsi\_ioctl（）函数不同于大多数ioctl，因为它不接受major / minor号作为dev字段。相反，它采用指向struct scsi\_device的指针。

#### drivers / scsi / scsi\_lib.c

SCSI排队库。

**int \_\_scsi\_execute（struct scsi\_device \* sdev，const unsigned char \* cmd，int data\_direction，void \* buffer，unsigned bufflen，unsigned char \* sense，struct scsi\_sense\_hdr \* sshdr，int timeout，int retries，blk\_opf\_t flags ，req\_flags\_t rq\_flags，int \* resid）**

插入请求并等待结果

1）参数

struct scsi\_device \* sdev

SCSI设备

const unsigned char \* cmd

SCSI命令

int data\_direction

数据方向

void \* buffer

数据缓冲区

unsigned bufflen

缓冲区长度

unsigned char \* sense

可选的感觉缓冲区

struct scsi\_sense\_hdr \* sshdr

可选的解码感觉头

int timeout

在HZ中的请求超时

int retries

请求重试次数

blk\_opf\_t flags

-> cmd\_flags的标志

req\_flags\_t rq\_flags

-> rq\_flags的标志

int \* resid

可选的剩余长度

2）说明

如果执行了命令，则返回scsi\_cmnd结果字段，如果我们没有达到那个位置，则返回负Linux错误代码。

**struct scsi\_device \* scsi\_device\_from\_queue（struct request\_queue \* q）**

返回与request\_queue关联的sdev

1）参数

struct request\_queue \* q

要从中返回sdev的请求队列

2）说明

返回与请求队列关联的sdev，如果请求队列未引用SCSI设备，则返回NULL。

**int scsi\_mode\_select（struct scsi\_device \* sdev，int pf，int sp，unsigned char \* buffer，int len，int timeout，int retries，struct scsi\_mode\_data \* data，struct scsi\_sense\_hdr \* sshdr）**

发出模式选择

1）参数

struct scsi\_device \* sdev

要查询的SCSI设备

int pf

页面格式位（1 ==标准，0 ==供应商特定）

int sp

保存页位（0 ==不保存，1 ==保存）

unsigned char \* buffer

请求缓冲区（大小不能小于八个字节）

int len

请求缓冲区的长度。

int timeout

命令超时

int retries

失败之前的重试次数

struct scsi\_mode\_data \* data

返回抽象模式头数据的结构

struct scsi\_sense\_hdr \* sshdr

放置感觉数据的地方（如果收集不到感觉，则为NULL）。必须是SCSI\_SENSE\_BUFFERSIZE大。

如果成功，则返回零；在出错时返回负错误号或SCSI状态

**int scsi\_mode\_sense（struct scsi\_device \* sdev，int dbd，int modepage，unsigned char \* buffer，int len，int timeout，int retries，struct scsi\_mode\_data \* data，struct scsi\_sense\_hdr \* sshdr）**

发出模式感知，如果必要，从10字节降级到6字节。

1）参数

struct scsi\_device \* sdev

要查询的SCSI设备

int dbd

设置以防止模式感知返回块说明符

int modepage

请求的模式页

unsigned char \* buffer

请求缓冲区（大小不能小于八个字节）

int len

请求缓冲区的长度。

int timeout

命令超时

int retries

失败之前的重试次数

struct scsi\_mode\_data \* data

返回抽象模式头数据的结构

struct scsi\_sense\_hdr \* sshdr

放置感觉数据的地方（如果收集不到感觉，则为NULL）。必须是SCSI\_SENSE\_BUFFERSIZE大。

如果成功，则返回零；在出错时返回负错误号.

**int scsi\_test\_unit\_ready（struct scsi\_device \* sdev，int timeout，int retries，struct scsi\_sense\_hdr \* sshdr）**

测试单元是否就绪

1）参数

struct scsi\_device \* sdev

SCSI 设备更改状态。

int 超时

命令超时

int 重试次数

失败之前的重试次数

struct scsi\_sense\_hdr \*sshdr

解码感官信息的输出指针。

如果 TUR 失败，则返回零或错误。对于可移动介质，UNIT\_ATTENTION 设置 ->changed 标志。

**int scsi\_device\_set\_state(struct scsi\_device \*sdev，enum scsi\_device\_state state)**

通过设备状态模型取得指定设备状态。

1）参数

struct scsi\_device \*sdev

要更改状态的 SCSI 设备。

enum scsi\_device\_state state

要更改到的状态。

如果成功则返回零或请求转换非法则返回错误。

**void sdev\_evt\_send(struct scsi\_device \*sdev，struct scsi\_event \*evt)**

发送事件到 uevent 线程

1）参数

struct scsi\_device \*sdev

事件发生的 SCSI 设备

struct scsi\_event \*evt

要发送的事件

异步断言 SCSI 设备事件。

**struct scsi\_event \*sdev\_evt\_alloc(enum scsi\_device\_event evt\_type，gfp\_t gfpflags)**

分配一个新的 SCSI 事件

1）参数

enum scsi\_device\_event evt\_type

要分配的事件类型

gfp\_t gfpflags

用于分配的 GFP 标志

分配并返回新的 scsi\_event。

**void sdev\_evt\_send\_simple(struct scsi\_device \*sdev，enum scsi\_device\_event evt\_type，gfp\_t gfpflags)**

发送事件到 uevent 线程

1）参数

struct scsi\_device \*sdev

事件发生的 SCSI 设备

enum scsi\_device\_event evt\_type

要发送的事件类型

gfp\_t gfpflags

用于分配的 GFP 标志

异步断言 SCSI 设备事件，给定事件类型。

**int scsi\_device\_quiesce(struct scsi\_device \*sdev)**

阻止除电源管理外的所有命令。

1）参数

struct scsi\_device \*sdev

要静音的 SCSI 设备。

通过尝试转换到 SDEV\_QUIESCE 状态（必须是合法的转换）来达到这个目的。当设备处于此状态时，只有接受电源管理请求，所有其他请求都将被延迟。

必须在用户上下文中调用，可能会睡眠。

如果不成功则返回零或错误。

**void scsi\_device\_resume(struct scsi\_device \*sdev)**

恢复静音设备的用户发出的命令。

1）参数

struct scsi\_device \*sdev

要恢复的 SCSI 设备。

将设备从静音恢复到运行并重新启动队列。

必须在用户上下文中调用，可能会睡眠。

**int scsi\_internal\_device\_block\_nowait(struct scsi\_device \*sdev)**

尝试转换到 SDEV\_BLOCK 状态

1）参数

struct scsi\_device \*sdev

要阻止的设备

2）说明

在指定设备上暂停 SCSI 命令处理。不睡眠。

如果成功则返回零或负错误代码。

3）注释

此例程将设备转换为 SDEV\_BLOCK 状态（必须是合法的转换）。当设备处于此状态时，命令处理被暂停，直到设备离开 SDEV\_BLOCK 状态。另请参见 scsi\_internal\_device\_unblock\_nowait()。

**int scsi\_internal\_device\_unblock\_nowait(struct scsi\_device \*sdev，enum scsi\_device\_state new\_state)**

在块请求之后恢复设备

1）参数

struct scsi\_device \*sdev

要恢复的设备

enum scsi\_device\_state new\_state

解除阻止后要设置设备的状态

2）**说明**

重新启动以前已经暂停的 SCSI 设备队列。不睡眠。

如果成功则返回零或负错误代码。

3）注释

此例程将设备转换为 SDEV\_RUNNING 状态或一个离线状态（必须是合法的转换），允许中间层启动该设备的队列。

**void \*scsi\_kmap\_atomic\_sg(struct scatterlist \*sgl，int sg\_count，size\_t \*offset，size\_t \*len)**

查找并原子地映射 SG 元素

1）参数

struct scatterlist \*sgl

散列收集列表

int sg\_count

sg 中的段数

size\_t \*offset

在 sg 中的字节偏移，在映射区域中返回偏移量

size\_t \*len

要映射的字节数，在返回时返回映射的字节数

2）说明

返回映射页面开始的虚拟地址

**void scsi\_kunmap\_atomic\_sg(void \*virt)**

使用 scsi\_kmap\_atomic\_sg 映射虚拟地址

1）参数

void \*virt

要取消映射的虚拟地址

**int scsi\_vpd\_lun\_id(struct scsi\_device \*sdev，char \*id，size\_t id\_len)**

返回唯一设备标识符

1）参数

struct scsi\_device \*sdev

SCSI 设备

char \*id

用于标识的缓冲区

size\_t id\_len

缓冲区长度

2）说明

根据设备的 VPD 页面 0x83 中的信息，将唯一设备标识符复制到 id 中。该字符串将格式化为 SCSI 名称字符串。

返回标识符的长度或失败时返回错误。如果标识符长度超过所提供的缓冲区，则返回实际的标识符长度，并且不填零缓冲区。

#### drivers/scsi/scsi\_lib\_dma.c

依赖于 DMA（映射和取消映射散列收集列表）的 SCSI 库函数。

**int scsi\_dma\_map(struct scsi\_cmnd \*cmd)**

对命令的 SG 列表执行 DMA 映射

1）参数

struct scsi\_cmnd \*cmd

SCSI 命令

2）说明

返回实际使用的 SG 列表数，如果 SG 列表为空则为零，或 -ENOMEM（映射失败）。

**void scsi\_dma\_unmap(struct scsi\_cmnd \*cmd)**

取消映射 scsi\_dma\_map 映射的命令的 SG 列表

1）参数

struct scsi\_cmnd \*cmd

SCSI 命令

#### drivers/scsi/scsi\_proc.c

此文件中的函数提供了 PROC 文件系统与 SCSI 设备驱动之间的接口。主要用于调试、统计和直接传递信息到底层驱动程序。例如，管理 /proc/scsi/\* 时使用的管道。

**void scsi\_proc\_hostdir\_add(struct scsi\_host\_template \*sht)**

在 /proc 中为 SCSI 主机创建目录。

1）参数

struct scsi\_host\_template \*sht

此目录的所有者。

2）说明

将 sht->proc\_dir 设置为新目录。

**void scsi\_proc\_hostdir\_rm(struct scsi\_host\_template \*sht)**

在 /proc 中删除 SCSI 主机的目录。

1）参数

struct scsi\_host\_template \*sht

此目录的所有者。

**void scsi\_proc\_host\_add(struct Scsi\_Host \*shost)**

将此主机的条目添加到适当的 /proc 目录中。

1）参数

struct Scsi\_Host \*shost

要添加的主机。

**void scsi\_proc\_host\_rm(struct Scsi\_Host \*shost)**

从 /proc 中删除此主机的条目。

1）参数

struct Scsi\_Host \*shost

哪个主机。

**int proc\_print\_scsidevice(struct device \*dev, void \*data)**

返回有关此主机的数据。

1）参数

struct device \*dev

一个 SCSI 设备。

void \*data

输出到的 struct seq\_file。

2）说明

打印 Host、通道、ID、LUN、Vendor、Model、Rev、Type 和 revision。

**int scsi\_add\_single\_device(uint host, uint channel, uint id, uint lun)**

响应用户请求探测/添加设备。

1）参数

uint host

用户提供的十进制整数。

uint channel

用户提供的十进制整数。

uint id

用户提供的十进制整数。

uint lun

用户提供的十进制整数。

2）说明

通过在 /proc/scsi/scsi 中写入“scsi add-single-device”来调用。执行 scsi\_host\_lookup()，并且如果该传输类型支持，则执行 user\_scan()；否则，执行 scsi\_scan\_host\_selected()。

3）注意

这似乎专门针对 SCSI 并行总线。

**int scsi\_remove\_single\_device(uint host, uint channel, uint id, uint lun)**

响应用户请求删除设备。

1）参数

uint host

用户提供的十进制整数。

uint channel

用户提供的十进制整数。

uint id

用户提供的十进制整数。

uint lun

用户提供的十进制整数。

2）说明

通过在 /proc/scsi/scsi 中写入“scsi remove-single-device”来调用。执行 scsi\_device\_lookup() 和 scsi\_remove\_device()。

s**size\_t proc\_scsi\_write(struct file \*file, const char \_\_user \*buf, size\_t length, loff\_t \*ppos)**

处理 /proc/scsi/scsi 中的写操作。

1）参数

struct file \*file

未使用。

const char \_\_user \*buf

要写入的缓冲区。

size\_t length

buf 的长度，至多为 PAGE\_SIZE。

loff\_t \*ppos

未使用。

2）说明

这提供了一种遗留机制，可以通过 Host、通道、ID 和 LUN 来添加或删除设备。要使用，“echo 'scsi add-single-device 0 1 2 3' > /proc/scsi/scsi”或“echo 'scsi remove-single-device 0 1 2 3' > /proc/scsi/scsi”，其中“0 1 2 3”分别替换为 Host、通道、ID 和 LUN。

3）注意

这似乎针对并行 SCSI。大多数现代总线（USB、SATA、Firewire、光纤通道等）动态分配这些值以提供唯一的标识符，仅此而已。

**int proc\_scsi\_open(struct inode \*inode, struct file \*file)**

粘合函数。

1）参数

struct inode \*inode

未使用。

struct file \*file

传递给 single\_open()。

2）说明

将 proc\_scsi\_show 与此文件关联。

**int scsi\_init\_procfs(void)**

在 procfs 中创建 scsi 和 scsi/scsi。

1）参数

void

无参数。

**void scsi\_exit\_procfs(void)**

从 procfs 中删除 scsi/scsi 和 scsi。

2）参数

void

无参数。

#### drivers/scsi/scsi\_netlink.c

通过 netlink 提供异步事件的基础结构从传输到用户空间，使用一个 NETLINK\_SCSITRANSPORT 协议用于所有传输。有关更多详细信息，请参见原始修补程序提交。

**void scsi\_nl\_rcv\_msg(struct sk\_buff \*skb)**

接收消息处理程序。

1）参数

struct sk\_buff \*skb

套接字接收缓冲区。

2）说明

从接收缓冲区中提取消息。验证消息头并调用适当的传输消息处理程序。

**void scsi\_netlink\_init(void)**

由 SCSI 子系统调用以初始化 SCSI 传输 netlink 接口。

1）参数

void

无参数。

**void scsi\_netlink\_exit(void)**

由 SCSI 子系统调用以禁用 SCSI 传输 netlink 接口。

1）参数

void

无参数。

#### drivers/scsi/scsi\_scan.c

扫描主机以确定连接了哪些（如果有）设备。通常的扫描/探测算法如下，根据设备特定标志、编译选项和全局变量（引导或模块加载时）设置进行了例外处理。通过 INQUIRY 命令扫描特定的 LUN；如果 LUN 有连接的设备，则为其分配并设置 scsi\_device。对于给定主机上的每个通道的每个 ID，从扫描 LUN 0 开始。跳过完全不响应 LUN 0 的主机。否则，如果 LUN 0 有连接的设备，则为其分配并设置 scsi\_device。如果目标是 SCSI-3 或更高版本，则发出 REPORT LUN，并扫描 REPORT LUN 返回的所有 LUN；否则，顺序扫描 LUN，直到达到一些最大值或者看到不能连接设备的 LUN。

**int scsi\_complete\_async\_scans(void)**

等待异步扫描完成

1）参数

无

无参数

2）说明

当此函数返回时，任何在此函数调用之前开始扫描的主机都已完成其扫描。在此函数被调用之后开始扫描的主机可能已完成或未完成。

**void scsi\_unlock\_floptical(struct scsi\_device \*sdev, unsigned char \*result)**

通过特殊的 MODE SENSE 命令解锁设备

1）参数

struct scsi\_device \*sdev

发送命令的 scsi 设备

unsigned char \*result

存储 MODE SENSE 结果的区域

2）说明

发送厂商特定的 MODE SENSE（而不是 MODE SELECT）命令。仅对 BLIST\_KEY 设备调用。

**struct scsi\_device \*scsi\_alloc\_sdev(struct scsi\_target \*starget, u64 lun, void \*hostdata)**

分配并设置一个 SCSI 设备

1）参数

struct scsi\_target \*starget

要为其分配 SCSI 设备的目标

u64 lun

哪个 LUN

void \*hostdata

通常为 NULL，并由 ->slave\_alloc 设置

2）说明

分配、初始化 IO 并返回指向 SCSI 设备的指针。将 shost、channel、id 和 lun 存储在 SCSI 设备中，并将 SCSI 设备添加到适当的列表中。

返回：

SCSI 设备指针，或在失败时为 NULL。

**void scsi\_target\_reap\_ref\_release(struct kref \*kref)**

从可见性中删除目标

1）参数

struct kref \*kref

被释放的目标的 reap\_ref

2）说明

当 reap\_ref 的最后一个 put 调用时调用，这表明不再有任何设备在此目标下可见，因此在 sysfs 中使目标不可见。注意：我们必须在用户上下文中执行此操作，因为目标 reap 应该在移除 SCSI 设备可见性的位置执行。

**struct scsi\_target \*scsi\_alloc\_target(struct device \*parent, int channel, uint id)**

分配一个新的或查找现有的目标

1）参数

struct device \*parent

目标的父级（不需要是 SCSI 主机）

int channel

目标通道号（如果没有通道，则为零）

uint id

目标 ID 号码

2）说明

如果已存在现有目标，则返回该目标，前提是它还没有进入 STARGET\_DEL 状态，否则分配新目标。

返回目标时，其引用次数已增加，因此调用者负责回收并进行最后一次放置。

**void scsi\_target\_reap(struct scsi\_target \*starget)**

检查目标是否正在使用并销毁

1）参数

struct scsi\_target \*starget

要检查的目标

2）说明

在删除 LUN 或对目标进行最后一次放置之后使用此函数，它会原子地检查是否有任何内容正在使用该目标，并在出现这种情况时删除该目标。

**int scsi\_probe\_lun(struct scsi\_device \*sdev, unsigned char \*inq\_result, int result\_len, blist\_flags\_t \*bflags)**

使用 SCSI INQUIRY 探测单个 LUN

1）参数

struct scsi\_device \*sdev

要探测的 SCSI 设备

unsigned char \*inq\_result

存储 INQUIRY 结果的区域

int result\_len

inq\_result 的长度

blist\_flags\_t \*bflags

将任何 bflags 存储在此处

2）说明

使用标准 SCSI INQUIRY 探测与请求相关联的 LUN；如果 INQUIRY 成功，则返回零，并将 INQUIRY 数据存储在 inq\_result 中；将 scsi\_level 和 INQUIRY 长度复制到 scsi\_device 中，任何 flags 值都存储在 \*bflags 中。

**int scsi\_add\_lun(struct scsi\_device \*sdev, unsigned char \*inq\_result, blist\_flags\_t \*bflags, int async)**

分配并完全初始化 SCSI 设备

1）参数

struct scsi\_device \*sdev

包含要存储在新 SCSI 设备中的信息

unsigned char \*inq\_result

保存先前对 LUN 进行的 INQUIRY 的结果

blist\_flags\_t \*bflags

黑 / 白名单标志

int async

如果此设备正在异步扫描，则为 1

2）说明

初始化 SCSI 设备 sdev。根据 \*bflags 中的值可选地设置字段。

3）返回

SCSI\_SCAN\_NO\_RESPONSE：无法分配或设置 SCSI 设备 SCSI\_SCAN\_LUN\_PRESENT：已分配并初始化新 SCSI 设备

**unsigned char \*scsi\_inq\_str(unsigned char \*buf, unsigned char \*inq, unsigned first, unsigned end)**

从最小到最大索引打印 INQUIRY 数据，去除尾随空格

1）参数

unsigned char \*buf

输出缓冲区，至少有 end-first+1 字节的空间

unsigned char \*inq

Inquiry 缓冲区（输入）

unsigned first

从 inq 中提取的字符串的偏移量

unsigned end

inq 中最后一个字符之后的索引

**int scsi\_probe\_and\_add\_lun(struct scsi\_target \*starget, u64 lun, blist\_flags\_t \*bflagsp, struct scsi\_device \*\*sdevp, enum scsi\_scan\_mode rescan, void \*hostdata)**

探测 LUN，如果找到 LUN，则将其添加

1）参数

struct scsi\_target \*starget

指向目标设备结构的指针

u64 lun

目标设备的 LUN

blist\_flags\_t \*bflagsp

如果不为 NULL，则在此处存储 bflags

struct scsi\_device \*\*sdevp

探测与此 scsi\_device 对应的 LUN

enum scsi\_scan\_mode rescan

如果不等于 SCSI\_SCAN\_INITIAL，则跳过仅需要在第一次扫描时使用的一些代码

void \*hostdata

传递到 scsi\_alloc\_sdev()

2）说明

调用 scsi\_probe\_lun，如果找到连接设备的 LUN，则电话 scsi\_add\_lun 进行分配和设置。

3）返回

SCSI\_SCAN\_NO\_RESPONSE：无法分配或设置 SCSI 设备

SCSI\_SCAN\_TARGET\_PRESENT：目标已响应，但在LUN上未连接设备

SCSI\_SCAN\_LUN\_PRESENT：已分配并初始化新的scsi\_device

**void scsi\_sequential\_lun\_scan(struct scsi\_target \*starget, blist\_flags\_t bflags, int scsi\_level, enum scsi\_scan\_mode rescan)**

按顺序扫描SCSI目标

1）参数

struct scsi\_target \*starget

指向要扫描的目标struct 的指针

blist\_flags\_t bflags

用于LUN 0的黑名单/白名单标志

int scsi\_level

该设备遵循哪个版本的标准

enum scsi\_scan\_mode rescan

传递到scsi\_probe\_add\_lun（）中

2）说明

通常从LUN 1（假定已经扫描了LUN 0）开始扫描到某个最大的LUN，直到找到一个没有连接设备的LUN。使用bflags来确定任何奇怪的情况。

修改sdevscan->lun。

**int scsi\_report\_lun\_scan(struct scsi\_target \*starget，blist\_flags\_t bflags，enum scsi\_scan\_mode rescan)**

使用SCSI REPORT LUN结果进行扫描

1）参数

struct scsi\_target \*starget

哪个目标

blist\_flags\_t bflags

BLIST\_NOLUN，BLIST\_REPORTLUN2或BLIST\_NOREPORTLUN的零值或混合

enum scsi\_scan\_mode rescan

如果我们只需要跳过第一次扫描所需的代码，则非零值

2）说明

通过发送REPORT LUN命令进行快速扫描现代（SCSI-3）设备。通过调用scsi\_probe\_and\_add\_lun扫描LUN列表的结果。

如果设置了BLINK\_REPORTLUN2，则即使其早于SCSI-3，也可以扫描支持8个以上LUN的目标。如果设置了BLIST\_NOREPORTLUN，则始终返回1。如果设置了BLIST\_NOLUN，则始终返回0。如果设置了starget->no\_report\_luns，则始终返回1。

3）返回

0：扫描完成（或无法进行进一步的扫描，因此是徒劳的）1：无法使用REPORT LUN进行扫描

**struct async\_scan\_data \*scsi\_prep\_async\_scan(struct Scsi\_Host \*shost)**

准备进行异步扫描

1）参数

struct Scsi\_Host \*shost

将被扫描的主机

2）返回

一个cookie，将传递给scsi\_finish\_async\_scan（）

3）说明

告诉midlayer这个主机将做异步扫描。它保留了主机在扫描列表中的位置，并确保在此之后启动的其他异步扫描不会影响发现的设备的排序。

**void scsi\_finish\_async\_scan(struct async\_scan\_data \*data)**

异步扫描完成

1）参数

struct async\_scan\_data \*data

从先前调用scsi\_prep\_async\_scan（）返回的cookie

2）说明

所有当前连接到该主机的设备都已找到。此函数将发现的所有设备公告给系统的其余部分。

#### drivers / scsi / scsi\_sysctl.c

设置sysctl条目：“/ dev / scsi / logging\_level”（DEV\_SCSI\_LOGGING\_LEVEL），用于设置/返回scsi\_logging\_level

#### drivers / scsi / scsi\_sysfs.c

SCSI sysfs接口例程。

**void scsi\_remove\_device(struct scsi\_device \*sdev)**

从SCSI总线注销设备

1）参数

struct scsi\_device \*sdev

要注销的scsi\_device

**void scsi\_remove\_target(struct device \*dev)**

尝试删除目标及其所有设备

1）参数

struct device \*dev

要删除的一般的starget或父级starget

2）注意

这有点赛车。如果用户请求添加另一个设备，那么目标可能不会被删除。

#### drivers/scsi/hosts.c

从中间到低级SCSI驱动程序接口

**void scsi\_remove\_host(struct Scsi\_Host \*shost)**

删除SCSI主机

1）参数

struct Scsi\_Host \*shost

指向要删除的scsi主机的指针

**int scsi\_add\_host\_with\_dma(struct Scsi\_Host \*shost，struct device \*dev，struct device \*dma\_dev)**

使用dma设备添加scsi主机

1）参数

struct Scsi\_Host \*shost

要添加的scsi主机指针

struct device \*dev

一种类型为scsi类的struct

struct device \*dma\_dev

主机DMA设备

2）注意

你很少需要担心这一点，除非你在虚拟主机环境中，因此请改用更简单的scsi\_add\_host（）函数。

返回：

成功时为0 /！= 0为错误

**struct Scsi\_Host \*scsi\_host\_alloc(struct scsi\_host\_template \*sht，int privsize)**

注册scsi主机适配器实例。

1）参数

struct scsi\_host\_template \*sht

指向scsi主机模板的指针

int privsize

为驱动程序分配的额外字节数

2）注意

分配新的Scsi\_Host并执行基本初始化。在调用scsi\_add\_host之前，不会将主机发布到scsi中层。

3）说明

返回：

指向新Scsi\_Host的指针

**struct Scsi\_Host \*scsi\_host\_lookup（unsigned short hostnum）**

通过主机编号获取对Scsi\_Host的引用

1）参数

unsigned short hostnum

要查找的主机编号

2）说明

返回：

指向所定位的Scsi\_Host的指针或NULL。

调用者必须执行scsi\_host\_put（）以放弃scsi\_host\_get（）所获得的引用。下面的put\_device（）从class\_find\_device（）放弃了参考。

**struct Scsi\_Host \*scsi\_host\_get（struct Scsi\_Host \*shost）**

增加Scsi\_Host引用计数

1）参数

struct Scsi\_Host \*shost

要增加的Scsi\_Host的指针。

**int scsi\_host\_busy（struct Scsi\_Host \*shost）**

返回主机繁忙计数

参数

struct Scsi\_Host \*shost

要增加的Scsi\_Host的指针。

**void scsi\_host\_put（struct Scsi\_Host \*shost）**

减少Scsi\_Host引用计数

参数

struct Scsi\_Host \*shost

要减少的Scsi\_Host的指针。

**int scsi\_queue\_work（struct Scsi\_Host \*shost，struct work\_struct \*work）**

将工作排入 Scsi\_Host 工作队列。

1）参数

struct Scsi\_Host \*shost

Scsi\_Host 指针。

struct work\_struct \*work

要排队执行的工作。

2）说明

返回：

1 - 工作已排队执行 0 - 工作已经被排队 -EINVAL - 工作队列不存在

**void scsi\_flush\_work(struct Scsi\_Host \*shost)**

刷新 Scsi\_Host 工作队列。

参数

struct Scsi\_Host \*shost

Scsi\_Host 指针。

#### drivers/scsi/scsi\_common.c

通用支持函数

**const char \*scsi\_device\_type(unsigned type)**

返回一个指示设备类型的 17 个字符的字符串。

参数

unsigned type

要查找的类型编号

**u64 scsilun\_to\_int(struct scsi\_lun \*scsilun)**

将 scsi\_lun 转换为整数

1）参数

struct scsi\_lun \*scsilun

要转换的 struct scsi\_lun。

2）说明

将 scsilun 从 struct scsi\_lun 转换为四字节的主机字节顺序的整数，并返回结果。调用者必须在使用此函数之前检查截断。

3）注

有关 LUN 格式的说明，请参见 SCSI 架构模型后 SCSI-3，有关 SCSI-3，请参见 SCSI 控制器命令。

对于 struct scsi\_lun：d2 04 0b 03 00 00 00 00，该函数返回整数：0x0b03d204

此编码将返回标准整数 LUN，适用于小于 256 的 LUN，它们通常使用单级 LUN 结构和地址方法 0。

**void int\_to\_scsilun(u64 lun, struct scsi\_lun \*scsilun)**

将整数转换回 scsi\_lun

1）参数

u64 lun

要还原的整数

struct scsi\_lun \*scsilun

要设置的 struct scsi\_lun。

2）说明

撤消 scsilun\_to\_int 的功能，该函数将 8 字节的 LUN 值打包到整数中。此例程将整数解包回 LUN 值。

3）注

给定整数：0x0b03d204，此函数返回 struct scsi\_lun：d2 04 0b 03 00 00 00 00

**bool scsi\_normalize\_sense(const u8 \*sense\_buffer, int sb\_len, struct scsi\_sense\_hdr \*sshdr)**

将固定或说明符感知数据格式的主要元素标准化为通用格式。

1）参数

const u8 \*sense\_buffer

包含由设备返回的感知数据的字节数组

int sb\_len

sense\_buffer 中有效字节数

struct scsi\_sense\_hdr \*sshdr

将写入常用元素的struct 的实例的指针。

2）注

感知数据的“主要元素”为：响应代码、感知密钥、asc、ascq 和 additional\_length（仅适用于说明符格式）。

通常，此函数可在设备用 CHECK\_CONDITION 状态响应 SCSI 命令后调用。

3）说明

返回：

如果找到有效的感知数据信息，则为 true，否则为 false；

**const u8 \*scsi\_sense\_desc\_find(const u8 \*sense\_buffer, int sb\_len, int desc\_type)**

在说明符感知数据格式中搜索指定的说明符类型。

1）参数

const u8 \* sense\_buffer

说明符格式感知数据的字节数组

int sb\_len

sense\_buffer 中有效字节数

int desc\_type

要查找的说明符类型值（例如 0 -> 信息）

2）注

仅当感知数据在说明符格式中有效时才有效

3）说明

返回：

如果找到，则为第一个说明符的起始指针，否则为 NULL

**void scsi\_build\_sense\_buffer(int desc, u8 \*buf, u8 key, u8 asc, u8 ascq)**

在缓冲区中构建感知数据

1）参数

int desc

感知格式（非零 == 说明符格式，0 == 固定格式）

u8 \*buf

构建感知数据的位置

u8 key

感知密钥

u8 asc

附加感知代码

u8 ascq

附加感知代码限定符

**int scsi\_set\_sense\_information(u8 \*buf, int buf\_len, u64 info)**

在格式化的感知数据缓冲区中设置信息字段

1）参数

u8 \*buf

构建感知数据的位置

int buf\_len

缓冲区长度

u64 info

要设置的 64 位信息值

2）说明

返回：

成功返回 0 或缓冲区长度无效时返回 -EINVAL

**int scsi\_set\_sense\_field\_pointer(u8 \*buf, int buf\_len, u16 fp, u8 bp, bool cd)**

在格式化的感知数据缓冲区中设置字段指针感知密钥特定信息

1）参数

u8 \*buf

构建感知数据的位置

int buf\_len

缓冲区长度

u16 fp

要设置的字段指针

u8 bp

要设置的位指针

bool cd

指令/数据位

2）**说明**

返回：

成功返回 0 或缓冲区长度无效时返回 -EINVAL

传输类

传输类是 SCSI 下层驱动程序的服务库，它们在 sysfs 中公开传输属性。

光纤通道传输

文件 drivers/scsi/scsi\_transport\_fc.c 定义了光纤通道的传输属性。

**u32 fc\_get\_event\_number(void)**

获取下一个顺序 FC 事件编号

1）参数

void

无参数

2）注

我们可以内联，但这将需要公开 fc\_event\_seq。暂时，接受子程序调用。原子用于避免锁定/解锁…

**void fc\_host\_post\_event(struct Scsi\_Host \*shost, u32 event\_number, enum fc\_host\_event\_code event\_code, u32 event\_data)**

在 fc\_host 上发布事件。

1）参数

struct Scsi\_Host \*shost

事件发生的主机

u32 event\_number

从 get\_fc\_event\_number() 获取的 FC 事件编号

枚举fc\_host\_event\_code事件代码

发布fc\_host事件

u32事件数据

发布的事件的32位数据

2）注释

这个函数在输入时假定没有任何锁定。

**void fc\_host\_post\_vendor\_event(struct Scsi\_Host \*shost, u32 event\_number, u32 data\_len, char \*data\_buf, u64 vendor\_id)**

在fc\_host上发布唯一的供应商事件

1）参数

struct Scsi\_Host \*shost

事件发生的主机

u32 event\_number

从get\_fc\_event\_number()获得的fc事件编号

u32 data\_len

供应商唯一数据的字节数

char \* data\_buf

指向供应商唯一数据的指针

u64 vendor\_id

供应商ID

2）注释

这个函数在输入时假定没有任何锁定。

**enum blk\_eh\_timer\_return fc\_eh\_timed\_out（struct scsi\_cmnd \*scmd）**

FC传输I/O超时拦截处理程序

1）参数

struct scsi\_cmnd \*scmd

超时的SCSI命令

2）说明

这个例程保护错误处理程序在rport处于阻塞状态时被唤醒，通常是因为临时丢失连接。如果允许错误处理程序继续，请求中止i/o，重置目标等将很可能失败，因为没有办法与设备通信执行所请求的功能。这些失败可能导致中间层将设备下线，需要手动干预才能恢复操作。

每当i/o超时时调用此例程，验证基础rport的状态。如果rport被阻止，则返回EH\_RESET\_TIMER，它将继续重新安排超时。最终，设备将返回，或者devloss\_tmo将触发，当超时时，它将以正常方式处理。如果rport没有被阻止，正常的错误处理将继续。

3）注释

这个函数在输入时假定没有任何锁定。

**void fc\_remove\_host(struct Scsi\_Host \*shost)**

被称为终止一个scsi主机的任何fc\_transport相关元素。

1）参数

struct Scsi\_Host \*shost

哪个Scsi\_Host

2）说明

这个例程被期望在一个驱动程序调用scsi\_remove\_host()之前立即被调用。

警告：使用fc\_transport的驱动程序，如果在scsi\_remove\_host()之前没有调用此例程，则会在/sys/class/fc\_remote\_ports中留下悬空对象。访问其中的任何对象可能导致系统崩溃！

3）注释

这个函数在输入时假定没有任何锁定。

**struct fc\_rport \*fc\_remote\_port\_add(struct Scsi\_Host \*shost, int channel, struct fc\_rport\_identifiers \*ids)**

通知fc传输存在远程FC端口。

1）参数

struct Scsi\_Host \*shost

远程端口所连接到的scsi主机。

int channel

连接到shost端口上的通道。

struct fc\_rport\_identifiers \*ids

远程端口的世界范围名称、fc地址和FC4端口角色。

2）说明

LLDD调用此函数通知传输存在远程端口。LLDD提供端口的唯一标识符（wwpn、wwn）、它的FC地址（port\_id）以及激活的FC4角色。

对于FCP目标（也称为scsi目标）的端口，FC传输代表LLDD维护一致的目标id绑定。一致的目标id绑定是将目标id分配给远程端口标识符的一种分配方法，该分配在附加的scsi主机时持续存在。远程端口可能消失，然后再次出现，它的目标ID分配仍然相同。这允许FC寻址的变化（如果按照wwpn或wwnn绑定）而不对基于scsi主机号和目标ID值的scsi子系统造成任何明显的更改。绑定只在附加scsi主机期间有效。如果主机分离，然后稍后重新附加，目标id绑定可能会改变。

此例程负责返回远程端口结构。该例程会在其内部维护的一致目标ID映射列表中搜索。如果找到，则将重新使用远程端口结构。否则，将分配一个新的远程端口结构。

每当分配远程端口时，都会创建一个新的fc\_remote\_port类设备。

不应从中断上下文中调用。

3）注释

这个函数在输入时假定没有任何锁定。

void fc\_remote\_port\_delete(struct fc\_rport \*rport)

通知fc传输某个远程端口不再存在。

1）参数

struct fc\_rport \*rport

不再存在的远程端口

2）说明

LLDD调用此例程通知传输，某个远程端口不再是拓扑的一部分。请注意：虽然某个端口可能不再是拓扑的一部分，但它可能仍然存在于fc\_host显示的远程端口中。我们在以下情况下这样做：

如果该端口是scsi目标，则通过“阻止”它来延迟其删除。这允许端口在暂时消失、然后重新出现时不会破坏附加到其上的SCSI设备树。在“阻止”期间，端口仍然存在。

如果端口是SCSI目标并且消失的时间比我们预期的更长，我们将删除该端口并撤销附加到该端口的SCSI设备树。但是，如果最终存在该端口，我们希望半持久化分配给该端口的目标ID。端口结构将保持不变（尽管只有最少的信息），以便目标ID绑定也保持不变。

如果远程端口不是FCP目标，则将完全撤销和释放fc\_remote\_port类设备。

如果远程端口是FCP目标，则将该端口放置在临时阻止状态。从LLDD的角度来看，rport不再存在。从SCSI中间层的角度来看，SCSI目标存在，但所有附加在其上的sdev都被阻止进行进一步的I/O。然后期望如下。

如果远程端口没有返回（通过LLDD调用fc\_remote\_port\_add()发出信号）并在dev\_loss\_tmo超时内，则将删除SCSI目标-杀死所有未完成的I/O并删除附加到其上的SCSI设备。端口结构将被标记为不存在，并且会部分清除，只留下足够的信息以识别远程端口相对于SCSI目标ID绑定，如果它后来出现。只要有有效的绑定（例如，直到用户更改绑定类型或卸载具有绑定的SCSI主机），端口就会保持不变。

如果远程端口在dev\_loss\_tmo值内返回（并根据目标ID绑定类型进行匹配），则将重复使用该端口结构。如果它不再是SCSI目标，则将删除目标。如果它仍然是SCSI目标，则将解除阻止（允许恢复I/O），并激活扫描以确保检测到所有逻辑单元。

仅从正常进程上下文调用-不能从中断调用。

3）笔记

此例程假定输入时未保持任何锁定。

**void fc\_remote\_port\_rolechg（struct fc\_rport \* rport，u32 roles）**

通知FC运输角色在远程上可能已更改。

1）参数

struct fc\_rport \* rport

更改的远程端口。

u32 roles

此端口的新角色。

2）说明

LLDD调用此例程以通知传输角色在远程端口上可能已更改。这的最大影响是，如果某个端口现在变为FCP目标，则必须为其分配SCSI目标ID。如果端口不再是FCP目标，则分配给它的任何SCSI目标ID值将保留，以防角色再次更改以包括FCP目标。如果角色更改，SCSI中间层中不会引发任何更改（预计角色将被恢复。如果没有，将发生正常的错误处理）。

不应从中断上下文调用。

3）笔记

此例程假定输入时未保持任何锁定。

**int fc\_block\_rport（struct fc\_rport \* rport**）

为阻止的fc\_rport阻止SCSI eh线程。

1）参数

struct fc\_rport \* rport

SCSI\_eh正在尝试恢复的远程端口。

2）说明

此例程可以从FC LLD scsi\_eh回调中调用。它阻止scsi\_eh线程，直到fc\_rport离开FC\_PORTSTATE\_BLOCKED状态或fast\_io\_fail\_tmo启动。这是必要的，以避免scsi\_eh对已阻止的rports失败恢复操作，这将导致已脱机的SCSI设备。

3）返回

如果fc\_rport离开状态FC\_PORTSTATE\_BLOCKED，则返回0。

FAST\_IO\_FAIL如果fast\_io\_fail\_tmo已触发，则应将其传回scsi\_eh。

**int fc\_block\_scsi\_eh（struct scsi\_cmnd \* cmnd）**

为阻止的fc\_rport阻止SCSI eh线程

1）参数

struct scsi\_cmnd \* cmnd

scsi\_eh正在尝试恢复的SCSI命令。

2）说明

此例程可以从FC LLD scsi\_eh回调中调用。它阻止scsi\_eh线程，直到fc\_rport离开FC\_PORTSTATE\_BLOCKED状态或fast\_io\_fail\_tmo启动。这是必要的，以避免scsi\_eh对已阻止的rports失败恢复操作，这将导致已脱机的SCSI设备。

3）返回

如果fc\_rport离开状态FC\_PORTSTATE\_BLOCKED，则返回0。

FAST\_IO\_FAIL如果fast\_io\_fail\_tmo已触发，则应将其传回scsi\_eh。

**struct fc\_vport \* fc\_vport\_create(struct Scsi\_Host \* shost，int channel，struct fc\_vport\_identifiers \* ids)**

管理应用程序或LLDD请求创建vport

1）参数

struct Scsi\_Host \* shost

虚拟端口连接到的scsi主机。

int channel

连接到shost端口的通道。

struct fc\_vport\_identifiers \* ids

虚拟端口的世界范围名称，FC4端口角色等。

2）注释

此例程假定输入时未保持任何锁定。

**int fc\_vport\_terminate(struct fc\_vport \* vport)**

管理应用程序或LLDD请求终止vport

1）参数

struct fc\_vport \* vport

要终止的fc\_vport

2）说明

调用LLDD vport\_delete()函数，然后从shost和对象树中分配并移除vport。

3）注意事项

此例程假定在进入时未保持锁定状态。

### iSCSI传输类

文件drivers / scsi / scsi\_transport\_iscsi.c为iSCSI类定义传输属性，该类通过TCP / IP连接发送SCSI数据包。

**struct iscsi\_bus\_flash\_session \* iscsi\_create\_flashnode\_sess（struct Scsi\_Host \* shost，int index，struct iscsi\_transport \* transport，int dd\_size）**

在sysfs中添加flashnode会话条目

1）参数

struct Scsi\_Host \* shost

主机数据指针

int index

要在sysfs中添加的flashnode的索引

struct iscsi\_transport \* transport

传输数据指针

int dd\_size

要分配的总大小

2）说明

为flashnode会话属性添加sysfs条目

3）返回

成功时分配的flashnode sess指针，失败时为NULL

**struct iscsi\_bus\_flash\_conn \* iscsi\_create\_flashnode\_conn（struct Scsi\_Host \* shost，struct iscsi\_bus\_flash\_session \* fnode\_sess，struct iscsi\_transport \* transport，int dd\_size）**

在sysfs中添加flashnode连接条目

1）参数

struct Scsi\_Host \* shost

主机数据指针

struct iscsi\_bus\_flash\_session \* fnode\_sess

父flashnode会话条目指针

struct iscsi\_transport \* transport

传输数据指针

int dd\_size

要分配的总大小

2）说明

为flashnode连接属性添加sysfs条目

3）返回

成功时分配的flashnode conn指针，失败时为NULL

**struct device \* iscsi\_find\_flashnode\_sess（struct Scsi\_Host \* shost，void \* data，int（\* fn）（struct device \* dev，void \* data））**

查找flashnode会话条目

1）参数

struct Scsi\_Host \* shost

主机数据指针

void \* data

包含用于比较的值的数据指针

int（\* fn）（struct device \* dev，void \* data）

执行实际比较的函数指针

2）说明

查找flashnode会话对象，比较传递的数据使用在传递的函数指针中定义的逻辑

3）返回

成功时找到的flashnode会话设备对象指针，失败时为NULL

**struct device \* iscsi\_find\_flashnode\_conn（struct iscsi\_bus\_flash\_session \* fnode\_sess）**

查找flashnode连接条目

1）参数

struct iscsi\_bus\_flash\_session \* fnode\_sess

父flashnode会话条目指针

2）说明

查找flashnode连接对象，比较传递的数据使用在传递的函数指针中定义的逻辑

3）返回

成功时找到的flashnode连接设备对象指针，失败时为NULL

**void iscsi\_destroy\_flashnode\_sess（struct iscsi\_bus\_flash\_session \* fnode\_sess）**

销毁flashnode会话条目

1）参数

struct iscsi\_bus\_flash\_session \* fnode\_sess

要销毁的flashnode会话条目指针

2）说明

从sysfs中删除flashnode会话条目和所有子flashnode连接条目

**void iscsi\_destroy\_all\_flashnode（struct Scsi\_Host \* shost）**

销毁所有flashnode会话条目

1）参数

struct Scsi\_Host \* shost

主机数据指针

2）说明

从sysfs中销毁所有flashnode会话条目和所有对应的子flashnode连接条目

**int iscsi\_scan\_finished（struct Scsi\_Host \* shost，unsigned long time）**

辅助函数，用于报告运行扫描完成的时间

1）参数

struct Scsi\_Host \* shost

SCSI主机

unsigned long time

扫描运行时间

2）**说明**

此函数可由类似qla4xxx的驱动程序使用，报告扫描在模块加载时启动的时间到SCSI层时已完成。

**int iscsi\_block\_scsi\_eh（struct scsi\_cmnd \* cmd）**

阻止SCSI eh，直到会话状态已转换

1）参数

struct scsi\_cmnd \* cmd

传递给SCSI eh处理程序的SCSI命令

2）说明

如果会话已关闭，则此函数将等待恢复计时器触发或会话重新登录。如果恢复计时器启动，则返回FAST\_IO\_FAIL。调用者应将此错误值传递给SCSI eh。

**void iscsi\_unblock\_session（struct iscsi\_cls\_session \* session）**

将会话设置为登录并启动IO。

1）参数

struct iscsi\_cls\_session \* session

iscsi会话

2）说明

将会话标记为准备好接受IO。

**struct iscsi\_cls\_session \* iscsi\_create\_session（struct Scsi\_Host \* shost，struct iscsi\_transport \* transport，int dd\_size，unsigned int target\_id）**

创建iscsi类会话

1）参数

struct Scsi\_Host \* shost

SCSI主机

struct iscsi\_transport \* transport

iscsi传输

int dd\_size

私有驱动程序数据大小

unsigned int target\_id

目标id

2）说明

可以从LLD或iscsi\_transport中调用。

s**truct iscsi\_cls\_conn \* iscsi\_create\_conn（struct iscsi\_cls\_session \* session，int dd\_size，uint32\_t cid）**

创建iscsi类连接

1）参数

struct iscsi\_cls\_session \* session

iscsi cls会话

int dd\_size

私有驱动程序数据大小

uint32\_t cid

连接id

2）说明

可以从LLD或iscsi\_transport中调用。连接是会话的子级，因此cid必须对会话中的所有连接都是唯一的。

由于我们不支持MCS，cid通常为零。在一些软件iscsi的情况下，我们可能会尝试预分配连接结构，因此可能会有两个连接结构，并且cid将不为零。

**int iscsi\_destroy\_conn（struct iscsi\_cls\_conn \* conn）**

销毁iscsi类连接

1）参数

struct iscsi\_cls\_conn \* conn

iscsi cls会话

2）说明

可以从LLD或iscsi\_transport中调用。

**int iscsi\_session\_event（struct iscsi\_cls\_session \* session，enum iscsi\_uevent\_e event）**

发送会话destr。完成事件

1）参数

struct iscsi\_cls\_session \*session

iSCSI类会话

enum iscsi\_uevent\_e event

事件类型

### 串行附加SCSI（SAS）传输类

文件drivers/scsi/scsi\_transport\_sas.c定义了串行附加SCSI的传输属性，这是一种针对大型高端系统的SATA变体。

SAS传输类包含用于处理SAS HBA的通用代码，驱动程序模型中SAS拓扑的近似表示，以及各种sysfs属性，以向用户空间公开这些拓扑和管理接口。

除了基本的SCSI核心对象外，这个传输类还引入了两个附加的中间对象：“出站”SAS PHY由struct sas\_phy表示，用于SAS HBA或扩展器上，“传入”SAS远程PHY由struct sas\_rphy表示，用于SAS扩展器或终端设备上。请注意，这仅仅是一个软件概念，PHY和远程PHY的底层硬件是完全相同的。

这个代码中没有SAS端口的概念，用户可以根据port\_identifier属性看到哪些PHY形成了广域端口，该属性对于端口中的所有PHY是相同的。

**void sas\_remove\_children(struct device \*dev)**

撤销一个设备的SAS数据结构

1）Parameters

struct device \*dev

属于SAS对象的设备

2）Description

撤销给定对象的所有SAS PHY和远程PHY。

**void sas\_remove\_host(struct Scsi\_Host \*shost)**

撤销一个Scsi\_Host的SAS数据结构

1）参数

struct Scsi\_Host \*shost

将被撤销的Scsi Host

2）说明

撤销给定的Scsi\_Host的所有SAS PHY和远程PHY，并删除Scsi\_Host。

3）注意

不要再对Scsi\_Host调用scsi\_remove\_host()，因为它已经被删除了。

**u64 sas\_get\_address(struct scsi\_device \*sdev)**

返回设备的SAS地址

1）参数

struct scsi\_device \*sdev

scsi设备

2）说明

返回scsi设备的SAS地址。

**unsigned int sas\_tlr\_supported(struct scsi\_device \*sdev)**

检查vpd 0x90中的TLR位

1）参数

struct scsi\_device \*sdev

scsi设备struct

2）说明

检查传输层重试是否受支持。如果vpd页面0x90存在，则支持传输层重试。

**void sas\_disable\_tlr(struct scsi\_device \*sdev)**

设置TLR标志位

1）参数

struct scsi\_device \*sdev

scsi设备struct

2）说明

将tlr\_enabled标志位设置为0。

**void sas\_enable\_tlr(struct scsi\_device \*sdev)**

设置TLR标志位

1）参数

struct scsi\_device \*sdev

scsi设备struct

2）说明

将tlr\_enabled标志位设置为1。

**struct sas\_phy \*sas\_phy\_alloc(struct device \*parent, int number)**

分配并初始化一个SAS PHYstruct

1）参数

struct device \*parent

父设备

int number

PHY索引

2）说明

分配SAS PHYstruct 。它将被添加到由parent指定的设备树中，其必须是Scsi\_Host或sas\_rphy。

3）返回

分配的SAS PHY或空指针，如果分配失败。

**int sas\_phy\_add(struct sas\_phy \*phy)**

将SAS PHY添加到设备层次结构中

1）参数

struct sas\_phy \*phy

要添加的PHY

2）说明

向系统中发布SAS PHY。

**void sas\_phy\_free(struct sas\_phy \*phy)**

释放SAS PHY

1）参数

struct sas\_phy \*phy

要释放的SAS PHY

2）说明

释放指定的SAS PHY。

3）注意

只有尚未通过sas\_phy\_add()成功添加的PHY才能调用此函数。

**void sas\_phy\_delete(struct sas\_phy \*phy)**

删除SAS PHY

1）参数

struct sas\_phy \*phy

要删除的SAS PHY

2）说明

删除指定的SAS PHY。如果该SAS PHY有相关的远程PHY，则在删除之前取消链接。

**int scsi\_is\_sas\_phy(const struct device \*dev)**

检查struct device是否表示SAS PHY

1）参数

const struct device \*dev

要检查的设备

2）返回

如果设备表示SAS PHY，则返回1，否则返回0。

**int sas\_port\_add(struct sas\_port \*port)**

将SAS端口添加到设备层次结构中

1）参数

struct sas\_port \*port

要添加的端口

2）说明

将端口发布到系统的其余部分。

**void sas\_port\_free(struct sas\_port \*port)**

释放SAS端口

1）参数

struct sas\_port \*port

要释放的SAS端口

2）说明

释放指定的SAS端口。

3）注意

只有尚未通过sas\_port\_add()成功添加的端口才能调用此函数。

**void sas\_port\_delete(struct sas\_port \*port)**

删除SAS端口

1）参数

struct sas\_port \*port

要删除的SAS端口

2）说明

删除指定的SAS端口。如果该SAS端口有相关的PHY，则也要将它们从端口中断开链接。

**int scsi\_is\_sas\_port(const struct device \*dev)**

检查struct device是否表示SAS端口

1）参数

const struct device \*dev

要检查的设备

2）返回

如果该设备表示SAS端口，则返回1，否则返回0。

**struct sas\_phy \*sas\_port\_get\_phy(struct sas\_port \*port)**

尝试在端口成员上引用

1）参数

struct sas\_port \*port

要检查的端口

**void sas\_port\_add\_phy(struct sas\_port \*port, struct sas\_phy \*phy)**

添加另一个物理端口以形成宽端口

1）参数

struct sas\_port \*port

要将端口添加到的端口

struct sas\_phy \*phy

要添加的物理端口

2）说明

当端口最初创建时，它是空的（没有物理端口）。所有端口都必须至少有一个物理端口才能运行，所有宽端口都必须至少有两个。当前的代码没有区分端口和宽端口，但是唯一可以连接到远程设备的对象是端口，因此如果与任何东西连接，所有具有物理端口的设备都必须形成端口。

**void sas\_port\_delete\_phy(struct sas\_port \*port, struct sas\_phy \*phy)**

从端口或宽端口中删除物理端口

1）参数

struct sas\_port \*port

要从中删除物理端口的端口

struct sas\_phy \*phy

要删除的物理端口

2）说明

此操作用于拆除端口。在调用 sas\_port\_delete 之前，必须对每个端口或宽端口进行此操作。

**struct sas\_rphy \*sas\_end\_device\_alloc(struct sas\_port \*parent)**

为端(终)设备分配 rphy

1）参数

struct sas\_port \*parent

哪个端口

2）说明

为连接到父端口的 SAS 远程 PHY 结构分配空间。

3）返回

分配的 SAS PHY 或如果分配失败则为 NULL。

**struct sas\_rphy \*sas\_expander\_alloc(struct sas\_port \*parent, enum sas\_device\_type type)**

为扩展器设备分配 rphy

1）参数

struct sas\_port \*parent

哪个端口

enum sas\_device\_type type

SAS\_EDGE\_EXPANDER\_DEVICE 或 SAS\_FANOUT\_EXPANDER\_DEVICE

2）说明

为连接到父端口的 SAS 远程 PHY 结构分配空间。

3）返回

分配的 SAS PHY 或如果分配失败则为 NULL。

**int sas\_rphy\_add(struct sas\_rphy \*rphy)**

将 SAS 远程 PHY 添加到设备层次结构

1）参数

struct sas\_rphy \*rphy

要添加的远程 PHY

2）说明

向系统中发布 SAS 远程 PHY。

**void sas\_rphy\_free(struct sas\_rphy \*rphy)**

释放 SAS 远程 PHY

1）参数

struct sas\_rphy \*rphy

要释放的 SAS 远程 PHY

2）说明

释放指定的 SAS 远程 PHY。

3）注意

此函数只能在尚未使用 sas\_rphy\_add() 成功添加（或已使用 sas\_rphy\_remove()）的远程 PHY 上调用。

**void sas\_rphy\_delete(struct sas\_rphy \*rphy)**

删除并释放 SAS 远程 PHY

1）参数

struct sas\_rphy \*rphy

要删除和释放的 SAS 远程 PHY

2）说明

删除指定的 SAS 远程 PHY 并释放它。

**void sas\_rphy\_unlink(struct sas\_rphy \*rphy)**

取消链接 SAS 远程 PHY

1）参数

struct sas\_rphy \*rphy

要从其父端口取消链接的 SAS 远程 PHY

2）说明

删除端口对 rphy 的引用。

**void sas\_rphy\_remove(struct sas\_rphy \*rphy)**

删除 SAS 远程 PHY

1）参数

struct sas\_rphy \*rphy

要删除的 SAS 远程 PHY

2）说明

删除指定的 SAS 远程 PHY。

**int scsi\_is\_sas\_rphy(const struct device \*dev)**

检查结构 device 是否表示 SAS 远程 PHY

1）参数

const struct device \*dev

要检查的设备

2）返回

1 表示设备表示 SAS 远程 PHY，0 表示否

**struct scsi\_transport\_template \*sas\_attach\_transport(struct sas\_function\_template \*ft)**

实例化 SAS 传输模板

1）参数

struct sas\_function\_template \*ft

SAS 传输类函数模板

**void sas\_release\_transport(struct scsi\_transport\_template \*t)**

释放 SAS 传输模板实例

1）参数

struct scsi\_transport\_template \*t

传输模板实例

### SATA 传输类

SATA 传输由 libata 处理，该库有自己的文档。

### 并行 SCSI（SPI）传输类

文件 drivers/scsi/scsi\_transport\_spi.c 定义了传统（快速/宽/超大）SCSI 总线的传输属性。

**void spi\_schedule\_dv\_device(struct scsi\_device \*sdev)**

调度域验证以在该设备上进行

1）参数

struct scsi\_device \*sdev

要验证的设备

与上面的 spi\_dv\_device() 相同，只是 DV 将被调度以在稍后的工作队列中进行。所有内存分配都是原子性的，因此可以从任何上下文（包括持有 SCSI 锁的上下文）中调用它们。

**void spi\_display\_xfer\_agreement(struct scsi\_target \*starget)**

打印当前的目标传输协议

1）参数

struct scsi\_target \*starget

要显示协议的目标

2）说明

每个 SPI 端口都必须维护一份用于总线上的每个其他端口的传输协议。此函数打印当前协议的一行摘要；更详细的信息可在 sysfs 中获得。

**int spi\_populate\_tag\_msg(unsigned char \*msg, struct scsi\_cmnd \*cmd)**

将标记消息放置到缓冲区中

1）参数

unsigned char \*msg

放置标记的区域指针

struct scsi\_cmnd \*cmd

用于标记的 scsi 命令的指针

2）说明

旨在为特定请求创建正确类型的标记消息。返回标记消息的大小。如果此设备禁用 TCQ，则可能返回 0。

### SCSI RDMA（SRP）传输类

文件 drivers/scsi/scsi\_transport\_srp.c 定义了 SCSI over Remote Direct Memory Access 的传输属性。

**int srp\_tmo\_valid(int reconnect\_delay, int fast\_io\_fail\_tmo, long dev\_loss\_tmo)**

检查超时组合的有效性

1）参数

int reconnect\_delay

重新连接延迟时间（秒）。

int fast\_io\_fail\_tmo

快速 I/O 失败超时时间（秒）。

long dev\_loss\_tmo

设备丢失超时时间（秒）。

2）说明

超时参数的组合必须使 SCSI 命令在合理的时间内完成。因此，不要允许快速 I/O 失败超时时间超过 SCSI\_DEVICE\_BLOCK\_MAX\_TIMEOUT，并且如果已禁用快速执行失败，则不要允许 dev\_loss\_tmo 超过该限制。此外，这些参数必须使多路径能够及时检测到失效路径。因此，请不要同时禁用所有三个参数。

**void srp\_start\_tl\_fail\_timers(struct srp\_rport \*rport)**

启动传输层故障定时器

1）参数

struct srp\_rport \*rport

SRP 目标端口。

2）说明

启动传输层快速 I/O 失败和设备丢失定时器。不要修改已经启动的定时器。

**int srp\_reconnect\_rport(struct srp\_rport \*rport)**

重新连接 SRP 目标端口

1）参数

struct srp\_rport \*rport

SRP 目标端口。

2）说明

在调用 reconnect() 之前阻塞 SCSI 命令队列，这样从 SCSI EH 外部并发地调用 queuecommand() 就不会被启动。这很重要，因为 reconnect() 实现可能会重新分配 queuecommand() 需要的资源。

3）注解

此函数既不等待未完成请求的完成，也不尝试中止这些请求。在重新连接到目标端口之前，reconnect() 函数的责任是完成未完成的命令。

调用者有责任确保在此函数执行过程中不使用 reconnect() 函数重新分配的资源。一种可能的策略是仅从 SCSI EH 线程上下文中调用该函数。另一种可能的策略是在 SCSI LLD 回调的每个 SCSI EH（scsi\_host\_template.eh\_\*() 函数和 scsi\_host\_template.queuecommand() 函数）内部锁定 rport 互斥锁。

**enum blk\_eh\_timer\_return srp\_timed\_out(struct scsi\_cmnd \*scmd)**

SCSI 超时 EH 的 SRP 传输拦截器

1）参数

struct scsi\_cmnd \*scmd

SCSI 命令。

2）说明

如果在 rport 处于阻塞状态时超时发生，请请求 SCSI EH 继续等待（BLK\_EH\_RESET\_TIMER）。否则让 SCSI 核心处理超时（BLK\_EH\_DONE）。

3）注解

此函数在 soft-IRQ 上下文中调用，并保持请求队列锁。

**void srp\_rport\_get(struct srp\_rport \*rport)**

增加 rport 引用计数

1）参数

struct srp\_rport \*rport

SRP 目标端口。

**void srp\_rport\_put(struct srp\_rport \*rport)**

减少 rport 引用计数

1）参数

struct srp\_rport \*rport

SRP 目标端口。

**struct srp\_rport \*srp\_rport\_add(struct Scsi\_Host \*shost, struct srp\_rport\_identifiers \*ids)**

将 SRP 远程端口添加到设备层次结构中

1）参数

struct Scsi\_Host \*shost

连接远程端口的 SCSI 主机。

struct srp\_rport\_identifiers \*ids

远程端口的端口 id。

2）说明

将一个端口发布到系统的其他部分。

**void srp\_rport\_del(struct srp\_rport \*rport)**

删除 SRP 远程端口

1）参数

struct srp\_rport \*rport

要删除的 SRP 远程端口

2）说明

删除指定的 SRP 远程端口。

**void srp\_remove\_host(struct Scsi\_Host \*shost)**

拆除 Scsi\_Host 的 SRP 数据结构

1）参数

struct Scsi\_Host \*shost

要拆除的 Scsi Host

2）说明

删除给定 Scsi\_Host 的所有 SRP 远程端口。在 SRP HBA 的 SCSI 移除主机之前必须调用。

**void srp\_stop\_rport\_timers(struct srp\_rport \*rport)**

停止传输层恢复计时器

1）参数

struct srp\_rport \*rport

要停止计时器的 SRP 远程端口。

2）说明

必须在 srp\_remove\_host() 和 scsi\_remove\_host() 之后调用。调用者必须对 rport（rport->dev）和 SCSI 主机（rport->dev.parent）保持引用。

**struct scsi\_transport\_template \*srp\_attach\_transport(struct srp\_function\_template \*ft)**

实例化 SRP 传输模板

1）参数

struct srp\_function\_template \*ft

SRP 传输类模板函数

**void srp\_release\_transport(struct scsi\_transport\_template \*t)**

释放 SRP 传输模板实例

参数

struct scsi\_transport\_template \*t

传输模板实例

## SCSI 较低层

主机总线适配器传输类型

许多现代设备控制器使用 SCSI 命令集作为协议，通过许多不同类型的物理连接与它们的设备通信。

在 SCSI 语言中，一个能够承载 SCSI 命令的总线被称为“传输”，连接到这样的总线的控制器被称为“主机总线适配器”（HBA）。

### 调试传输

文件 drivers/scsi/scsi\_debug.c 模拟带有可变数量的磁盘（或类似磁盘的设备）连接的主机适配器，共享一定量的 RAM。它做了很多检查以确保我们没有混淆块，并在看到任何异常情况时使内核崩溃。

更加真实的是，模拟设备具有 SAS 磁盘的传输属性。

有关文档请参见http://sg.danny.cz/sg/sdebug26.html

### 待办事项

并行（快/宽/超）SCSI，USB，SATA，SAS，纤维通道，FireWire，ATAPI设备，Infiniband，I2O，并行端口，netlink…

# libATA开发人员指南

## 介绍

libATA是一个用于支持ATA主机控制器和设备的Linux内核中使用的库。 libATA提供ATA驱动程序API，ATA和ATAPI设备的类传输以及根据T10 SAT规格说明书为ATA设备提供SCSI <-> ATA转换。

此指南记录了libATA驱动程序API，库函数，库内部以及一些示例ATA低级驱动程序。

## libata驱动程序API

针对每个低级libata硬件驱动程序定义struct ata\_port\_operations，并控制低级驱动程序与ATA和SCSI层的交互。

基于FIS的驱动程序将使用->qc\_prep()和->qc\_issue()高级钩子连接系统。类似于PCI IDE硬件的行为的硬件可以使用多个通用帮助程序，至少定义ATA阴影注册块的总线I/O地址。

**struct ata\_port\_operations**

### 禁用ATA端口

void (\*port\_disable) (struct ata\_port \*);

在ata\_bus\_probe()错误路径以及从SCSI模块（rmmod，热插拔）注销时调用。此函数应该执行需要执行的操作以停用端口。在大多数情况下，ata\_port\_disable()可以用作此钩子。

在探测失败时从ata\_bus\_probe()调用。在ata\_scsi\_release()中调用。

### 后标识符设备配置

void (\*dev\_config) (struct ata\_port \*, struct ata\_device \*);

在发现每个设备之后发出标识符[PACKET] DEVICE之后调用。通常用于在发出SET FEATURES-XFER MODE和操作之前应用特定于设备的修复程序。

此条目可能在ata\_port\_operations中指定为NULL。

### 设置PIO / DMA模式

void (\*set\_piomode) (struct ata\_port \*, struct ata\_device \*);

void (\*set\_dmamode) (struct ata\_port \*, struct ata\_device \*);

void (\*post\_set\_mode) (struct ata\_port \*);

unsigned int (\*mode\_filter) (struct ata\_port \*, struct ata\_device \*, unsigned int);

在发出SET FEATURES-XFER MODE命令之前调用的钩子。当libata构建可能模式的掩码时，可选的->mode\_filter()钩子被调用。这被传递给->mode\_filter()函数，该函数应返回在由于硬件限制而过滤掉不适合的模式后的有效模式的掩码。不允许使用此接口添加模式。

当调用->set\_piomode()和->set\_dmamode()时，dev->pio\_mode和dev->dma\_mode是保证有效的。此时，任何共用电缆的其他驱动器的定时也将有效。也就是说，在尝试设置它们之前，库将记录每个通道上每个驱动器的模式的决策。

在SET FEATURES-XFER MODE命令成功完成后，无条件调用->post\_set\_mode()。

->set\_piomode()始终调用（如果存在），但只有在DMA可用时才调用->set\_dma\_mode()。

### 任务文件读/写

void (\*sff\_tf\_load) (struct ata\_port \*ap, struct ata\_taskfile \*tf);

void (\*sff\_tf\_read) (struct ata\_port \*ap, struct ata\_taskfile \*tf);

->tf\_load()用于将给定的任务文件加载到硬件寄存器/ DMA缓冲区中。->tf\_read()用于读取硬件寄存器/ DMA缓冲区，以获取当前的任务文件寄存器值集。PIO或MMIO的大多数任务文件型硬件驱动程序使用ata\_sff\_tf\_load()和ata\_sff\_tf\_read()来进行这些钩子。

### PIO数据读/写

void (\*sff\_data\_xfer) (struct ata\_device \*, unsigned char \*, unsigned int, int);

所有bmdma样式的驱动程序都必须实现此钩子。这是实际在PIO数据传输期间复制数据字节的底层操作。通常，驱动程序将选择ata\_sff\_data\_xfer()或ata\_sff\_data\_xfer32()之一。

### ATA命令执行

void (\*sff\_exec\_command)(struct ata\_port \*ap, struct ata\_taskfile \*tf);

导致先前使用->tf\_load（）加载的ATA命令在硬件中启动。大多数任务文件型硬件的驱动程序使用ata\_sff\_exec\_command()进行此钩子。

### 每个命令的ATAPI DMA能力过滤器

int (\*check\_atapi\_dma) (struct ata\_queued\_cmd \*qc);

允许底层驱动程序过滤ATA PACKET命令，返回指示是否可以使用提供的PACKET命令的DMA的状态。

此钩子可以指定为NULL，在这种情况下，libata会假定可以支持atapi dma。

### 读取特定ATA阴影寄存器

u8 (\*sff\_check\_status)(struct ata\_port \*ap);

u8 (\*sff\_check\_altstatus)(struct ata\_port \*ap);

从硬件中读取Status / AltStatus ATA阴影寄存器。在某些硬件上，读取状态寄存器的副作用是清除中断条件。大多数任务文件型硬件的驱动程序使用ata\_sff\_check\_status()进行此钩子。

### 写入特定的ATA阴影寄存器

void (\*sff\_set\_devctl)(struct ata\_port \*ap, u8 ctl);

将设备控件ATA阴影寄存器写入硬件中。大多数驱动程序不需要定义此功能。

### 选择ATA总线上的设备

void (\*sff\_dev\_select)(struct ata\_port \*ap, unsigned int device);

发出低级硬件命令，导致N个硬件设备中的一个被视为“选定”（活动且可供使用）在ATA总线上。这通常在FIS型设备上没有意义。

大多数基于任务文件的硬件驱动程序使用ata\_sff\_dev\_select()完成此挂钩。

### 私有调整方法

void (\*set\_mode) (struct ata\_port \*ap);

默认情况下，libata根据ATA计时规则执行驱动器和控制器调整，并应用黑名单和电缆限制。一些控制器需要特殊处理，并具有自定义调整规则，通常是使用ATA命令但实际上不进行驱动器定时的RAID控制器。

**警告**

当控制器存在特殊问题时，不应使用此挂钩替换标准控制器调整逻辑。在这种情况下替换默认调整逻辑将绕过对可能重要的驱动器和桥接异常处理的处理。如果控制器需要过滤模式选择，则应使用mode\_filter挂钩。

### 控制PCI IDE BMDMA引擎

void (\*bmdma\_setup) (struct ata\_queued\_cmd \*qc);

void (\*bmdma\_start) (struct ata\_queued\_cmd \*qc);

void (\*bmdma\_stop) (struct ata\_port \*ap);

u8 (\*bmdma\_status) (struct ata\_port \*ap);

在设置IDE BMDMA事务时，这些挂钩（->bmdma\_setup）装载、执行（->bmdma\_start）和停止（->bmdma\_stop）硬件的DMA引擎，->bmdma\_status用于读取标准PCI IDE DMA状态寄存器。

这些挂钩通常是无操作或在基于FIS的驱动程序中根本没有实现的。

大多数遗留的IDE驱动程序使用ata\_bmdma\_setup()作为bmdma\_setup()挂钩。 ata\_bmdma\_setup()会将指向PRD表的指针写入IDE PRD表地址寄存器，启用DMA命令寄存器中的DMA并调用exec\_command()开始传输。

大多数遗留的IDE驱动程序使用ata\_bmdma\_start()作为bmdma\_start()挂钩。 ata\_bmdma\_start()会将ATA\_DMA\_START标志写入DMA命令寄存器。

许多遗留的IDE驱动程序使用ata\_bmdma\_stop()作为bmdma\_stop()挂钩。 ata\_bmdma\_stop()会清除DMA命令寄存器中的ATA\_DMA\_START标志。

许多遗留的IDE驱动程序使用ata\_bmdma\_status()作为bmdma\_status()挂钩。

### 高级任务文件挂钩

void (\*qc\_prep) (struct ata\_queued\_cmd \*qc);

int (\*qc\_issue) (struct ata\_queued\_cmd \*qc);

这两个挂钩是高级的，可以潜在地取代上述任务文件/DMA引擎挂钩中的几个。->qc\_prep在缓冲区被DMA映射后调用，通常用于填充硬件的DMA散射聚集表。一些驱动程序使用标准的ata\_bmdma\_qc\_prep()和ata\_bmdma\_dumb\_qc\_prep()辅助函数，但更高级的驱动程序会自己编写。

->qc\_issue用于使命令处于活动状态，一旦硬件和S/G表已经准备好。 IDE BMDMA驱动程序使用辅助函数ata\_sff\_qc\_issue()进行任务文件协议调度。更高级的驱动程序实现自己的->qc\_issue。

ata\_sff\_qc\_issue()调用->sff\_tf\_load()，->bmdma\_setup()和->bmdma\_start()，以便根据需要启动传输。

### 异常和探针处理（EH）

void (\*eng\_timeout) (struct ata\_port \*ap);

void (\*phy\_reset) (struct ata\_port \*ap);

已弃用。请改用->error\_handler()。

void (\*freeze) (struct ata\_port \*ap);

void (\*thaw) (struct ata\_port \*ap);

当HSM违规或其他条件干扰端口的正常操作时，会调用ata\_port\_freeze()。冻结的端口不允许执行任何操作，直到端口恢复正常操作，通常是在成功重置后。

可选的->freeze()回调可用于硬件冻结端口（例如，屏蔽中断并停止DMA引擎）。如果无法对端口进行硬件冻结，则中断处理程序必须在端口被冻结时无条件地确认和清除中断。

->thaw()回调可用于执行->freeze()的相反操作：再次为正常操作准备端口。解除屏蔽中断，启动DMA引擎等。

void (\*error\_handler) (struct ata\_port \*ap);

->error\_handler()是驱动程序对探测、热插拔、恢复和其他异常情况的挂钩。实现的主要责任是使用一组EH挂钩调用ata\_do\_eh()或ata\_bmdma\_drive\_eh()：

在执行EH重置之前，在EH重置中调用“prereset”挂钩（可能为NULL）。

在执行EH重置后，在EH重置之后调用“postreset”挂钩（可能为NULL）。根据现有条件、问题严重程度和硬件能力，

执行低级EH重置的“softreset”（可能为NULL）或“hardreset”（可能为NULL）将被调用。

void (\*post\_internal\_cmd) (struct ata\_queued\_cmd \*qc);

在使用ata\_exec\_internal()执行探测时间或EH时间命令后，执行任何必要的硬件特定操作。

### 硬件中断处理

irqreturn\_t (\*irq\_handler)(int, void \*, struct pt\_regs \*);

void (\*irq\_clear) (struct ata\_port \*);

->irq\_handler是由libata在系统中注册的中断处理例程。 ->irq\_clear在探测期间仅在注册中断处理程序之前调用，以确保硬件安静。

第二个**参数**dev\_instance应转换为指向struct ata\_host\_set的指针。

大多数遗留的IDE驱动程序使用ata\_sff\_interrupt()作为irq\_handler挂钩，该挂钩扫描主机集中的所有端口，确定哪个排队的命令处于活动状态（如果有），并调用ata\_sff\_host\_intr(ap，qc)。

大多数遗留的IDE驱动程序使用ata\_sff\_irq\_clear()作为irq\_clear()挂钩，该挂钩仅清除DMA状态寄存器中的中断和错误标志。

### SATA物理读/写

int (\*scr\_read) (struct ata\_port \*ap, unsigned int sc\_reg,u32 \*val);

int (\*scr\_write) (struct ata\_port \*ap, unsigned int sc\_reg,u32 val);

读写标准SATA phy寄存器。目前仅用于->phy\_reset钩子调用sata\_phy\_reset()助手函数时。sc\_reg是SCR\_STATUS、SCR\_CONTROL、SCR\_ERROR或SCR\_ACTIVE之一。

### 初始化和关闭

int (\*port\_start) (struct ata\_port \*ap);

void (\*port\_stop) (struct ata\_port \*ap);

void (\*host\_stop) (struct ata\_host\_set \*host\_set);

->port\_start()在初始化每个端口的数据结构后立即调用。通常用于分配每个端口的DMA缓冲区/表/环，启用DMA引擎和类似任务。一些驱动程序还使用这个入口点作为为ap- >private\_data分配驱动程序专用内存的机会。

许多驱动程序将ata\_port\_start()用作这个挂钩或从自己的port\_start()挂钩调用它。ata\_port\_start()为传统的IDE PRD表分配空间并返回。

->port\_stop()在->host\_stop()之后被调用。它的唯一功能是释放DMA/内存资源，因为它们不再被积极使用。许多驱动程序在此时还会释放端口的驱动程序私有数据。

->host\_stop()在所有->port\_stop()调用完成后调用。挂钩必须完成硬件关闭，释放DMA和其他资源等等。此钩子可能被指定为NULL，此时它不会被调用。

## 错误处理

本章说明了在libata下如何处理错误。读者建议先阅读SCSI EH（SCSI EH）和ATA异常文档。

### 命令的起源

在libata中，命令使用结构ata\_queued\_cmd或qc表示。qc在端口初始化期间预分配，并重复用于命令执行。目前每个端口仅分配一个qc，但尚未合并的NCQ分支为每个标记分配一个qc，并将每个qc映射到NCQ标记1到1。

libata命令可以从两个来源产生- libata本身和SCSI中间层。libata内部命令用于初始化和错误处理。所有用于SCSI模拟的普通blk请求和命令都通过SCSI主机模板的queuecommand回调作为SCSI命令传递。

### 如何发布命令

#### 内部命令

首先，使用 分配和初始化 qc [ata\_qc\_new\_init()](https://www.kernel.org/doc/html/v4.19/driver-api/libata.html" \l "c.ata_qc_new_init" \o "ata_qc_new_init)。虽然[ata\_qc\_new\_init()](https://www.kernel.org/doc/html/v4.19/driver-api/libata.html" \l "c.ata_qc_new_init" \o "ata_qc_new_init)在 qc 不可用时没有实现任何等待或重试机制，但内部命令目前仅在初始化和错误恢复期间发出，因此没有其他命令处于活动状态并且分配可以保证成功。

一旦分配了qc任务文件，就会为将要执行的命令初始化qc。qc目前有两种机制可以通知完成。一种是通过qc->complete\_fn()回调，另一种是完成qc->waiting。qc->complete\_fn()回调是由正常的SCSI翻译命令使用的异步路径，而qc->waiting是由内部命令使用的同步（发行者在进程上下文中睡眠）路径。

初始化完成后，host\_set锁被获取并发出qc。

#### SCSI命令

所有libata驱动程序都使用ata\_scsi\_queuecmd()作为hostt->queuecommand回调。scmds可以是模拟的或翻译的。在处理模拟scmd时不涉及任何qc。结果会立即计算并完成scmd。

qc->complete\_fn()回调用于完成通知。ATA命令使用ata\_scsi\_qc\_complete()，而ATAPI命令使用atapi\_qc\_complete()。这两个函数最终都会调用qc->scsidone，在qc完成时通知上层。翻译完成后，qc通过ata\_qc\_issue()发出。

请注意，SCSI中间层在持有host\_set锁时调用hostt->queuecommand，因此所有上述内容都在持有host\_set锁的同时发生。

### 如何处理命令

根据使用的协议和控制器的不同，处理命令的方式也不同。为了讨论的目的，假定使用taskfile接口和所有标准回调的控制器。

目前使用6种ATA命令协议。根据其处理方式，它们可以分为以下四个类别。

**ATA无数据或DMA**

ATA\_PROT\_NODATA和ATA\_PROT\_DMA属于此类别。这些类型的命令在发出后不需要任何软件干预。设备将在完成时引发中断。

**ATA PIO**

ATA\_PROT\_PIO属于这个类别。libata目前使用轮询实现PIO。ATA\_NIEN位设置为关闭中断，pio\_task on ata\_wq执行轮询和IO。

**ATAPI 无数据或DMA**

ATA\_PROT\_ATAPI\_NODATA和ATA\_PROT\_ATAPI\_DMA属于此类别。 packet\_task用于发出PACKET命令后轮询BSY位。一旦设备关闭BSY，packet\_task即传输CDB并将处理交给中断处理程序。

**ATAPI PIO**

ATA\_PROT\_ATAPI属于此类别。 ATA\_NIEN位被设置，与ATAPI NODATA或DMA一样，packet\_task提交cdb。然而，在提交cdb之后，进一步的处理（数据传输）交给pio\_task。

### 如何完成命令

一旦发出，所有qc都会通过ata\_qc\_complete()完成或超时。对于由中断处理的命令，ata\_host\_intr()会调用ata\_qc\_complete()，对于PIO任务，pio\_task会调用ata\_qc\_complete()。在错误情况下，packet\_task也可能完成命令。

ata\_qc\_complete()执行以下操作。

1）DMA内存解除映射。

2）ATA\_QCFLAG\_ACTIVE从qc->flags中清除。

3）调用qc->complete\_fn回调。如果回调的返回不为零。完成将被短路，ata\_qc\_complete()将返回。

4) 调用\_\_ata\_qc\_complete()，该函数执行以下操作：

qc->flags被设置为0。

ap->active\_tag和qc->tag被清空。

qc->waiting被清空并完成（按顺序）。

通过清除ap->qactive中适当的位来释放qc的内存。

因此，它基本上通知上层，并释放qc的内存。唯一的例外是#3中的短路路径，它由atapi\_qc\_complete()使用。

对于所有非ATAPI命令，无论成功或失败，几乎采用相同的代码路径并且很少进行错误处理。如果命令成功，那么将以成功状态完成qc；否则将以失败状态完成qc。

但是，失败的ATAPI命令需要更多的处理，因为需要使用REQUEST SENSE来获取感官数据。如果ATAPI命令失败，则会使用错误状态调用ata\_qc\_complete()，然后通过qc->complete\_fn()回调调用atapi\_qc\_complete()。

这使得atapi\_qc\_complete()将scmd->result设置为SAM\_STAT\_CHECK\_CONDITION，完成scmd并返回1。由于感官数据为空，但是scmd->result为CHECK CONDITION，因此SCSI中间层将为scmd调用EH，返回1将使ata\_qc\_complete()不释放qc的内存。这将导致我们进入具有部分完成的qc的ata\_scsi\_error()。

#### ata\_scsi\_error()

ata\_scsi\_error()是libata的当前transport-> eh\_strategy\_handler()。如上所述，它将在两种情况下进入-超时和ATAPI错误完成。该函数调用低级别的libata驱动程序的eng\_timeout()回调函数，其标准回调函数为ata\_eng\_timeout()。它检查qc是否激活，并在是的情况下在qc上调用ata\_qc\_timeout()。实际的错误处理发生在ata\_qc\_timeout()中。

如果超时时调用EH，ata\_qc\_timeout()会停止BMDMA并完成qc。请注意，由于我们当前正在EH中，因此无法调用scsi\_done。如SCSI EH文档中所述，恢复的scmd应通过scsi\_queue\_insert()或使用scsi\_finish\_command()完成。在这里，我们将qc->scsidone覆盖为scsi\_finish\_command()，然后调用ata\_qc\_complete()。

如果由于失败的ATAPI qc而调用EH，则此处将完成qc，但不将其内存释放。此半完成的目的是使用qc作为占位符，以使EH代码到达此位置。这有点半吊子的，但是有效。

一旦控制权到达此处，则通过显式调用\_\_ata\_qc\_complete()来释放qc的内存。然后发出REQUEST SENSE的内部qc。获取感官数据后，通过在scmd上直接调用scsi\_finish\_command()完成scmd。请注意，由于我们已经完成并释放了与scmd关联的qc，因此我们不需要/无法再次调用ata\_qc\_complete()。

### 当前EH存在的问题

错误表示太粗糙。目前，所有错误条件都用ATA STATUS和ERROR寄存器表示。不是ATA设备错误的错误都通过设置ATA\_ERR位将其视为ATA设备错误。需要更好的错误说明符，以正确表示ATA和其他错误/异常。

处理超时时，没有采取任何措施使设备忘记超时命令并准备就绪接收新命令。

通过ata\_scsi\_error()进行的EH处理无法正确地受到常规命令处理的保护。在EH进入时，设备不处于不动状态。超时的命令可能随时成功或失败。pio\_task和atapi\_task仍然可以运行。

错误恢复太弱。导致HSM不匹配错误和其他错误的设备/控制器通常需要重置以返回到已知状态。此外，高级错误处理是必要的，以支持诸如NCQ和热插拔之类的功能。

ATA错误直接在中断处理程序中处理，PIO错误在pio\_task中处理。从以下几个原因来说，这对于高级错误处理是有问题的：

首先，高级错误处理通常需要上下文和内部qc执行。

其次，即使是简单的故障（例如，CRC错误）也需要收集信息并可能触发复杂的错误处理（例如，重置和重新配置）。在多个代码路径中收集信息，进入EH并触发操作使生活变得痛苦。

第三，分散的EH代码使实现低级别驱动程序变得困难。低级别驱动程序会覆盖libata回调。如果EH分散在几个位置上，则每个受影响的回调都应执行其部分错误处理。这可能会出错并且很痛苦。

## libata库

### struct ata\_link \*ata\_link\_next(struct ata\_link \*link, struct ata\_port \*ap, enum ata\_link\_iter\_mode mode)

链接迭代辅助函数

**参数**

struct ata\_link \*link

上一个链接，NULL以开始

struct ata\_port \*ap

包含要迭代的链接的ATA端口

enum ata\_link\_iter\_mode mode

迭代模式，ATA\_LITER\_\*之一

**说明**

锁定：主机锁或EH上下文。

**返回**

下一个链接的指针。

### struct ata\_device \*ata\_dev\_next(struct ata\_device \*dev, struct ata\_link \*link, enum ata\_dev\_iter\_mode mode)

设备迭代辅助函数

**参数**

struct ata\_device \*dev

前一个设备，NULL以开始

struct ata\_link \*link

包含要迭代的设备的ATA链接

enum ata\_dev\_iter\_mode mode

迭代模式，ATA\_DITER\_\*之一

锁定：主机锁或EH上下文。

**返回**

下一个设备的指针。

### int atapi\_cmd\_type(u8 opcode)

从SCSI操作码确定ATAPI命令类型

**参数**

u8 opcode

SCSI操作码

从操作码确定ATAPI命令类型。

锁定：无。

**返回**

ATAPI\_{READ|WRITE|READ\_CD|PASS\_THRU|MISC}

### void ata\_tf\_to\_fis(const struct ata\_taskfile \* tf, u8 pmp, int is\_cmd, u8 \* fis)

将ATA任务文件转换为SATA FIS结构

**参数**

const struct ata\_taskfile \* tf

要转换的Taskfile

u8 pmp

端口倍增器端口

int is\_cmd

此FIS是指令

u8 \* fis

将数据输出到其中的缓冲区

**说明**

将标准ATA任务文件转换为Serial ATA FIS结构（Register-Host to Device）。

锁定：从调用者继承。

### void ata\_tf\_from\_fis(const u8 \* fis, struct ata\_taskfile \* tf)

将SATA FIS转换为ATA任务文件

**参数**

const u8 \* fis

将从中输入数据的缓冲区

struct ata\_taskfile \* tf

输出的Taskfile

**说明**

将Serial ATA FIS结构转换为标准ATA任务文件。

锁定：从调用者继承。

### unsigned int ata\_pack\_xfermask(unsigned int pio\_mask, unsigned int mwdma\_mask, unsigned int udma\_mask)

将pio、mwdma和udma掩码打包到xfer\_mask中

**参数**

unsigned int pio\_mask

pio\_mask

unsigned int mwdma\_mask

mwdma\_mask

unsigned int udma\_mask

udma\_mask

**说明**

将pio\_mask、mwdma\_mask和udma\_mask打包成一个unsigned int xfer\_mask。

锁定：无。

**返回**

已打包的xfer\_mask。

### void ata\_unpack\_xfermask(unsigned long xfer\_mask, unsigned long \* pio\_mask, unsigned long \* mwdma\_mask, unsigned long \* udma\_mask)

将xfer\_mask解包为pio、mwdma和udma掩码

**参数**

unsigned long xfer\_mask

要解包的xfer\_mask

unsigned long \* pio\_mask

结果pio\_mask

unsigned long \* mwdma\_mask

结果mwdma\_mask

unsigned long \* udma\_mask

结果udma\_mask

**说明**

将xfer\_mask解包为pio\_mask、mwdma\_mask和udma\_mask。任何空目标掩码都将被忽略。

### u8 ata\_xfer\_mask2mode(unsigned int xfer\_mask)

查找与给定xfer\_mask匹配的XFER\_\*

**参数**

unsigned int xfer\_mask

感兴趣的xfer\_mask

返回与xfer\_mask匹配的XFER\_\*值。只考虑xfer\_mask的最高位。

锁定：无。

**返回**

匹配的XFER\_\*值，如果没有找到，则为0xFF。

### **unsigned** int ata\_xfer\_mode2mask(u8 xfer\_mode)

查找与XFER\_\*匹配的xfer\_mask

**参数**

u8 xfer\_mode

感兴趣的XFER\_\*

返回与xfer\_mode匹配的xfer\_mask。

锁定：无。

**返回**

匹配的xfer\_mask，如果没有找到，则为0。

### int ata\_xfer\_mode2shift(u8 xfer\_mode)

查找与XFER\_\*匹配的xfer\_shift

**参数**

u8 xfer\_mode

感兴趣的XFER\_\*

返回与xfer\_mode匹配的xfer\_shift。

锁定：无。

**返回**

匹配的xfer\_shift，如果没有找到，则为-1。

### const char \*ata\_mode\_string(unsigned int xfer\_mask)

将xfer\_mask转换为字符串

**参数**

unsigned int xfer\_mask

支持的位掩码；只计算最高位。

确定表示最高速度（modemask中的最高位）的字符串。

锁定：无。

**返回**

表示mode\_mask中最高速度的常量C字符串，或常量C字符串“<n/a>”。

### unsigned int ata\_dev\_classify(const struct ata\_taskfile \* tf)

基于ATA规范签名确定设备类型

**参数**

const struct ata\_taskfile \* tf

要确定的设备的ATA任务文件寄存器集

根据ATA/PI规范（卷1，第5.14节）的“签名和持久性”部分从任务文件寄存器内容确定设备是否为ATA或ATAPI。

锁定：无。

**返回**

设备类型，ATA\_DEV\_ATA、ATA\_DEV\_ATAPI、ATA\_DEV\_PMP、ATA\_DEV\_ZAC或ATA\_DEV\_UNKNOWN失败的情况下。

### void ata\_id\_string(const u16 \* id, unsigned char \* s, unsigned int ofs, unsigned int len)

将IDENTIFY DEVICE页面转换为字符串

**参数**

const u16 \* id

要检查的IDENTIFY DEVICE结果

unsigned char \* s

输出数据的字符串

unsigned int ofs

进入识别设备页面

unsigned int len

要返回的字符串长度。必须是偶数。

在IDENTIFY DEVICE页面中，字符串被分解为16位块。运行整个字符串，并线性输出每个8位块，无论平台如何。

锁定：调用者。

### void ata\_id\_c\_string(const u16 \* id, unsigned char \* s, unsigned int ofs, unsigned int len)

将IDENTIFY DEVICE页面转换为C字符串

**参数**

const u16 \* id

要检查的IDENTIFY DEVICE结果

unsigned char \* s

输出数据的字符串

unsigned int ofs

进入识别设备页面

unsigned int len

要返回的字符串长度。必须是奇数。

此函数与ata\_id\_string相同，除了修剪尾随空格并以null终止结果字符串。len必须是实际的最大长度（偶数）+1。

锁定：调用者。

### unsigned int ata\_id\_xfermask(const u16 \* id)

从给定的IDENTIFY数据计算xfermask

**参数**

const u16 \* id

要从中计算xfer掩码的标识数据

计算此设备的xfermask。如果我们必须正确考虑早期设备，则这并不像看起来那么简单。

FIXME：预IDE驱动器定时（我们在乎吗？）。

锁定：无。

**返回**

计算的xfermask

### unsigned int ata\_pio\_need\_iordy(const struct ata\_device \* adev)

检查是否需要iordy

**参数**

const struct ata\_device \* adev

ATA设备

检查设备的当前速度是否需要IORDY。由各种控制器用于芯片配置。

### unsigned int ata\_do\_dev\_read\_id(struct ata\_device \* dev, struct ata\_taskfile \* tf, \_\_le16 \* id)

默认的ID读取方法

**参数**

struct ata\_device \*dev

设备

struct ata\_taskfile \*tf

建议的taskfile

\_\_le16 \*id

数据缓冲区

发出识别taskfile并返回包含识别数据的缓冲区。对于某些RAID控制器和早期的ATA设备，此函数被驱动程序封装或替换

### int ata\_cable\_40wire**(**struct ata\_port \*ap)

返回40根线缆类型

**参数**

struct ata\_port \*ap

端口

帮助驱动程序硬连接40根线缆检测。

### int ata\_cable\_80wire(struct ata\_port \*ap)

返回80根线缆类型

**参数**

struct ata\_port \*ap

端口

帮助驱动程序硬连接80根线缆检测。

### int ata\_cable\_unknown(struct ata\_port \*ap)

返回未知的PATA电缆。

**参数**

struct ata\_port \*ap

端口

帮助那些没有PATA电缆检测的驱动程序。

### int ata\_cable\_ignore(struct ata\_port \*ap)

返回忽略的PATA电缆。

**参数**

struct ata\_port \*ap

端口

帮助那些不使用电缆类型限制传输模式的驱动程序。

### int ata\_cable\_sata(struct ata\_port \*ap)

返回SATA电缆类型

**参数**

struct ata\_port \*ap

端口

帮助驱动程序具有SATA电缆的驱动程序

### struct ata\_device \*ata\_dev\_pair(struct ata\_device \*adev)

返回同一电缆上的其他设备

**参数**

struct ata\_device \*adev

设备

获取同一电缆上的其他设备，如果不存在，返回NULL

### int sata\_set\_spd(struct ata\_link \* link)

根据速度限制设置SATA速度

**参数**

struct ata\_link \* link

要设置SATA速度的链接

**说明**

根据sata\_spd\_limit设置链接的SATA spd。

锁定：从调用者继承。

**返回**

如果不需要更改spd，则为0，如果已更改spd，则为1。如果SCR寄存器无法访问，负errno。

### u8 ata\_timing\_cycle2mode(unsigned int xfer\_shift, int cycle)

查找指定周期持续时间的传输模式

**参数**

unsigned int xfer\_shift

要检查的传输类型的ATA\_SHIFT\_\*值。

int cycle

周期持续时间（以纳秒为单位）

**说明**

返回匹配的xfer模式以进行循环。返回的模式是由xfer\_shift指定的传输类型的模式。如果周期对于xfer\_shift太慢，则返回0xff。如果周期比已知的最快模式更快，则返回最快模式。

锁定：无。

**返回**

匹配的xfer\_mode，如果没有找到匹配，则为0xff。

### int ata\_do\_set\_mode(struct ata\_link \* link, struct ata\_device \*\* r\_failed\_dev)

编程定时并发出SET特征-XFER

**参数**

struct ata\_link \* link

将对其中编程的链接进行超时

struct ata\_device \*\* r\_failed\_dev

输出失败的设备的参数

**说明**

用于调整和设置ATA设备磁盘传输模式（PIO3，UDMA6等）的函数的标准实现。如果ata\_dev\_set\_mode（）失败，则在r\_failed\_dev中返回指向失败设备的指针。

锁定：PCI / etc总线探测信号。

**返回**

成功则返回0，否则为负errno

### int ata\_wait\_after\_reset(**struct** ata\_link \* link, unsigned long deadline, int (\*check\_ready) (struct ata\_link \*link)

重置后等待链接准备就绪

**参数**

struct ata\_link \* link

要等待的链接

unsigned long deadline

操作的截止时间

int (\*)(struct ata\_link \*link) check\_ready

回调以检查链接的准备就绪

**说明**

等待重置后的链接准备就绪。

锁定：EH上下文。

**返回**

如果链接在deadline之前就绪，则为0；否则为-errno。

### int sata\_link\_debounce(struct ata\_link \* link, const unsigned long \* params, unsigned long deadline)

除去SATA phy状态抖动

**参数**

struct ata\_link \* link

对SATA phy状态进行除抖动的ATA链接

const unsigned long \* params

时间参数{interval，duration，timeout }以毫秒为单位

unsigned long deadline

操作的截止时间

**说明**

确保link的SStatus达到稳定状态，该状态由保持不为1的DET的相同值确定，在duration内以interval轮询，然后在超时之前。超时约束稳定状态的开始。由于某些控制器在热拔插后DET卡在1上，因此此功能会等待超时，然后如果DET在1处稳定，则返回0。

timeout还受deadline限制。将使用两者中较早的那个。

锁定：内核线程上下文（可能会睡眠）

**返回**

成功则返回0，失败则为-errno。

### int sata\_link\_resume(struct ata\_link \* link, const unsigned long \* params, unsigned long deadline)

恢复SATA链接

**参数**

struct ata\_link \* link

要恢复SATA的ATA链接

const unsigned long \* params

时间参数{interval，duration，timeout }以毫秒为单位

unsigned long deadline

操作的截止时间

**说明**

恢复SATA phy链接并除去抖动。

锁定：内核线程上下文（可能会睡眠）

**返回**

成功则返回0，失败则为-errno。

### int sata\_link\_scr\_lpm(struct ata\_link \* link, enum ata\_lpm\_policy policy, bool spm\_wakeup)

操纵SControl IPM和SPM字段

**参数**

struct ata\_link \* link

要操纵SControl的ATA链接

枚举 ata\_lpm\_policy policy

要配置的LPM策略

bool spm\_wakeup

启动LPM过渡到活动状态

**说明**

根据策略来操纵link的SControl寄存器的IPM字段。如果政策是ATA\_LPM\_MAX\_POWER，并且spm\_wakeup为真，则操纵SPM字段以唤醒链接。此函数还在返回之前清除PHYRDY\_CHG。

锁定：EH上下文。

成功返回0，否则返回-errno。

### int ata\_std\_prereset（struct ata\_link \* link，unsigned long deadline）

准备重置

**参数**

struct ata\_link \* link

要重置的ATA链接

unsigned long deadline

操作的截止时间

**说明**

链接即将被重置。 初始化它。 来自prereset的故障将使libata中止整个重置序列并放弃该端口，因此prereset应该尽力而为。 它尽力为重置序列做准备，但如果事情出了问题，它应该只是哀叹，而不是失败。

锁定：内核线程上下文（可以休眠）

**返回**

成功返回0，否则返回-errno。

### int sata\_link\_hardreset（struct ata\_link \* link，const unsigned long \* timing，unsigned long deadline，bool \* online，int（\* check\_ready）（struct ata\_link \*）

通过SATA phy重置重置链接

**参数**

struct ata\_link \*link

需要重置的链接

const unsigned long \* timing

以毫秒为单位的时间参数{间隔，持续时间，超时}

unsigned long deadline

操作的截止时间

bool \*online

指示链接在线性的可选输出参数

int（\*）（struct ata\_link \*）check\_ready

用于检查链接准备就绪的可选回调

**说明**

使用SControl寄存器的DET位对SATA phy-reset链接进行重置。 硬复位后，如果指定了check\_ready，则会使用ata\_wait\_ready（）等待链接准备就绪。 LLD允许不指定check\_ready并在此函数返回后自行等待。 设备分类是LLD的责任。

如果重置成功并且链接在重置后处于在线状态，则将\* online设置为1。

锁定：内核线程上下文（可以休眠）

**返回**

成功返回0，否则返回-errno。

### int sata\_std\_hardreset（struct ata\_link \* link，unsigned int \* class，unsigned long deadline）

COMRESET而不等待或分类

**参数**

struct ata\_link \* link

要重置的链接

unsigned int \* class

附加设备的结果类

unsigned long deadline

操作的截止时间

标准SATA COMRESET而不等待或分类。

锁定：内核线程上下文（可以休眠）

**返回**

如果链接离线，则为0，如果链接在线，则为-EAGAIN，如果发生错误，则为-errno。

### void ata\_std\_postreset（struct ata\_link \* link，unsigned int \* classes）

标准postreset回调

**参数**

struct ata\_link \*链接

目标ata\_link

unsigned int \* classes

已附加设备的类

此函数在成功重置后调用。 请注意，可能已使用不同的重置方法多次重置设备，然后才调用postreset。

锁定：内核线程上下文（可以休眠）

### unsigned int ata\_dev\_set\_feature（struct ata\_device \* dev，u8 subcmd，u8 action）

发出SET FEATURES

**参数**

struct ata\_device \* dev

将发送命令的设备

u8 subcmd

要发送的SET FEATURES子命令

u8 action

扇区计数表示特定子命令的操作

将SET FEATURES命令发送到通过扇区计数连接到端口ap的设备dev

锁定：PCI /等总线探测信号。

**返回**

成功返回0，否则为AC\_ERR\_ \*掩码。

### int ata\_std\_qc\_defer（struct ata\_queued\_cmd \* qc）

检查qc是否需要延迟

**参数**

struct ata\_queued\_cmd \* qc

疑问中的ATA命令

非NCQ命令不能与任何其他命令，NCQ或非NCQ一起运行。 由于上层只知道队列深度，因此我们负责维护排除。 此函数检查是否可以发出新命令qc。

锁定：spin\_lock\_irqsave（host lock）

**返回**

如果需要推迟，则为ATA\_DEFER\_ \*，否则为0。

### void ata\_sg\_init（struct ata\_queued\_cmd \* qc，struct scatterlist \* sg，unsigned int n\_elem）

将命令与分散聚集表相关联。

**参数**

struct ata\_queued\_cmd \* qc

要关联的命令

struct scatterlist \* sg

分散聚集表。

unsigned int n\_elem

S / G表中的元素数。

**说明**

将queued\_cmd qc的数据相关元素初始化为指向分散聚集表sg的元素，其中包含n\_elem个元素。

锁定：spin\_lock\_irqsave（host lock）

### void ata\_qc\_complete（struct ata\_queued\_cmd \* qc）

完成一个活动的ATA命令

**参数**

struct ata\_queued\_cmd \* qc

要完成的命令

指示中间和上层已完成一个ATA命令，具有ok或not-ok状态。

成功完成多个NCQ命令时，请勿多次调用此函数。 必须使用ata\_qc\_complete\_multiple（），它将适当地更新IRQ期望状态。

锁定：spin\_lock\_irqsave（host lock）

### int ata\_qc\_complete\_multiple（struct ata\_port \* ap，u64 qc\_active）

完成多个成功的qc

**参数**

struct ata\_port \* ap

问题端口

u64 qc\_active

新的qc\_active掩码

**说明**

完成正在飞行的命令。 此函数旨在从低级驱动程序的中断例程中调用以正常完成请求。 按ap->qc\_active和qc\_active进行比较，并根据需要完成命令。

在IRQ处理程序中完成多个NCQ命令时，请始终使用此函数而不是多次调用ata\_qc\_complete（），以使IRQ期望状态保持同步。

锁定：spin\_lock\_irqsave（host lock）

**返回**

成功时完成的命令数，否则为-errno。

### int sata\_scr\_valid（struct ata\_link \* link）

测试SCR是否可访问

**参数**

struct ata\_link \*链接

要为测试SCR可访问性的ATA链接

**说明**

测试是否可以为链接访问SCR。

锁定：无。

**返回**

如果可以访问SCR，则为1，否则为0。

### int sata\_scr\_read（struct ata\_link \* link，int reg，u32 \* val）

阅读指定端口的SCR寄存器

**参数**

struct ata\_link \* link

要读取SCR的ATA链路

int reg

要读取的SCR

u32 \* val

存储读取值的位置

**说明**

将link的SCR寄存器reg读入\*val。如果link是ap->link，端口的电缆类型是SATA并且端口实现->scr\_read，则此函数保证成功。

锁：如果link是ap->link，则为无。否则是内核线程上下文。

**返回**

成功时为0，失败时为负的errno。

### int sata\_scr\_write(struct ata\_link \* link，int reg，u32 val)

写入指定端口的SCR寄存器

**参数**

struct ata\_link \* link

要为其写SCR的ATA链接

int reg

要写入的SCR

u32 val

要写入的值

**说明**

将val写入link的SCR寄存器reg。如果link是ap->link，端口的电缆类型是SATA并且端口实现->scr\_read，则此函数保证成功。

锁：如果link是ap->link，则为无。否则是内核线程上下文。

**返回**

成功时为0，失败时为负的errno。

### int sata\_scr\_write\_flush(struct ata\_link \* link，int reg，u32 val)

写入指定端口的SCR寄存器并刷新

**参数**

struct ata\_link \* link

要为其写SCR的ATA链接

int reg

要写入的SCR

u32 val

要写入的值

**说明**

此函数与sata\_scr\_write()相同，只是在写入寄存器后执行刷新。

锁：如果link是ap->link，则为无。否则是内核线程上下文。

**返回**

成功时为0，失败时为负的errno。

### bool ata\_link\_online(struct ata\_link \* link)

测试给定链接是否在线

**参数**

struct ata\_link \* link

要测试的ATA链接

测试链接是否在线。如果没有从属链接，则与ata\_phys\_link\_online()相同。如果有从属链接，则仅在主链接上调用此函数，并且如果M/S链接中有任何一个在线，则返回true。

锁：无。

**返回**

如果端口在线状态可用且在线，则为True。

### bool ata\_link\_offline(struct ata\_link \* link)

测试给定链接是否离线

**参数**

struct ata\_link \* link

要测试的ATA链接

测试链接是否离线。如果没有从属链接，则与ata\_phys\_link\_offline()相同。如果有从属链接，则仅在主链接上调用此函数，并且如果M/S链接都离线，则返回true。

锁：无。

**返回**

如果端口离线状态可用且离线，则为True。

### void ata\_host\_suspend(struct ata\_host \* host，pm\_message\_t mesg)

暂停主机

**参数**

struct ata\_host \* host

要暂停的主机

pm\_message\_t mesg

PM消息

暂停主机。端口暂停执行实际操作。

### void ata\_host\_resume(struct ata\_host \* host)

恢复主机

**参数**

struct ata\_host \* host

恢复的主机

恢复主机。端口恢复执行实际操作。

### struct ata\_host \*ata\_host\_alloc(struct device \* dev，int max\_ports)

分配和初始化基本ATA主机资源

**参数**

struct device \* dev

此主机与之关联的通用设备

int max\_ports

与此主机关联的ATA端口的最大数量

分配并初始化基本ATA主机资源。LLD调用此函数以分配主机，完全初始化它并使用ata\_host\_register()附加它。

分配max\_ports端口，并将host->n\_ports初始化为max\_ports。调用者可以在调用ata\_host\_register()之前减少host->n\_ports。未使用的端口将自动释放。

**返回**

成功时分配ATA主机，失败时为NULL。

锁：从调用层继承（可能休眠）。

### struct ata\_host \*ata\_host\_alloc\_pinfo(struct device \* dev，const struct ata\_port\_info \* const \* ppi，int n\_ports)

分配主机并使用port\_info数组进行初始化

**参数**

struct device \* dev

此主机与之关联的通用设备

const struct ata\_port\_info \* const \* ppi

要使用的ATA port\_info数组初始化主机

int n\_ports

附加到此主机的ATA端口数量

分配ATA主机并使用ppi中的信息进行初始化。如果以NULL结尾，则ppi可以包含少于n\_ports的条目。剩余端口将使用最后一个条目。

**返回**

成功时分配ATA主机，失败时为NULL。

锁：从调用层继承（可能休眠）。

### int ata\_slave\_link\_init(struct ata\_port \* ap)

初始化从属链接

**参数**

struct ata\_port \* ap

要为其初始化从属链接的端口

**说明**

创建并初始化ap的从属链接。这使能了在端口上处理从属链接。

在libata中，一个端口包含链接，一个链接包含设备。有单个主链接，但是如果将PMP连接到其上，则可以有多个扇出链接。在SATA上，通常将单个设备连接到链接上，但是模拟基于TF界面的PATA和SATA控制器可以有两个 - 主和从属。

但是，还有一些控制器不太适合这种抽象 - SATA控制器，它们模拟具有主设备和从属设备的TF接口，但也具有每个设备的单独SCR寄存器集。这些控制器需要单独的链接用于物理链接处理（例如在线，链接速度），但在其他方面（例如命令发出，软复位）应像传统的M / S控制器一样处理。

slave\_link是libata处理这类控制器的方法，不会对核心层产生太大影响。除了物理链路处理之外，对于其他任何事情，默认的主机链路都用于主从。对于物理链路处理，使用单独的ap->slave\_link。所有脏细节都在libata核心层中实现。从LLD的角度来看，唯一的区别是为从链路调用了prereset、hardreset和postreset，因此复位序列如下所示：

prereset(M) -> prereset(S) -> hardreset(M) -> hardreset(S) -> softreset(M) -> postreset(M) -> postreset(S)

请注意，软复位仅为主机调用。软复位根据定义可重置M/S，因此主机上的SRST应同时处理这两者（标准方法将完美地工作）。

锁定：在注册主机之前应调用。

**返回**

成功返回0，失败返回-errno。

### int ata\_host\_start(struct ata\_host \*host)

启动和冻结ATA主机的端口

**参数**

struct ata\_host \*host

要为其启动端口的ATA主机

启动然后冻结主机的端口。已启动状态记录在host->flags中，因此可以多次调用此函数。保证端口仅启动一次。如果host->ops尚未初始化，则将其设置为第一个非虚假端口操作。

锁定：从调用层继承（可能休眠）。

**返回**

如果成功启动所有端口，则返回0，否则返回-errno。

### void ata\_host\_init(struct ata\_host \*host, struct device \*dev, struct ata\_port\_operations \*ops)

为sas（ipr、libsas）初始化主机结构

**参数**

struct ata\_host \*host

要初始化的主机

struct device \*dev

主机附加到的设备

struct ata\_port\_operations \*ops

port\_ops

### int ata\_host\_register(struct ata\_host \*host, struct scsi\_host\_template \*sht)

注册初始化的ATA主机

**参数**

struct ata\_host \*host

要注册的ATA主机

struct scsi\_host\_template \*sht

用于SCSI主机的模板

注册初始化的ATA主机。主机使用ata\_host\_alloc()分配，并由LLD完全初始化。此函数启动端口，在ATA和SCSI层中注册主机并探测已注册的设备。

锁定：从调用层继承（可能休眠）。

**返回**

成功返回0，否则返回-errno。

### int ata\_host\_activate(struct ata\_host \*host, int irq, irq\_handler\_t irq\_handler, unsigned long irq\_flags, struct scsi\_host\_template \*sht)

启动主机，请求IRQ并注册它

**参数**

struct ata\_host \*host

目标ATA主机

int irq

要请求的IRQ

irq\_handler\_t irq\_handler

请求IRQ时使用的irq\_handler

unsigned long irq\_flags

请求IRQ时使用的irq\_flags

struct scsi\_host\_template \*sht

注册主机时要使用的scsi\_host\_template

**说明**

分配ATA主机并初始化后，大多数libata LLD执行三个步骤以激活主机-启动主机，请求IRQ并注册它。这个助手获取必要的参数并一次性执行三个步骤。

无效的IRQ跳过IRQ注册，并且希望主机在端口上设置轮询模式。在这种情况下，irq\_handler应为NULL。

锁定：从调用层继承（可能休眠）。

**返回**

成功返回0，否则返回-errno。

### void ata\_host\_detach(struct ata\_host \*host)

分离ATA主机的所有端口

**参数**

struct ata\_host \*host

要分离的主机

分离主机的所有端口。

锁定：内核线程上下文（可能休眠）。

### void ata\_pci\_remove\_one(struct pci\_dev \*pdev)

设备移除的PCI层回调

**参数**

struct pci\_dev \*pdev

已移除的PCI设备

PCI层通过此挂钩向libata指示已发生热拔插或模块卸载事件。分离所有端口。资源释放由devres处理。

锁定：从PCI层继承（可能休眠）。

### int ata\_platform\_remove\_one(struct platform\_device \*pdev)

设备移除的平台层回调

**参数**

struct platform\_device \*pdev

已移除的平台设备

**说明**

平台层通过此挂钩向libata指示已发生热拔插或模块卸载事件。分离所有端口。资源释放由devres处理。

锁定：从平台层继承（可能休眠）。

### void ata\_msleep(struct ata\_port \*ap, unsigned int msecs)

ATA EH所有者感知msleep

**参数**

struct ata\_port \*ap

要将睡眠时间归因于的ATA端口

unsigned int msecs

要睡眠的持续时间（以毫秒为单位）

**说明**

睡眠毫秒。如果当前任务是ap的EH的所有者，则在进入睡眠状态之前释放所有权，并在完整睡眠后重新获取所有权。换句话说，在此任务睡眠时，共享ap->host的其他端口将被允许拥有EH。

锁定：可能会休眠。

### u32 ata\_wait\_register(struct ata\_port \*ap, void \_\_iomem \*reg, u32 mask, u32 val, unsigned long interval, unsigned long timeout)

等待寄存器值更改

**参数**

struct ata\_port \*ap

要等待寄存器的ATA端口，可以为NULL

void \_\_iomem \*reg

IO映射的寄存器

u32 mask

要应用于读取寄存器值的掩码

u32 val

等待条件

unsigned long interval

以毫秒为单位的轮询间隔

unsigned long timeout

超时时间（以毫秒为单位）

**说明**

等待寄存器的一些位更改是ATA控制器的常见操作。此函数读取32位LE IO映射的寄存器reg，并测试以下条件。

（\*reg & mask）! = val

如果满足条件，则返回；否则，在interval\_msec之后重复该过程，直到超时。

锁定：内核线程上下文（可能会睡觉）

**返回**

最终的寄存器值。

### bool sata\_lpm\_ignore\_phy\_events（struct ata\_link \* link）

测试是否应忽略物理事件

**参数**

struct ata\_link \*链接

接收事件的链接

**说明**

测试是否需要忽略接收到的物理事件。

锁住：无：

**返回**

如果必须忽略事件，则为True。

## libata核心内部

### struct ata\_link \* ata\_dev\_phys\_link（struct ata\_device \* dev）

查找设备的物理链接

**参数**

struct ata\_device \* dev

要查找物理链接的ATA设备

**说明**

查找连接到dev的物理链接。请注意，当dev在从机链接上时，这与dev->link不同。对于所有其他情况，它与dev->link相同。

锁定：不介意。

**返回**

指向找到的物理链接的指针。

### void ata\_force\_cbl(struct ata\_port \* ap)

按照libata.force强制电缆类型

**参数**

struct ata\_port \* ap

感兴趣的ATA端口

根据libata.force强制电缆类型并对此进行抱怨。使用具有匹配端口号的最后一个条目，因此它可以作为设备强制参数的一部分指定。例如，“a：40c，1.00：udma4”和“1.00：40c，udma4”的效果相同。

锁住：EH上下文。

### void ata\_force\_link\_limits(struct ata\_link \* link)

按照libata.force强制链接限制

**参数**

struct ata\_link \*链接

感兴趣的ATA链接

根据libata.force强制链接标志和SATA spd限制并对其进行抱怨。当仅指定端口部分（例如1:）时，限制适用于连接到主链接和通过PMP连接的所有扇出端口的所有链接。如果将设备部分指定为0（例如1.00：），则指定第一个扇出链接而不是主链接。设备号15始终指向主链接，无论是否连接了PMP。如果控制器具有从机链接，则设备号16指向它。

锁定：EH上下文。

### void ata\_force\_xfermask(struct ata\_device \* dev)

根据libata.force强制xfermask

**参数**

struct ata\_device \* dev

感兴趣的ATA设备

根据libata.force强制xfer\_mask并对其进行抱怨。为了保持与链接选择的一致性，设备号15选择连接到主链接的第一个设备。

锁住：EH上下文。

### void ata\_force\_horkage(struct ata\_device \* dev)

根据libata.force强制horkage

**参数**

struct ata\_device \* dev

感兴趣的ATA设备

根据libata.force强制horkage并对其进行抱怨。为了保持与链接选择的一致性，设备号15选择连接到主链接的第一个设备。

锁住：EH上下文。

### int ata\_rwcmd\_protocol(struct ata\_taskfile \* tf，struct ata\_device \* dev)

设置任务文件的r / w命令和协议

**参数**

struct ata\_taskfile \* tf

要检查和配置的命令

struct ata\_device \* dev

tf所属的设备

检查设备配置和tf->flags以计算使用的适当读/写命令和协议。

锁定：呼叫者。

### u64 ata\_tf\_read\_block（const struct ata\_taskfile \* tf，struct ata\_device \* dev）

从ATA任务文件中读取块地址

**参数**

const struct ata\_taskfile \* tf

感兴趣的ATA任务文件

struct ata\_device \* dev

ATA设备tf所属的

锁住：无。

从tf中读取块地址。此函数可处理所有三种地址格式-LBA，LBA48和CHS。tf->protocol和标志选择要使用的地址格式。

**返回**

从TF中读取的块地址。

### int ata\_build\_rw\_tf（struct ata\_queued\_cmd \* qc，u64 block，u32 n\_block，unsigned int tf\_flags，int class）

为给定的读/写请求构建ATA任务文件

**参数**

struct ata\_queued\_cmd \* qc

与要构建任务文件的元数据

u64 block

块地址

u32 n\_block

块数

unsigned int tf\_flags

RW / FUA等…

int class

IO优先级类

锁住：无。

为命令qc构建ATA任务文件，用于由块，n\_block，tf\_flags和class说明的读/写请求。

**返回**

成功时为0，如果请求对于dev太大则为-ERANGE，如果请求无效则为-EINVAL。

### int ata\_read\_native\_max\_address（struct ata\_device \* dev，u64 \* max\_sectors）

读取本地最大地址

**参数**

struct ata\_device \* dev

目标设备

u64 \* max\_sectors

结果本机最大地址的输出参数

对所讨论的设备执行LBA48或LBA28本机大小查询。

返回

成功时为0，如果命令被驱动器中止，则为-EACCES。发生其他错误时为-EIO。

### int ata\_set\_max\_sectors（struct ata\_device \* dev，u64 new\_sectors）

设置最大扇区数

**参数**

struct ata\_device \* dev

目标设备

u64 new\_sectors

要为设备设置的新最大扇区值

将dev的最大扇区数设置为new\_sectors。

**返回**

成功时为0，如果命令被驱动器中止或拒绝（由于先前的非易失性SET\_MAX）则为-EACCES。发生其他错误时为-EIO。

### int ata\_hpa\_resize（struct ata\_device \* dev）

调整具有HPA设置的设备的大小

**参数**

struct ata\_device \*dev

需要调整大小的设备

读取带有HPA特征的LBA28或LBA48磁盘的大小，如果需要则将其调整为介质的完整大小。调用者必须检查驱动器已启用HPA功能集。

**返回**

成功返回0，失败返回-errno。

### void ata\_dump\_id(struct ata\_device \*dev，const u16 \*id)

识别设备信息的调试输出

**参数**

struct ata\_device \*dev

获取信息的设备

const u16 \*id

要转储的IDENTIFY DEVICE页面

从给定的IDENTIFY DEVICE页面中转储所选的16位字。

锁定：调用者。

### unsigned ata\_exec\_internal\_sg(struct ata\_device \*dev, struct ata\_taskfile \*tf, const u8 \*cdb，int dma\_dir，struct scatterlist \*sgl，unsigned int n\_elem，unsigned int timeout)

执行libata内部命令

**参数**

struct ata\_device \*dev

发送命令的设备

struct ata\_taskfile \*tf

用于命令和结果的Taskfile寄存器

const u8 \* cdb

用于分组命令的CDB

int dma\_dir

命令的数据传输方向

struct scatterlist \* sg1

用于命令数据缓冲区的sg列表

unsigned int n\_elem

sg条目数

unsigned int timeout

以毫秒为单位的超时(默认为0)

使用超时执行libata内部命令。tf在入口处包含命令，在返回时包含结果。超时和错误条件通过返回报告。命令超时后不会采取恢复动作。这是调用者在超时后清理的职责。

锁定：无。应该在内核上下文中调用，可能会休眠。

**返回**

成功返回零，失败返回AC\_ERR\_\*掩码。

### unsigned ata\_exec\_internal(struct ata\_device \*dev，struct ata\_taskfile \*tf，const u8 \*cdb，int dma\_dir，void \*buf，unsigned int buflen，unsigned int timeout)

执行libata内部命令

**参数**

struct ata\_device \*dev

发送命令的设备

struct ata\_taskfile \*tf

用于命令和结果的Taskfile寄存器

const u8 \* cdb

用于分组命令的CDB

int dma\_dir

命令的数据传输方向

void \* buf

命令的数据缓冲区

unsigned int buflen

数据缓冲区的长度

unsigned int timeout

以毫秒为单位的超时(默认为0)

ata\_exec\_internal\_sg()的包装器，它使用简单的缓冲区而不是sg列表。

锁定：无。应该在内核上下文中调用，可能会休眠。

**返回**

成功返回零，失败返回AC\_ERR\_\*掩码。

### u32 ata\_pio\_mask\_no\_iordy(const struct ata\_device \*adev)

返回非IORDY掩码

**参数**

const struct ata\_device \*adev

ATA设备

如果我们没有使用iordy，计算最高可能的模式。如果没有iordy模式可用，则返回-1。

### int ata\_dev\_read\_id(struct ata\_device \*dev, unsigned int \* p\_class，unsigned int flags，u16 \* id)

从指定的设备读取ID数据

**参数**

struct ata\_device \*dev

目标设备

unsigned int \* p\_class

指向目标设备类的指针(可能会更改)

unsigned int flags

ATA\_READID\_\*标志

u16 \* id

读取IDENTIFY数据的缓冲区

从指定的设备读取ID数据。在ATA设备上执行ATA\_CMD\_ID\_ATA，在ATAPI设备上执行ATA\_CMD\_ID\_ATAPI。此函数还为早期驱动器发出ATA\_CMD\_INIT\_DEV\_PARAMS。

FIXME：ATA\_CMD\_ID\_ATA对于早期的驱动器是可选的，如果我们遇到这种情况，现在我们会中止。

锁定：内核线程上下文(可能会休眠)

**返回**

成功返回0，否则返回-errno。

### unsigned int ata\_read\_log\_page(struct ata\_device \*dev，u8 log，u8 page，void \* buf，unsigned int sectors)

读取特定的日志页面

**参数**

struct ata\_device \*dev

目标设备

u8 log

要读取的日志

u8 page

要读取的页面

void \* buf

缓冲区以存储已读取的页面

unsigned int sectors

要读取的扇区数

使用READ\_LOG\_EXT命令读取日志页面。

锁定：内核线程上下文(可能会休眠)。

**返回**

成功返回0，否则返回AC\_ERR\_\*掩码。

### int ata\_dev\_configure(struct ata\_device \*dev)

配置指定的ATA/ATPI设备

**参数**

struct ata\_device \*dev

要配置的目标设备

根据dev->id配置dev。还应用通用和低级驱动程序特定的修复程序。

锁定：内核线程上下文(可能会休眠)

**返回**

成功返回0，否则返回-errno

### int ata\_bus\_probe(struct ata\_port \* ap)

重置和探测ATA总线

**参数**

struct ata\_port \*ap

要探测的总线

主ATA总线探测函数。启动硬件相关的总线复位，然后尝试识别在总线上找到的任何设备。

锁定：PCI等总线探测信号量。

**返回**

成功返回零，否则返回负的errno。

### void sata\_print\_link\_status(struct ata\_link \*link)

打印SATA链接状态

**参数**

struct ata\_link \*link

要打印链接状态的SATA链接

此函数打印SATA链接的链接速度和状态。

锁定：无。

### int sata\_down\_spd\_limit(struct ata\_link \*link，u32 spd\_limit)

调整SATA spd下限

**参数**

struct ata\_link \*link

要调整SATA spd下限的链接

u32 spd\_limit

附加限制

将链接的SATA spd限制向下调整。请注意，此函数仅调整限制。必须使用sata\_set\_spd()应用更改。

锁定：继承自调用程序。

成功返回0，失败返回负errno

### int sata\_set\_spd\_needed(struct ata\_link \*link)

SATA SPD配置是否需要

**参数**

struct ata\_link \*link

相关链接

**说明**

测试SControl中的spd限制是否与link->sata\_spd\_limit匹配。此函数用于确定是否需要应用SATA spd配置进行硬复位。

锁定：继承自调用程序。

**返回**

如果需要SATA spd配置，则返回1，否则返回0。

### int ata\_down\_xfermask\_limit(struct ata\_device \*dev，unsigned int sel)

降低设备xfer掩码

**参数**

struct ata\_device \*dev

要调整xfer掩码的设备

unsigned int sel

ATA\_DNXFER\_\*选择器

降低dev的xfer掩码。请注意，此函数不应用更改。随后调用ata\_set\_mode()将应用限制。

锁定：继承自调用程序。

**返回**

成功返回0，失败返回负errno

### int ata\_wait\_ready(struct ata\_link \*link，unsigned long deadline，int (\*check\_ready)(struct ata\_link \*link))

等待链接准备就绪

**参数**

struct ata\_link \*link

要等待的链接

unsigned long deadline

操作的截止jiffies

int (\*check\_ready)(struct ata\_link \*link)

检查链接准备的回调

等待链接准备就绪。如果链接准备就绪，check\_ready应返回正数；如果不是，则返回0;-ENODEV如果链接似乎没有被占用，则为其他errno为其他错误条件。

允许跨ATA\_TMOUT\_FF\_WAIT的暂时-ENODEV条件。

锁定：EH环境。

**返回**

如果链接在截止日期之前准备好，则返回0；否则，返回-errno。

### int ata\_dev\_same\_device(struct ata\_device \*dev，unsigned int new\_class，const u16 \*new\_id)

确定新ID是否与配置的设备匹配

**参数**

struct ata\_device \*dev

要比较的设备

unsigned int new\_class

新设备的类别

const u16 \*new\_id

新设备的IDENTIFY页面

将new\_class和new\_id与dev进行比较，并确定dev是否是由new\_class和new\_id指示的设备。

锁定：无。

**返回**

如果dev与new\_class和new\_id匹配，则返回1，否则返回0。

### int ata\_dev\_reread\_id(struct ata\_device \*dev，unsigned int readid\_flags)

重新读取标识数据

**参数**

struct ata\_device \*dev

目标ATA设备

unsigned int readid\_flags

读取ID标志

重新读取识别页面，并确保dev仍连接到端口。

锁定：内核线程上下文（可能会睡眠）

**返回**

成功返回0，否则返回负errno

### int ata\_dev\_revalidate(struct ata\_device \*dev，unsigned int new\_class，unsigned int readid\_flags)

重新验证ATA设备

**参数**

struct ata\_device \*dev

要重新验证的设备

unsigned int new\_class

新类别代码

unsigned int readid\_flags

读取ID标志

重新读取识别页面，确保dev仍连接到端口，并根据新的IDENTIFY页面重新配置它。

锁定：内核线程上下文（可能会睡眠）

**返回**

成功返回0，否则返回负errno

### int ata\_is\_40wire(struct ata\_device \*dev)

检查驱动器端的检测

**参数**

struct ata\_device \*dev

设备

执行驱动器侧检测解码，允许设备供应商无法遵循文档。

### int cable\_is\_40wire(struct ata\_port \*ap)

40/80/SATA决策者

**参数**

struct ata\_port \*ap

要考虑的端口

这个函数将速度管理策略封装在一个地方。目前我们不缓存结果，但有一个好的案例可以在我们用未知电缆调用时将ap->cbl设置为结果（并找出它是否会影响热插拔）。

如果电缆似乎为40电缆，则返回1。

### void ata\_dev\_xfermask(struct ata\_device \*dev)

计算给定设备的支持xfer掩码

**参数**

struct ata\_device \*dev

计算xfer掩码的设备

计算dev的支持xfer掩码，并将其存储在dev->\*\_mask中。此函数负责应用所有已知限制，包括主机控制器限制，设备黑名单等...

锁定：无。

### unsigned int ata\_dev\_set\_xfermode(struct ata\_device \*dev)

发出SET FEATURES-XFER MODE命令

**参数**

struct ata\_device \*dev

将要发送命令的设备

向端口ap的设备dev发出设置特性-XFER MODE命令。

锁定：PCI/等总线探测信号。

**返回**

成功返回0，否则返回AC\_ERR\_\*掩码。

### unsigned int ata\_dev\_init\_params(struct ata\_device \*dev，u16 heads，u16 sectors)

发出初始化DEV PARAMS命令

**参数**

struct ata\_device \*dev

将要发送命令的设备

u16 heads

磁头数（任务文件参数）

u16 sectors

扇区数（任务文件参数）

锁定：内核线程上下文（可能会睡眠）

**返回**

成功返回0，否则返回AC\_ERR\_\*掩码。

### int atapi\_check\_dma(struct ata\_queued\_cmd \*qc)

检查是否支持ATAPI DMA

**参数**

struct ata\_queued\_cmd \*qc

用于检查任务文件的元数据

允许低级驱动程序过滤ATA PACKET命令，返回一个指示能否为所提供的PACKET命令使用DMA的状态。

锁定：spin\_lock\_irqsave（主机锁定）

**返回**

当可以使用ATAPI DMA时返回0

否则为非零

LOCKING: spin\_lock\_irqsave（主机锁）

### int ata\_sg\_setup（struct ata\_queued\_cmd \* qc）

DMA映射与命令相关的散射-聚集表。

**参数**

struct ata\_queued\_cmd \* qc

具有散射-聚集表要映射的命令。

DMA映射与queued\_cmd qc相关联的散射-聚集表。

LOCKING：spin\_lock\_irqsave（主机锁）

**返回**

成功为零，错误为负。

### void swap\_buf\_le16（u16 \* buf，unsigned int buf\_words）

地方交换16位字的一半

**参数**

u16 \* buf

要交换的缓冲区

unsigned int buf\_words

缓冲区中的16位字数。

如果需要将16位字的后一半交换以将字节顺序从小端转换为本机CPU字节顺序或反之亦然，则交换16位字的一半。

锁定：从调用方继承。

### struct ata\_queued\_cmd \* ata\_qc\_new\_init（struct ata\_device \* dev，int tag）

请求可用的ATA命令并初始化它

**参数**

struct ata\_device \* dev

我们从中请求可用命令结构的设备

int tag

标签

**说明**

LOCKING：无。

### void ata\_qc\_free（struct ata\_queued\_cmd \* qc）

释放未使用的ata\_queued\_cmd

**参数**

struct ata\_queued\_cmd \* qc

要完成的命令

设计在某些情况下阻止使用它时释放未使用的ata\_queued\_cmd对象。

锁定：spin\_lock\_irqsave（主机锁）

### void ata\_qc\_issue（struct ata\_queued\_cmd \* qc）

发出任务文件到设备

**参数**

struct ata\_queued\_cmd \* qc

要发出到设备的命令

准备ATA命令以提交到设备。这包括将数据映射到可进行DMA的区域，填充S / G表，最后将任务文件写入硬件，启动命令。

锁定：spin\_lock\_irqsave（主机锁）

bool ata\_phys\_link\_online（struct ata\_link \* link）

测试给定的链接是否在线

**参数**

struct ata\_link \* link

要测试的ATA链接

测试链接是否在线。请注意，如果无法获取链接的在线状态，则此函数返回0，因此ata\_link\_online（link）！=！ata\_link\_offline（link）。

锁定：无。

**返回**

如果端口在线状态可用且在线，则为True。

### bool ata\_phys\_link\_offline（struct ata\_link \* link）

测试给定的链接是否离线

**参数**

struct ata\_link \* link

要测试的ATA链接

测试链接是否离线。请注意，如果无法获取链接的离线状态，则此函数返回0，因此ata\_link\_online（link）！=！ata\_link\_offline（link）。

锁定：无。

**返回**

如果端口离线状态可用并且离线，则为真。

### void ata\_dev\_init（struct ata\_device \* dev）

初始化ata\_device结构

**参数**

struct ata\_device \* dev

要初始化的设备结构

在探测之前初始化dev。

锁定：从调用方继承。

### void ata\_link\_init（struct ata\_port \* ap，struct ata\_link \* link，int pmp）

初始化ata\_link结构

**参数**

struct ata\_port \* ap

ATA端口连接的端口链接

struct ata\_link \* link

要初始化的链接结构

int pmp

端口乘法器端口号

初始化链接。

锁定：内核线程上下文（可能会睡眠）

### int sata\_link\_init\_spd（struct ata\_link \* link）

初始化link- > sata\_spd\_limit

**参数**

struct ata\_link \* link

要为其配置sata\_spd\_limit的链接

将link- > [hw\_ ]sata\_spd\_limit初始化为当前配置的值。

锁定：内核线程上下文（可能会睡眠）。

**返回**

成功则为0，失败则为-errno。

### struct ata\_port \* ata\_port\_alloc（struct ata\_host \* host）

分配并初始化基本ATA端口资源

**参数**

struct ata\_host \* host

此分配端口所属的ATA主机

分配和初始化基本ATA端口资源。

**返回**

成功则分配ATA端口，否则为NULL。

锁定：从调用层继承（可能会睡眠）。

### void ata\_finalize\_port\_ops（struct ata\_port\_operations \* ops）

最终ata\_port\_operations

**参数**

struct ata\_port\_operations \* ops

要完成的ata\_port\_operations

ata\_port\_operations可以继承自另一个ops，该ops可以再次继承自另一个ops。只要继承链中没有循环，就可以一直进行下去。

当主机启动时，Ops表会被完成。 NULL或未指定的条目从最接近的祖先继承该方法，该条目用它进行填充。完成后，ops表直接指向所有方法，而-> inherits不再需要并清除。

使用ATA\_OP\_NULL，继承ops可以将方法强制为NULL。

锁定：无。

### void ata\_port\_detach（struct ata\_port \* ap）

准备拆卸设备以分离ATA端口

**参数**

struct ata\_port \* ap

要分离的ATA端口

拆卸ap的所有ATA设备和相关的SCSI设备;然后，删除关联的SCSI主机。保证ap在此功能返回时是静止的。

锁定：内核线程上下文（可能会睡眠）。

### void \_\_ata\_ehi\_push\_desc（struct ata\_eh\_info \* ehi，const char \* fmt，...）

在不添加分隔符的情况下推送错误说明

**参数**

struct ata\_eh\_info \* ehi

目标EHI

const char \* fmt

printf格式字符串

...

变量参数

根据fmt格式化字符串并将其附加到ehi- > desc。

锁定：spin\_lock\_irqsave（主机锁）

### void ata\_ehi\_push\_desc（struct ata\_eh\_info \* ehi，const char \* fmt，...）

推送带分隔符的错误说明

**参数**

struct ata\_eh\_info \* ehi

目标EHI

const char \* fmt

printf格式字符串

...

变量参数

根据fmt格式化字符串并将其附加到ehi- > desc。如果ehi- > desc不为空，则在中间添加“，”。

锁定：spin\_lock\_irqsave（主机锁）

### void ata\_ehi\_clear\_desc（struct ata\_eh\_info \* ehi）

清除错误说明

**参数**

struct ata\_eh\_info \*ehi

目标EHI

清除ehi->desc

锁定: spin\_lock\_irqsave（host lock）

### void ata\_port\_desc（struct ata\_port \*ap，const char \*fmt，...）

附加端口说明

**参数**

struct ata\_port \*ap

目标ATA端口

const char \*fmt

printf格式字符串

...

变量参数

根据fmt格式字符串，附加端口说明。如果端口说明不为空，则在其中添加“ ”。此函数用于初始化ata\_host。该说明将打印在主机注册上。

### void ata\_port\_pbar\_desc（struct ata\_port \*ap，int bar，ssize\_t offset，const char \*name）

附加PCI BAR说明

**参数**

struct ata\_port \*ap

目标ATA端口

int bar

目标PCI BAR

ssize\_t offset

偏移到PCI BAR中

const char \*name

该区域的名称

如果偏移为负，则该函数格式化一个包含该BAR的名称、地址、大小和类型的字符串，并将其附加到端口说明中。如果偏移为零或正，则只附加名称和偏移地址。

锁定：没有

### unsigned int ata\_internal\_cmd\_timeout（struct ata\_device \*dev，u8 cmd）

确定内部命令的超时时间

**参数**

struct ata\_device \*dev

目标设备

u8 cmd

要发出的内部命令

确定设备的命令cmd的超时时间。

锁定：EH上下文。

**返回**

确定的超时时间。

### void ata\_internal\_cmd\_timed\_out（struct ata\_device \*dev，u8 cmd）

内部命令超时通知

**参数**

struct ata\_device \*dev

目标设备

u8 cmd

超时的内部命令

通知EH，该设备的命令cmd已超时。此函数仅应针对使用ata\_internal\_cmd\_timeout（）确定超时的命令调用。

锁定：EH上下文。

### void ata\_eh\_acquire（struct ata\_port \*ap）

获得EH所有权

**参数**

struct ata\_port \*ap

为其获取EH所有权的ATA端口

为ap获取EH所有权。这是端口共享主机的基本互斥机制。只有一个端口从同一主机下挂掉，可以声明EH的所有权。

锁定：EH context。

### void ata\_eh\_release（struct ata\_port \*ap）

释放EH所有权

**参数**

struct ata\_port \*ap

为其释放EH所有权的ATA端口

如果调用者，则释放ap的EH所有权。调用者必须预先使用ata\_eh\_acquire（）获取EH所有权。

锁定：EH context。

### void ata\_scsi\_error（struct Scsi\_Host \*host）

SCSI层错误处理程序回调

**参数**

struct Scsi\_Host \*host

发生错误的SCSI主机

处理SCSI层引发的错误事件。

锁定：从SCSI层继承（没有，可以睡眠）

**返回**

零。

### void ata\_scsi\_cmd\_error\_handler（struct Scsi\_Host \*host，struct ata\_port \*ap，struct list\_head \*eh\_work\_q）

一系列命令的错误回调

**参数**

struct Scsi\_Host \*host

包含端口的SCSI主机

struct ata\_port \*ap

主机内的ATA端口

struct list\_head \*eh\_work\_q

要处理的命令列表

**说明**

处理给定的一系列命令，并将已完成的命令返回到ap->eh\_done\_q。此函数是libata错误处理程序的第一部分，用于处理给定的失败命令列表。

### void ata\_scsi\_port\_error\_handler（struct Scsi\_Host \*host，struct ata\_port \*ap）

在命令后恢复端口

**参数**

struct Scsi\_Host \*host

包含端口的SCSI主机

struct ata\_port \*ap

ATA端口

**说明**

在所有命令恢复后，处理端口ap的恢复。

### void ata\_port\_wait\_eh（struct ata\_port \*ap）

等待当前挂起的EH完成

**参数**

struct ata\_port \*ap

等待EH的端口

等待当前挂起的EH完成。

锁定：内核线程上下文（可能睡眠）。

### void ata\_eh\_set\_pending（struct ata\_port \*ap，int fastdrain）

设置ATA\_PFLAG\_EH\_PENDING并激活快速排水

**参数**

struct ata\_port \*ap

目标ATA端口

int fastdrain

激活快速排水

如果fastdrain为非零且之前未挂起EH，则设置ATA\_PFLAG\_EH\_PENDING并激活快速排水。快速排水确保EH及时启动。

锁定：spin\_lock\_irqsave（host lock）

### void ata\_qc\_schedule\_eh（struct ata\_queued\_cmd \*qc）

为错误处理程序安排qc

**参数**

struct ata\_queued\_cmd \*qc

计划进行错误处理的命令

为qc安排错误处理。其他命令排干后，EH将开始工作。

锁定：spin\_lock\_irqsave（host lock）

### void ata\_std\_sched\_eh（struct ata\_port \*ap）

非Libsas ata\_ports使用此公共例程进行eh

**参数**

struct ata\_port \*ap

为其安排EH的ATA端口

锁定：从ata\_port\_schedule\_eh spin\_lock\_irqsave（host lock）继承

### void ata\_std\_end\_eh（struct ata\_port \*ap）

非Libsas ata\_ports使用此公共例程完成eh

**参数**

struct ata\_port \*ap

为其结束EH的ATA端口

**说明**

在libata对象模型中，ata\_port与shost具有1：1映射，因此可以在ap->lock下直接操作主机字段，在libsas情况下，我们需要在ha->level上保持锁定状态，以协调这些事件。

锁定：spin\_lock\_irqsave（host lock）

### void ata\_port\_schedule\_eh（struct ata\_port \*ap）

安排错误处理程序而不使用qc

**参数**

struct ata\_port \*ap

为其安排EH的ATA端口

为ap安排错误处理程序。当所有命令排干时，EH将开始工作。

锁定：spin\_lock\_irqsave（host lock）

### int ata\_link\_abort（struct ata\_link \*link）

放弃链接上的所有qc

**参数**

struct ata\_link \*link

要中止qc的ATA链接

放弃链路上所有活动的qc并计划EH。

锁定：spin\_lock\_irqsave（host lock）

**返回**

中止的qc数。

### int ata\_port\_abort（struct ata\_port \*ap）

放弃端口上的所有qc

**参数**

struct ata\_port \* ap

要中止qc的ATA端口

放弃ap的所有活动qc并计划EH。

锁定：spin\_lock\_irqsave（host\_set lock）

**返回**

中止的qc数。

### void \_\_ata\_port\_freeze（struct ata\_port \* ap）

冻结端口

**参数**

struct ata\_port \* ap

要冻结的ATA端口

当HSM违规或其他条件破坏端口正常操作时调用此函数。冻结的端口不允许执行任何操作，直到恢复端口为止，通常会进行成功复位。

ap->ops->freeze（）回调可用于冻结端口硬件（例如屏蔽中断和停止DMA引擎）。如果端口无法硬件冻结，则中断处理程序必须在端口冻结时无条件确认并清除中断。

锁定：spin\_lock\_irqsave（host lock）

### int ata\_port\_freeze（struct ata\_port \* ap）

放弃和冻结端口

**参数**

struct ata\_port \* ap

要冻结的ATA端口

放弃并冻结ap。必须先调用冻结操作，因为某些硬件要求在可以访问任务文件寄存器之前执行特殊操作。

锁定：spin\_lock\_irqsave（host lock）

**返回**

中止命令的数量。

### int sata\_async\_notification（struct ata\_port \* ap）

SATA异步通知处理程序

**参数**

struct ata\_port \* ap

接收异步通知的ATA端口

**说明**

当通过SDB FIS接收异步通知时调用处理程序。如果需要，此函数将计划EH。

锁定：spin\_lock\_irqsave（host lock）

**返回**

如果计划了EH，则为1，否则为0。

### void ata\_eh\_freeze\_port（struct ata\_port \* ap）

EH助手冻结端口

**参数**

struct ata\_port \* ap

要冻结的ATA端口

冻结ap。

锁定：无。

### void ata\_eh\_thaw\_port（struct ata\_port \* ap）

解冻端口的EH助手

**参数**

struct ata\_port \* ap

要解冻的ATA端口

解冻被冻结的端口ap。

锁定：无。

### void ata\_eh\_qc\_complete（struct ata\_queued\_cmd \* qc）

完成来自EH的活动ATA命令

**参数**

struct ata\_queued\_cmd \* qc

要完成的命令

向中间和上层指示ATA命令已完成。应从EH中使用。

### void ata\_eh\_qc\_retry（struct ata\_queued\_cmd \* qc）

告诉中间层在EH后重试ATA命令

**参数**

struct ata\_queued\_cmd \* qc

要重试的命令

指示中间和上层需要重试ATA命令。应从EH中使用。

SCSI中间层将重试次数限制为scmd->allowed。 scmd->allowed会增加由于不相关故障（qc-> err\_mask为零）而重试的命令。

### void ata\_dev\_disable（struct ata\_device \* dev）

禁用ATA设备

**参数**

struct ata\_device \* dev

要禁用的ATA设备

禁用dev。

锁定：EH上下文。

### void ata\_eh\_detach\_dev（struct ata\_device \* dev）

分离ATA设备

**参数**

struct ata\_device \* dev

要分离的ATA设备

分离dev。

锁定：无。

### void ata\_eh\_about\_to\_do（struct ata\_link \* link，struct ata\_device \* dev，unsigned int action）

即将执行eh\_action

**参数**

struct ata\_link \* link

目标ATA链接

struct ata\_device \* dev

执行每个设备操作的目标ATA dev（可以为NULL）

unsigned int action

即将执行的操作

执行EH操作前调用以清除link-> eh\_info中的相关位，使eh操作不被不必要地重复。

锁定：无。

### void ata\_eh\_done（struct ata\_link \* link，struct ata\_device \* dev，unsigned int action）

EH操作完成

**参数**

struct ata\_link \* link

完成EH操作的ATA链接

struct ata\_device \* dev

执行每个设备操作的目标ATA dev（可以为NULL）

unsigned int action

刚刚完成的操作

在执行EH操作后立即调用以清除link-> eh\_context中的相关位。

锁定：无。

### const char \* ata\_err\_string（unsigned int err\_mask）

将err\_mask转换为说明性字符串

**参数**

unsigned int err\_mask

要转换为字符串的错误掩码

将err\_mask转换为说明性字符串。错误根据严重程度进行优先级排序，仅报告最严重的错误。

锁定：无。

**返回**

err\_mask的说明性字符串

### int ata\_eh\_read\_log\_10h（struct ata\_device \* dev，int \* tag，struct ata\_taskfile \* tf）

读取NCQ错误详情的日志页面10h

**参数**

struct ata\_device \* dev

要从中读取日志页面10h的设备

int \* tag

失败的命令的结果标记

struct ata\_taskfile \* tf

失败的命令的结果taskfile寄存器

**说明**

读取日志页面10h以获取NCQ错误详细信息并清除错误条件。

锁定：内核线程上下文（可能会睡眠）。

**返回**

成功则为0，否则为-errno。

### unsigned int atapi\_eh\_tur（struct ata\_device \* dev，u8 \* r\_sense\_key）

执行ATAPI测试单元就绪

**参数**

struct ata\_device \* dev

目标ATAPI设备

u8 \* r\_sense\_key

感官键的输出参数

执行ATAPI TEST\_UNIT\_READY。

锁定：EH上下文（可能会睡眠）。

**返回**

成功则为0，失败则为AC\_ERR\_\*掩码。

### void ata\_eh\_request\_sense（struct ata\_queued\_cmd \* qc）

执行REQUEST\_SENSE\_DATA\_EXT

**参数**

struct ata\_queued\_cmd \*qc

要执行REQUEST\_SENSE\_SENSE\_DATA\_EXT的qc

在设备报告CHECK SENSE后执行REQUEST\_SENSE\_DATA\_EXT。这个函数是EH助手。

锁定：内核线程上下文（可能会休眠）。

### unsigned int atapi\_eh\_request\_sense(struct ata\_device \*dev，u8 \*sense\_buf，u8 dfl\_sense\_key）

执行ATAPI REQUEST\_SENSE

**参数**

struct ata\_device \*dev

要执行REQUEST\_SENSE的设备

u8 \*sense\_buf

结果感觉数据缓冲区（SCSI\_SENSE\_BUFFERSIZE字节长）

u8 dfl\_sense\_key

要使用的默认感觉键

在设备报告CHECK SENSE后执行ATAPI REQUEST\_SENSE。这个函数是EH助手。

锁定：内核线程上下文（可能会休眠）。

**返回**

成功为0，失败为AC\_ERR\_\*掩码

### void ata\_eh\_analyze\_serror(struct ata\_link \*link)

分析失败端口的SError

**参数**

struct ata\_link \*link

要分析SError的ATA链路

分析如果可用的SError并进一步确定失败的原因。

锁定：无。

### void ata\_eh\_analyze\_ncq\_error(struct ata\_link \*link)

分析NCQ错误

**参数**

struct ata\_link \* link

要分析NCQ错误的ATA链路

**说明**

读取日志页面10h，确定有问题的qc并获取错误状态TF。对于NCQ设备错误，所有LLDDs所要做的就是在ehi->err\_mask中设置AC\_ERR\_DEV。此函数负责其余部分。

锁定：内核线程上下文（可能会休眠）。

### unsigned int ata\_eh\_analyze\_tf(struct ata\_queued\_cmd \*qc)

分析失败的qc的taskfile

**参数**

struct ata\_queued\_cmd \* qc

要分析的qc

分析qc的taskfile并进一步确定失败的原因。如果可用，此函数还会请求ATAPI感觉数据。

锁定：内核线程上下文（可能会休眠）。

**说明**

确定的恢复动作

### unsigned int ata\_eh\_speed\_down\_verdict(struct ata\_device \*dev)

确定降速决定

**参数**

struct ata\_device \*dev

感兴趣的设备

此函数检查dev的错误环并确定是否需要关闭NCQ、降低传输速度或回退到PIO。

ECAT\_ATA\_BUS：任何命令的ATA\_BUS错误

ECAT\_TOUT\_HSMTIMEOUT适用于任何命令或IO命令的HSM违规

ECAT\_UNK\_DEV：IO命令的未知DEV错误

ECAT\_DUBIOUS\_\*与上面三个相同，但发生在

数据传输还没有得到验证。

裁决是

NCQ\_OFF：关闭NCQ

SPEED\_DOWN：降速但不回退

FALLBACK\_TO\_PIO：回退到PIO。

即使返回多个裁决，每个错误只采取一个动作。由非DUBIOUS错误触发的动作清除ERING，而由DUBIOUS\_\*错误触发的动作则不清除。这是为了在设备初始配置后加速降速决策。

以下是降速规则。#1和#2处理DUBIOUS错误。

如果在过去的5分钟内发生了多个DUBIOUS\_ATA\_BUS或DUBIOUS\_TOUT\_HSM错误，则速度减慢并回退到PIO。

如果在过去的5分钟内发生了多个DUBIOUS\_TOUT\_HSM或DUBIOUS\_UNK\_DEV错误，则关闭NCQ。

如果在过去的5分钟内发生了8个以上的ATA\_BUS、TOUT\_HSM或UNK\_DEV错误，则回退到PIO

如果在过去的10分钟内发生了3个以上的TOUT\_HSM或UNK\_DEV错误，则关闭NCQ。

如果在过去的10分钟内发生了3个以上的ATA\_BUS或TOUT\_HSM错误，或者6个以上的UNK\_DEV错误，则速度会降低。

锁定：承袭自调用者。

**返回**

OR ATA\_EH\_SPDN\_\*标志。

### unsigned int ata\_eh\_speed\_down(struct ata\_device \*dev，unsigned int eflags，unsigned int err\_mask）

记录错误并降速如果必要

**参数**

struct ata\_device \*dev

失败的设备

unsigned int eflags

ATA\_EFLAG\_\*标志的掩码

unsigned int err\_mask

错误的err\_mask

记录错误，并检查错误历史记录以确定是否需要调整传输速度。如果需要这样的调整，它还会适当地设置传输限制。

锁定：内核线程上下文（可能会休眠）。

**返回**

确定的恢复动作。

### int ata\_eh\_worth\_retry(struct ata\_queued\_cmd \*qc)

分析错误并决定是否重试

**参数**

struct ata\_queued\_cmd \* qc

可能重试的qc

查看错误的原因并决定是否重试可能有用。我们不想重试媒体错误，因为驱动器自己可能已经花费了10-30秒钟进行内部重试，然后才报告失败。

### bool ata\_eh\_quiet(struct ata\_queued\_cmd \*qc)

检查我们是否需要对命令错误保持安静

**参数**

struct ata\_queued\_cmd \* qc

要检查的qc

查看qc标志和其scsi命令请求标志，以确定我们是否需要对命令失败保持安静。

### void ata\_eh\_link\_autopsy(struct ata\_link \*link)

分析错误并确定恢复动作

**参数**

struct ata\_link \* link

要在其中执行尸体解剖的主机链接

分析为什么链接失败并确定需要哪些恢复操作。这个函数还设置了更详细的AC\_ERR\_\*值，并为ATAPI CHECK SENSE填充了感觉数据。

锁定：内核线程上下文（可能会休眠）。

### void ata\_eh\_autopsy(struct ata\_port \*ap)

分析错误并确定恢复动作

**参数**

struct ata\_port \*ap

要在其中执行尸体解剖的主机端口

分析ap的所有链接，并确定它们失败的原因和需要的恢复操作。

锁定：内核线程上下文（可能会休眠）。

### const char \* ata\_get\_cmd\_descript(u8 command)

获取ATA命令的说明

**参数**

u8 command

获取说明的ATA命令代码

**说明**

返回给定命令的文本说明，如果命令未知，则返回NULL。

锁定：无

### void ata\_eh\_link\_report(struct ata\_link \*link)

向用户报告错误处理

**参数**

struct ata\_link \*link

正在进行ATA链接EH

报告EH给用户。

锁定：无。

### void ata\_eh\_report(struct ata\_port \*ap)

向用户报告错误处理

**参数**

struct ata\_port \*ap

报告关于ATA端口的EH

报告EH给用户。

锁定：无。

### int ata\_set\_mode(struct ata\_link \*link, struct ata\_device \*\*r\_failed\_dev)

编程时序并发出SET FEATURES - XFER

**参数**

struct ata\_link \*link

在其上将编程定时的链接

struct ata\_device \*\*r\_failed\_dev

无效设备的输出参数

设置ATA设备磁盘传输模式（PIO3、UDMA6等）。如果ata\_set\_mode（）失败，则指向失败设备的指针将在r\_failed\_dev中返回。

锁定：PCI/等总线探测sem。

**返回**

成功返回0，否则返回负errno

### int atapi\_eh\_clear\_ua(struct ata\_device \*dev)

重设后清除ATAPI UNIT ATTENTION

**参数**

struct ata\_device \*dev

要为其清除UA的ATAPI设备

重置和其他操作可能会使ATAPI设备引发UNIT ATTENTION，这会导致下一操作失败。此函数清除UA。

锁定：EH上下文（可能休眠）。

**返回**

成功返回0，失败返回-errno。

### int ata\_eh\_maybe\_retry\_flush(struct ata\_device \*dev)

如果需要，重试FLUSH

**参数**

struct ata\_device \*dev

可能需要FLUSH重试的ATA设备

如果dev失败了FLUSH，需要立即向上层报告，因为这意味着dev无法重新映射并已经丢失至少一个扇区，并且进一步的FLUSH重试不会对丢失的扇区有任何影响。但是，如果FLUSH由于其他原因失败，例如传输错误，则需要重试FLUSH。

此函数确定是否需要FLUSH失败重试，如果需要，则执行它。

**返回**

EH可以继续，则返回0，否则返回-errno。

### int ata\_eh\_set\_lpm(struct ata\_link \*link, enum ata\_lpm\_policy policy, struct ata\_device \*\*r\_failed\_dev)

配置SATA接口电源管理

**参数**

struct ata\_link \*link

配置电源管理的链接

enum ata\_lpm\_policy policy

链接电源管理策略

struct ata\_device \*\*r\_failed\_dev

无效设备的输出参数

启用SATA接口电源管理。这将为min\_power和medium\_power\_with\_dipm策略启用设备接口电源管理（DIPM），然后调用特定于驱动程序的回调以启用Host Initiated Power管理。

锁定：EH上下文。

**返回**

成功返回0，失败返回-errno。

### int ata\_eh\_recover(struct ata\_port \*ap, ata\_prereset\_fn\_t prereset, ata\_reset\_fn\_t softreset, ata\_reset\_fn\_t hardreset, ata\_postreset\_fn\_t postreset, struct ata\_link \*\*r\_failed\_link)

在错误之后恢复主机端口

**参数**

struct ata\_port \*ap

要恢复的主机端口

ata\_prereset\_fn\_t prereset

prereset方法（可以为NULL）

ata\_reset\_fn\_t softreset

softreset方法（可以为NULL）

ata\_reset\_fn\_t hardreset

hardreset方法（可以为NULL）

ata\_postreset\_fn\_t postreset

postreset方法（可以为NULL）

struct ata\_link \*\*r\_failed\_link

无效链接的输出参数

这是libata异常处理的Alpha和Omega，Eum和Yang，心脏和灵魂。进入时，记录了恢复每个链接和热插拔请求所需的操作在链接的eh\_context中。此函数使用适当的重试和回退执行所有操作，以使失败的设备复活，分离上场者并向新手致以问候。

锁定：内核线程上下文（可能休眠）。

**返回**

成功返回0，失败返回-errno。

### void ata\_eh\_finish(struct ata\_port \*ap)

完成EH

**参数**

struct ata\_port \*ap

要完成EH的主机端口

恢复完成。清理EH状态并重试或完成失败的qcs。

锁定：无。

### void ata\_do\_eh(struct ata\_port \*ap, ata\_prereset\_fn\_t prereset, ata\_reset\_fn\_t softreset, ata\_reset\_fn\_t hardreset, ata\_postreset\_fn\_t postreset)

执行标准错误处理

**参数**

struct ata\_port \*ap

要为其处理错误的主机端口

ata\_prereset\_fn\_t prereset

prereset方法（可以为NULL）

ata\_reset\_fn\_t softreset

softreset方法（可以为NULL）

ata\_reset\_fn\_t hardreset

hardreset方法（可能为NULL）

ata\_postreset\_fn\_t postreset

postreset方法（可能为NULL）

执行标准错误处理序列。

锁定：内核线程上下文（可能休眠）。

### void ata\_std\_error\_handler(struct ata\_port \*ap)

标准错误处理程序

**参数**

struct ata\_port \*ap

要为其处理错误的主机端口

标准错误处理程序

锁定：内核线程上下文（可能休眠）。

### void ata\_eh\_handle\_port\_suspend(struct ata\_port \*ap)

执行端口挂起操作

**参数**

struct ata\_port \*ap

要挂起的端口

挂起ap。

锁定：内核线程上下文（可能休眠）。

### void ata\_eh\_handle\_port\_resume(struct ata\_port \*ap)

执行端口恢复操作

**参数**

struct ata\_port \*ap

要恢复的端口

恢复ap。

锁定：内核线程上下文（可能休眠）。

## libata SCSI转换/仿真

### struct ata\_port \* ata\_sas\_port\_alloc(struct ata\_host \* host, struct ata\_port\_info \* port\_info, struct Scsi\_Host \* shost)

为连接的SATA设备分配端口

**参数**

struct ata\_host \* host

所有SAS端口的ATA主机容器

struct ata\_port\_info \* port\_info

来自低级主机驱动程序的信息

struct Scsi\_Host \* shost

SCSI主机连接到的scsi设备

**说明**

锁定：PCI/等总线探测sem。

**返回**

成功时的ata\_port指针/失败时的NULL。

### int ata\_sas\_port\_start(struct ata\_port \* ap)

设置DMA端口。

**参数**

struct ata\_port \* ap

要初始化的端口

**说明**

在每个端口的数据结构初始化后调用。

可以用作ata\_port\_operations中的port\_start()入口。

锁定：从调用者继承。

### void ata\_sas\_port\_stop(struct ata\_port \* ap)

撤消ata\_sas\_port\_start（）

**参数**

struct ata\_port \* ap

要关闭的端口

**说明**

可以用作ata\_port\_operations中的port\_stop()入口。

锁定：从调用者继承。

void ata\_sas\_async\_probe(struct ata\_port \* ap)

简单地安排探测并**返回**

**参数**

struct ata\_port \* ap

要探测的端口

**说明**

用于批量调度sas附加的ata设备的探测，假定端口已经通过ata\_sas\_port\_init()。

### int ata\_sas\_port\_init(struct ata\_port \* ap)

初始化SATA设备

**参数**

struct ata\_port \* ap

要初始化的SATA端口

**说明**

锁定：PCI / etc。总线探测信号。

**返回**

成功为零，出错为非零。

### void ata\_sas\_port\_destroy(struct ata\_port \* ap)

销毁由ata\_sas\_port\_alloc分配的SATA端口

**参数**

struct ata\_port \* ap

要销毁的SATA端口

### int ata\_sas\_slave\_configure(struct scsi\_device \* sdev，struct ata\_port \* ap)

用于libata设备的默认slave\_config例程

**参数**

struct scsi\_device \* sdev

要配置的SCSI设备

struct ata\_port \* ap

连接SCSI设备的ATA端口

**返回**

零。

### int ata\_sas\_queuecmd(struct scsi\_cmnd \* cmd，struct ata\_port \* ap)

向由libata管理的设备发出SCSI cdb

**参数**

struct scsi\_cmnd \* cmd

要发送的SCSI命令

struct ata\_port \* ap

发送命令的ATA端口

**返回**

来自\_\_ata\_scsi\_queuecmd（）的返回，如果可以排队cmd，则返回0。

### int ata\_std\_bios\_param(struct scsi\_device \* sdev，struct block\_device \* bdev，sector\_t capacity，int geom)

由sd使用的通用BIOS头/扇区/磁柱计算器。

**参数**

struct scsi\_device \* sdev

要确定BIOS几何形状的SCSI设备

struct block\_device \* bdev

与sdev相关联的块设备

sector\_t capacity

SCSI设备的容量

int geom

将几何图形输出到的位置

**说明**

由sd使用的通用BIOS头/扇区/磁柱计算器。现在，大多数BIOS都期望XXX / 255/16（CHS）映射。如果不使用此方法，则可能出现某些情况导致磁盘无法启动。

锁定：由SCSI层定义。我们并不关心。

**返回**

零。

### void ata\_scsi\_unlock\_native\_capacity(struct scsi\_device \* sdev)

解锁本机容量

**参数**

struct scsi\_device \* sdev

要调整设备容量的SCSI设备

**说明**

如果sdev上的分区超出设备的末尾，则调用此函数。它请求EH解锁HPA。

锁定：由SCSI层定义。可能睡眠。

### int ata\_get\_identity(struct ata\_port \* ap，struct scsi\_device \* sdev，void \_\_user \* arg)

HDIO\_GET\_IDENTITY ioctl的处理程序

**参数**

struct ata\_port \* ap

目标端口

struct scsi\_device \* sdev

要获取识别数据的SCSI设备

void \_\_user \* arg

识别数据的用户缓冲区区域

**说明**

锁定：由SCSI层定义。我们并不关心。

**返回**

成功为零，负错误号为错误。

### int ata\_cmd\_ioctl（struct scsi\_device \* scsidev，void \_\_user \* arg）

HDIO\_DRIVE\_CMD ioctl的处理程序

**参数**

struct scsi\_device \* scsidev

我们正在向其发出命令的设备

void \_\_user \* arg

用于发出命令的用户提供的数据

**说明**

锁定：由SCSI层定义。我们并不关心。

**返回**

成功为零，负错误号为错误。

### int ata\_task\_ioctl（struct scsi\_device \* scsidev，void \_\_user \* arg）

HDIO\_DRIVE\_TASK ioctl的处理程序

**参数**

struct scsi\_device \* scsidev

我们正在向其发出命令的设备

void \_\_user \* arg

用于发出命令的用户提供的数据

**说明**

锁定：由SCSI层定义。我们并不关心。

**返回**

成功为零，负错误号为错误。

### struct ata\_queued\_cmd \* ata\_scsi\_qc\_new（struct ata\_device \* dev，struct scsi\_cmnd \* cmd）

获取新的ata\_queued\_cmd引用

**参数**

struct ata\_device \* dev

新命令所连接的ATA设备

struct scsi\_cmnd \* cmd

发起此ATA命令的SCSI命令

**说明**

获取对未使用的ata\_queued\_cmd结构的引用，该结构是表示发送到硬件的单个ATA命令的基本libata结构。

如果命令可用，请使用当前命令的信息填写结构的SCSI特定部分。

锁定：spin\_lock\_irqsave（主机锁定）

**返回**

分配的命令，如果没有可用则为NULL。

### void ata\_dump\_status（unsigned id，struct ata\_taskfile \* tf）

错误信息的用户友好显示

**参数**

unsigned id

问的端口ID

struct ata\_taskfile \* tf

填好的任务文件指针

**说明**

解码并转储ATA错误/状态寄存器，以便用户了解在非虚构层面上实际发生了什么。

锁定：从调用者继承

### void ata\_to\_sense\_error（unsigned id，u8 drv\_stat，u8 drv\_err，u8 \* sk，u8 \* asc，u8 \* ascq，int verbose）

将ATA错误转换为SCSI错误

**参数**

unsigned id

ATA设备号

u8 drv\_stat

ATA状态寄存器中包含的值

u8 drv\_err

ATA错误寄存器中包含的值

u8 \* sk

SCSI密钥

u8 \* asc

SCSI进一步码

u8 \* ascq

SCSI进一步码限定符

int verbose

我们将要填写的感应键

u8 \*asc

我们将要填写的附加感应码

u8 \*ascq

我们将要填写的附加感应码限定符

int verbose

详细说明

将ATA错误转换为SCSI错误。填写SK、ASC和ASCQ字节的指针，以便后续在固定或说明符格式的感应块中使用。

锁定：spin\_lock\_irqsave(host lock)

### void ata\_gen\_ata\_sense(struct ata\_queued\_cmd \*qc)

生成SCSI固定感应块

**参数**

struct ata\_queued\_cmd \*qc

我们正在出错的命令

详细说明

为失败的ATA命令qc生成感应块。使用说明符格式以适应LBA48块地址。

锁定：无。

### int atapi\_drain\_needed(struct request \*rq)

检查数据传输是否可能溢出

**参数**

struct request \*rq

要检查的请求

详细说明

传输可变长度数据到主机的ATAPI命令可能由于应用程序错误或硬件错误而溢出。此函数检查是否应该排空和忽略请求的溢出。

锁定：无。

**返回**

1如果是；否则为0。

### int ata\_scsi\_slave\_config(struct scsi\_device \*sdev)

设置SCSI设备属性

**参数**

struct scsi\_device \*sdev

要检查的SCSI设备

详细说明

在我们实际开始读写设备之前调用此函数，以配置某些SCSI中间层行为。

锁定：由SCSI层定义。我们并不在意。

### void ata\_scsi\_slave\_destroy(struct scsi\_device \*sdev)

SCSI设备即将被销毁

**参数**

struct scsi\_device \*sdev

要销毁的SCSI设备

**说明**

sdev即将被热拔插销毁。如果此拔插由NULL dev->sdev指示的libata发起，则此函数不必做任何事情。否则，正在进行SCSI层启动的热插拔。清除dev->sdev，将设备安排为ATA分离并调用EH。

锁定：由SCSI层定义。我们并不在意。

### int \_\_ata\_change\_queue\_depth(struct ata\_port \*ap, struct scsi\_device \*sdev, int queue\_depth)

用于ata\_scsi\_change\_queue\_depth的帮助程序

**参数**

struct ata\_port \*ap

要更改队列深度的设备的ATA端口

struct scsi\_device \*sdev

要为其配置队列深度的SCSI设备

int queue\_depth

新的队列深度

说明

libsas和libata在将sdev关联到其ata\_port方面采用不同的方法。

### int ata\_scsi\_change\_queue\_depth(struct scsi\_device \*sdev, int queue\_depth)

队列深度配置的SCSI回调

**参数**

struct scsi\_device \*sdev

要为其配置队列深度的SCSI设备

int queue\_depth

新的队列深度

**说明**

这是libata标准hostt->change\_queue\_depth回调。当用户尝试通过sysfs设置队列深度时，SCSI将调用此回调。

锁定：SCSI层（我们不在意）

**返回**

新配置的队列深度。

### unsigned int ata\_scsi\_start\_stop\_xlat(struct ata\_queued\_cmd \*qc)

将SCSI START STOP UNIT命令转换为ATA命令

**参数**

struct ata\_queued\_cmd \*qc

存储已翻译的ATA任务文件

**说明**

设置ATA任务文件以发出STANDBY（停止）或READ VERIFY（启动）。也许这些命令应该先由CHECK POWER MODE跟踪，以查看设备已经在哪种电源模式下。【参见www.t10.org的SAT修订版5】

锁定：spin\_lock\_irqsave(host lock)

**返回**

成功时为零，错误时为非零。

### unsigned int ata\_scsi\_flush\_xlat(struct ata\_queued\_cmd \*qc)

将SCSI SYNCHRONIZE CACHE命令转换为ATA命令

**参数**

struct ata\_queued\_cmd \*qc

存储已翻译的ATA任务文件

**说明**

设置ATA任务文件以发出FLUSH CACHE或FLUSH CACHE EXT。

锁定：spin\_lock\_irqsave(host lock)

**返回**

成功时为零，错误时为非零。

### void scsi\_6\_lba\_len(const u8 \*cdb, u64 \*plba, u32 \*plen)

获取LBA和传输长度

**参数**

const u8 \*cdb

要翻译的SCSI命令

u64 \*plba

LBA

u32 \*plen

传输长度

**说明**

计算6字节命令的LBA和传输长度。

### void scsi\_10\_lba\_len(const u8 \*cdb, u64 \*plba, u32 \*plen)

获取LBA和传输长度

**参数**

const u8 \*cdb

要翻译的SCSI命令

u64 \*plba

LBA

u32 \*plen

传输长度

**说明**

计算10字节命令的LBA和传输长度。

### void scsi\_16\_lba\_len(const u8 \*cdb, u64 \*plba, u32 \*plen)

获取LBA和传输长度

**参数**

const u8 \*cdb

要翻译的SCSI命令

u64 \*plba

LBA

u32 \*plen

传输长度

**说明**

计算16字节命令的LBA和传输长度。

### unsigned int ata\_scsi\_verify\_xlat(struct ata\_queued\_cmd \*qc)

将SCSI VERIFY命令转换为ATA命令

**参数**

struct ata\_queued\_cmd \*qc

存储已翻译的ATA任务文件

**说明**

将SCSI VERIFY命令转换为ATA读验证命令。

锁定：spin\_lock\_irqsave(host lock)

**返回**

成功时为零，错误时为非零。

### unsigned int ata\_scsi\_rw\_xlat(struct ata\_queued\_cmd \*qc)

将SCSI读/写命令转换为ATA命令

**参数**

struct ata\_queued\_cmd \*qc

存储已翻译的ATA任务文件

**说明**

将六个SCSI读/写命令中的任何一个转换为ATA对应命令，包括起始扇区（LBA）、扇区计数，并考虑设备的LBA48支持。

READ\_6、READ\_10、READ\_16、WRITE\_6、WRITE\_10和WRITE\_16目前受支持。

锁定：spin\_lock\_irqsave（主机锁）

**返回**

成功时返回零，出错时返回非零。

### int ata\_scsi\_translate（struct ata\_device \* dev，struct scsi\_cmnd \* cmd，ata\_xlat\_func\_t xlat\_func）

翻译并发出SCSI命令到ATA设备

**参数**

struct ata\_device \* dev

命令要发送的ATA设备

struct scsi\_cmnd \* cmd

要执行的SCSI命令

ata\_xlat\_func\_t xlat\_func

翻译器，将cmd转换为ATA任务文件

**说明**

我们的 ->:c:func:queuecommand（）函数已经决定，发出的SCSI命令可以直接翻译成ATA命令，而不是在内部处理。

这个函数为SCSI命令设置一个ata\_queued\_cmd结构，并把该ata\_queued\_cmd发送给硬件。

xlat\_func参数（演员）返回0，如果准备好执行ATA命令，否则返回1以完成翻译。如果返回1，则假定cmd->result（和可能是cmd->sense\_buffer）已设置，表示错误条件或干净（早期）终止。

锁定：spin\_lock\_irqsave（主机锁）

**返回**

成功时返回0，如果需要延迟命令，则返回SCSI\_ML\_QUEUE\_DEVICE\_BUSY。

### void \* ata\_scsi\_rbuf\_get（struct scsi\_cmnd \* cmd，bool copy\_in，unsigned long \* flags）

映射响应缓冲区。

**参数**

struct scsi\_cmnd \* cmd

要映射的缓冲区SCSI命令。

bool copy\_in

从用户缓冲区中进行复制

unsigned long \* flags

存储irq启用状态的unsigned long变量。

**说明**

为模拟SCSI命令准备缓冲区。

锁定：spin\_lock\_irqsave（ata\_scsi\_rbuf\_lock），成功时

**返回**

响应缓冲区的指针。

### void ata\_scsi\_rbuf\_put（struct scsi\_cmnd \* cmd，bool copy\_out，unsigned long \* flags）

取消映射响应缓冲区。

**参数**

struct scsi\_cmnd \* cmd

要取消映射的缓冲区的SCSI命令。

bool copy\_out

复制输出结果

unsigned long \* flags

传递到ata\_scsi\_rbuf\_get()的flags参数。

**说明**

返回rbuf缓冲区。如果copy\_back为true，则将结果复制到cmd的缓冲区中。

锁定：解锁ata\_scsi\_rbuf\_lock。

### void ata\_scsi\_rbuf\_fill（struct ata\_scsi\_args \* args，unsigned int（\*actor）（struct ata\_scsi\_args \* args，u8 \* rbuf）

SCSI命令模拟器的包装器

**参数**

struct ata\_scsi\_args \* args

设备识别数据/感兴趣的SCSI命令。

unsigned int（\*）（struct ata\_scsi\_args \* args，u8 \* rbuf）演员

所需SCSI命令模拟器的回调钩子

**说明**

负责模拟SCSI命令的艰巨工作...映射响应缓冲区，调用命令的处理程序，并处理处理程序的返回。这个返回指示处理程序是否希望SCSI命令成功完成（0），或不希望（这种情况下假定设置了cmd->result和sense\_buffer）。

锁定：spin\_lock\_irqsave（主机锁）

### unsigned int ata\_scsiop\_inq\_std（struct ata\_scsi\_args \* args，u8 \* rbuf）

模拟INQUIRY命令

**参数**

struct ata\_scsi\_args \* args

设备识别数据/感兴趣的SCSI命令。

u8 \* rbuf

将模拟的SCSI cmd输出发送到的响应缓冲区。

**说明**

**返回**与非VPD INQUIRY命令输出相关联的标准设备识别数据。

锁定：spin\_lock\_irqsave（主机锁）

### unsigned int ata\_scsiop\_inq\_00（struct ata\_scsi\_args \* args，u8 \* rbuf）

模拟INQUIRY VPD页面0，页面列表

**参数**

struct ata\_scsi\_args \* args

设备识别数据/感兴趣的SCSI命令。

u8 \* rbuf

将模拟的SCSI cmd输出发送到的响应缓冲区。

**说明**

返回可用的inquiry VPD页面列表。

锁定：spin\_lock\_irqsave（主机锁）

### unsigned int ata\_scsiop\_inq\_80（struct ata\_scsi\_args \* args，u8 \* rbuf）

模拟INQUIRY VPD页面80，设备序列号

**参数**

struct ata\_scsi\_args \* args

设备识别数据/感兴趣的SCSI命令。

u8 \* rbuf

将模拟的SCSI cmd输出发送到的响应缓冲区。

**说明**

返回ATA设备序列号。

锁定：spin\_lock\_irqsave（主机锁）

### unsigned int ata\_scsiop\_inq\_83（struct ata\_scsi\_args \* args，u8 \* rbuf）

模拟INQUIRY VPD页面83，设备身份

**参数**

struct ata\_scsi\_args \* args

设备识别数据/感兴趣的SCSI命令。

u8 \* rbuf

将模拟的SCSI cmd输出发送到的响应缓冲区。

**说明**

生成包含ATA序列号的供应商特定ASCII，以及SAT**定义**的基于供应商ID的ASCII供应商名称（“ATA”），型号和序列号的逻辑单元设备标识符。

锁定：spin\_lock\_irqsave（主机锁）

### unsigned int ata\_scsiop\_inq\_89（struct ata\_scsi\_args \* args，u8 \* rbuf）

模拟INQUIRY VPD页面89，ATA信息

**参数**

struct ata\_scsi\_args \* args

设备识别数据/感兴趣的SCSI命令。

u8 \* rbuf

将模拟的SCSI cmd输出发送到的响应缓冲区。

**说明**

产生SAT指定的ATA VPD页面。

锁定：spin\_lock\_irqsave（主机锁）

### void modecpy（u8 \* dest，const u8 \* src，int n，bool changeable）

为MODE SENSE准备响应

**参数**

u8 \* dest

输出缓冲区

const u8 \* src

正在复制的数据

int n

模式页的长度

bool changeable

是否请求可变参数

**说明**

为当前或可变参数生成通用MODE SENSE页面。

锁定：无。

### unsigned int ata\_msense\_caching（u16 \* id，u8 \* buf，bool changeable）

模拟MODE SENSE高速缓存信息页面

**参数**

u16 \* id

设备识别数据

u8 \* buf

输出缓冲区

bool changeable

是否请求可变参数

**说明**

生成一个缓存信息页面，根据设备的能力，有条件地指示写缓存到SCSI层。

锁定：无。

### unsigned int ata\_msense\_control（struct ata\_device \* dev，u8 \* buf，bool changeable）

模拟MODE SENSE控制模式页面

**参数**

struct ata\_device \* dev

感兴趣的ATA设备

u8 \* buf

输出缓冲区

bool changeable

是否请求可更改参数

**说明**

生成通用的MODE SENSE控制模式页面。

锁定：无。

### unsigned int ata\_msense\_rw\_recovery(u8 \* buf, bool changeable)

模拟MODE SENSE读/写错误恢复页面

**参数**

u8 \* buf

输出缓冲区

bool changeable

是否请求可更改参数

**说明**

生成通用的MODE SENSE读/写错误恢复页面。

锁定：无。

### unsigned int ata\_scsiop\_mode\_sense(struct ata\_scsi\_args \* args, u8 \* rbuf)

模拟MODE SENSE 6，10命令

**参数**

struct ata\_scsi\_args \* args

设备IDENTIFY数据/感兴趣的SCSI命令。

u8 \* rbuf

响应缓冲区，模拟SCSI cmd输出发送到其中。

**说明**

模拟MODE SENSE命令。假设只针对直接访问设备（例如磁盘）调用。其他设备类型应该没有块说明符。

锁定：spin\_lock\_irqsave（主机锁定）

### unsigned int ata\_scsiop\_read\_cap(struct ata\_scsi\_args \* args, u8 \* rbuf)

模拟读CAPACITY [16]命令

**参数**

struct ata\_scsi\_args \* args

设备IDENTIFY数据/感兴趣的SCSI命令。

u8 \* rbuf

响应缓冲区，模拟SCSI cmd输出发送到其中。

**说明**

模拟读CAPACITY命令。

锁定：无。

### unsigned int ata\_scsiop\_report\_luns(struct ata\_scsi\_args \* args, u8 \* rbuf)

模拟REPORT LUNS命令

**参数**

struct ata\_scsi\_args \* args

设备IDENTIFY数据/感兴趣的SCSI命令。

u8 \* rbuf

响应缓冲区，模拟SCSI cmd输出发送到其中。

**说明**

模拟REPORT LUNS命令。

锁定：spin\_lock\_irqsave（主机锁定）

### unsigned int atapi\_xlat(struct ata\_queued\_cmd \* qc)

初始化数据包任务文件

**参数**

struct ata\_queued\_cmd \* qc

要初始化的命令结构

**说明**

锁定：spin\_lock\_irqsave（主机锁定）

**返回**

成功返回零，失败返回非零。

### struct ata\_device \* ata\_scsi\_find\_dev(struct ata\_port \* ap，const struct scsi\_device \* scsidev)

查找scsi\_cmnd的ata\_device

**参数**

struct ata\_port \* ap

设备附加到的ATA端口

const struct scsi\_device \* scsidev

我们从中派生ATA设备的SCSI设备

**说明**

根据struct scsi\_cmnd中提供的各种信息，将其映射到ATA总线，并使用此映射确定与要发送的SCSI命令相关联的ata\_device。

锁定：spin\_lock\_irqsave（主机锁定）

**返回**

关联的ATA设备，如果未找到则为NULL。

### unsigned int ata\_scsi\_pass\_thru(struct ata\_queued\_cmd \* qc)

将ATA pass-thru CDB转换为任务文件

**参数**

struct ata\_queued\_cmd \* qc

要初始化的命令结构

**说明**

处理12、16或32字节版本的CDB。

**返回**

成功返回零，失败返回非零。

### size\_t ata\_format\_dsm\_trim\_descr(struct scsi\_cmnd \* cmd，u32 trmax，u64 sector，u32 count)

SATL Write Same转换为DSM Trim

**参数**

struct scsi\_cmnd \* cmd

正在转换的SCSI命令

u32 trmax

适合sector\_size字节的最大条目数。

u64 sector

起始扇区

u32 count

逻辑扇区的总范围

**说明**

重写WRITE SAME说明符为DSM TRIM小端格式的说明符。

最多64个条目，格式如下：

63:48范围长度47:0 LBA

0的范围长度被忽略。 LBA应按顺序排序且不重叠。

注意

这与ADD LBA（S）TO NV CACHE PINNED SET格式相同

**返回**

复制到sglist中的字节数。

### unsigned int ata\_scsi\_write\_same\_xlat(struct ata\_queued\_cmd \* qc)

SATL Write Same转换为ATA SCT Write Same

**参数**

struct ata\_queued\_cmd \* qc

要转换的命令

**说明**

将SCSI WRITE SAME命令转换为DSM TRIM命令或SCT Write Same命令。基于WRITE SAME是否具有UNMAP标志：

当设置时，转换为DSM TRIM

当清除时，转换为SCT Write Same

### unsigned int ata\_scsiop\_maint\_in(struct ata\_scsi\_args \* args，u8 \* rbuf)

模拟MAINTENANCE\_IN的子集

**参数**

struct ata\_scsi\_args \* args

设备MAINTENANCE\_IN数据/感兴趣的SCSI命令。

u8 \* rbuf

响应缓冲区，模拟SCSI cmd输出发送到其中。

**说明**

产生一个子集，以满足scsi\_report\_opcode（）

锁定：spin\_lock\_irqsave（主机锁定）

### void ata\_scsi\_report\_zones\_complete(struct ata\_queued\_cmd \* qc)

转换ATA输出

**参数**

struct ata\_queued\_cmd \* qc

返回数据的命令结构

**说明**

将T-13小端字段表示转换为T-10大端字段表示。一团糟。

### int ata\_mselect\_caching(struct ata\_queued\_cmd \* qc，const u8 \* buf，int len，u16 \* fp)

模拟用于缓存信息页面的MODE SELECT

**参数**

struct ata\_queued\_cmd \* qc

用于翻译的存储ATA任务文件

const u8 \* buf

输入缓冲区

int len

输入缓冲区中有效字节数

u16 \* fp

错误时的失败字段的输出参数

**说明**

准备一个任务文件以修改设备的缓存信息。

锁定：无。

### int ata\_mselect\_control(struct ata\_queued\_cmd \* qc，const u8 \* buf，int len，u16 \* fp)

模拟控制页面的MODE SELECT

**参数**

struct ata\_queued\_cmd \* qc

用于翻译的存储ATA任务文件

const u8 \* buf

输入缓冲区

int len

输入缓冲区中有效字节数

u16 \* fp

错误时的失败字段的输出参数

**说明**

准备一个任务文件以修改设备的缓存信息。

锁定：无。

### unsigned int ata\_scsi\_mode\_select\_xlat(struct ata\_queued\_cmd \* qc)

模拟 MODE SELECT 6, 10 命令

**参数**

struct ata\_queued\_cmd \* qc

用于翻译 ATA 任务文件的存储空间

**说明**

将 MODE SELECT 命令转换为 ATA SET FEATURES 任务文件。假定这仅针对直接访问设备（例如磁盘）。对于其他设备类型，不应有块说明符。

锁定：spin\_lock\_irqsave（host lock）

### unsigned int ata\_scsi\_var\_len\_cdb\_xlat(struct ata\_queued\_cmd \* qc)

SATL 可变长度 CDB 到处理程序

**参数**

struct ata\_queued\_cmd \* qc

要翻译的命令

**说明**

将 SCSI 可变长度 CDB 转换为指定的命令。它检查 CDB 中的服务操作值以调用相应的处理程序。

**返回**

成功则为零，失败则为非零

### ata\_xlat\_func\_t ata\_get\_xlat\_func(struct ata\_device \* dev，u8 cmd)

检查 SCSI 到 ATA 翻译是否可行

**参数**

struct ata\_device \* dev

ATA 设备

u8 cmd

要考虑的 SCSI 命令操作码

**说明**

查找给定的 SCSI 命令，并确定是否翻译或模拟 SCSI 命令。

**返回**

如果可能，则返回指向翻译函数的指针；否则返回 NULL。

### void ata\_scsi\_dump\_cdb(struct ata\_port \* ap，struct scsi\_cmnd \* cmd)

将 SCSI 命令内容转储到 dmesg

**参数**

struct ata\_port \* ap

命令要发送到的 ATA 端口

struct scsi\_cmnd \* cmd

要转储的 SCSI 命令

**说明**

通过 printk() 打印 SCSI 命令的内容。

### int ata\_scsi\_queuecmd(struct Scsi\_Host \* shost，struct scsi\_cmnd \* cmd)

将 SCSI cdb 发送到 libata 管理的设备

**参数**

struct Scsi\_Host \* shost

要发送的命令的 SCSI 主机

struct scsi\_cmnd \* cmd

要发送的 SCSI 命令

**说明**

在某些情况下，此函数将 SCSI 命令转换为 ATA 任务文件，并将任务文件排队以发送到硬件。在其他情况下，此函数通过评估和响应某些 SCSI 命令模拟 SCSI 设备。从而使 ATA 和 ATAPI 设备出现为 SCSI 设备。

锁定：ATA 主机锁

**返回**

如果 cmd 可以排队，则返回\_\_ata\_scsi\_queuecmd() 的返回；否则返回 0。

### void ata\_scsi\_simulate(struct ata\_device \* dev，struct scsi\_cmnd \* cmd)

模拟 ATA 设备上的 SCSI 命令

**参数**

struct ata\_device \* dev

目标设备

struct scsi\_cmnd \* cmd

正在发送到设备的 SCSI 命令。

**说明**

解释并直接执行可以内部处理的一组 SCSI 命令。

锁定：spin\_lock\_irqsave（host lock）

### int ata\_scsi\_offline\_dev(struct ata\_device \* dev)

下线附加的 SCSI 设备

**参数**

struct ata\_device \* dev

要下线 SCSI 设备的 ATA 设备

**说明**

此函数从 ata\_eh\_hotplug() 调用，并负责将附加到 dev 的 SCSI 设备下线。此函数使用保护 dev->sdev 不被清除的主机锁进行调用。

锁定：spin\_lock\_irqsave（host lock）

**返回**

存在附加的 SCSI 设备则为 1，否则为 0。

### void ata\_scsi\_remove\_dev(struct ata\_device \* dev)

删除附加的 SCSI 设备

**参数**

struct ata\_device \* dev

要删除已附加 SCSI 设备的 ATA 设备

**说明**

此函数从 ata\_eh\_scsi\_hotplug() 调用，并负责删除附加到 dev 的 SCSI 设备。

锁定：内核线程上下文（可能会睡眠）。

### void ata\_scsi\_media\_change\_notify(struct ata\_device \* dev)

发送媒体更改事件

**参数**

struct ata\_device \* dev

具有媒体更改事件的磁盘设备的指针

**说明**

告诉块层发送媒体更改通知事件。

锁定：spin\_lock\_irqsave（host lock）

### void ata\_scsi\_hotplug(struct work\_struct \* work)

热插拔的 SCSI 部分

**参数**

struct work\_struct \* work

要在其上执行 SCSI 热插拔的 ATA 端口的指针

**说明**

执行 SCSI 部分的热插拔。EH 完成后，它从分离的工作队列中执行。这是必要的，因为 SCSI 热插拔需要工作的 EH，并且热拔插与互斥体同步。

锁定：内核线程上下文（可能会睡眠）。

### int ata\_scsi\_user\_scan(struct Scsi\_Host \* shost，unsigned int channel，unsigned int id，u64 lun)

用户启动的总线扫描指示

**参数**

结构 Scsi\_Host \* shost

要扫描的 SCSI 主机

unsigned int channel

要扫描的通道

unsigned int id

要扫描的 ID

u64 lun

要扫描的 LUN

**说明**

当用户明确请求总线扫描时调用此函数。设置探测挂起标志并调用 EH。

锁定：SCSI 层（我们不关心）

**返回**

零。

### voidata\_scsi\_dev\_rescan(struct work\_struct \* work)

启动 scsi\_rescan\_device()

**参数**

struct work\_struct \* work

要执行 scsi\_rescan\_device() 操作的 ATA 端口的指针

**说明**

执行 ATA pass thru (SAT) 命令成功后，libata 需要将更改传播到 SCSI 层。

锁定：内核线程上下文（可能会睡眠）。

## ATA 错误和异常

本章尝试确定 ATA/ATAPI 设备存在哪些错误/异常情况，并以实现中立的方式说明它们应该如何处理。

术语“错误”用于说明设备或命令是否超时报告明确的错误情况。

术语“异常”用于说明非错误的异常情况（例如电源或热插拔事件），或用于同时说明错误和非错误的异常情况。在必要区分错误和异常的情况下，使用术语“非错误异常”。

### 异常分类

异常主要针对历史任务文件+总线主控IDE接口进行说明。如果控制器提供了其他更好的错误报告机制，则将其映射到下面说明的类别中不应该很困难。

在以下章节中，提到了两种恢复操作-重置和重新配置传输。这些在EH恢复操作中进一步说明。

#### HSM违规

在发出或执行任何ATA / ATAPI命令期间，如果STATUS值与HSM要求不匹配，则指示此错误。

在尝试发出命令时，ATA\_STATUS不包含!BSY && DRDY && !DRQ。

PIO数据传输期间的!BSY && !DRQ。

命令完成时的DRQ。

在CDB传输开始但CDB的最后一个字节传输之前，!BSY && ERR。ATA / ATAPI标准规定：“在命令包的最后一个字节写入之前，设备不得以错误终止PACKET命令”在PACKET命令的错误输出说明和状态图中不包括这样的转换。

在这些情况下，HSM被违反，从STATUS或ERROR寄存器中不能获得有关错误的太多信息。换句话说，这个错误可能是任何东西-驱动程序错误，故障设备，控制器和/或电缆。

由于HSM被违反，重置是恢复已知状态的必要步骤。降低传输速度的重新配置也可能是有帮助的，因为传输错误有时会导致这种类型的错误。

#### ATA / ATAPI设备错误（非NCQ /非检查条件）

这些是由ATA / ATAPI设备检测到并报告的指示设备问题的错误。对于此类错误，STATUS和ERROR寄存器的值是有效的并说明错误条件。请注意，某些ATA总线错误由ATA / ATAPI设备检测到，并使用与设备错误相同的机制报告。这些情况稍后在本节中说明。

对于ATA命令，!BSY && ERR在命令执行和完成期间表示此类错误。

对于ATAPI命令，

在发出PACKET后的!BSY && ERR && ABRT表示不支持PACKET命令并属于此类别。

在传输CDB的最后一个字节后，!BSY && ERR（==CHK）&& !ABRT表示检查条件并且不属于此类别。

在传输CDB的最后一个字节后，!BSY && ERR（==CHK）&& ABRT \*可能\*表示检查条件，不属于此类别。

在上述发现的错误中，以下不是ATA / ATAPI设备错误，而是ATA总线错误，应根据ATA总线错误进行处理。

**数据传输期间的CRC错误**

这由ERROR寄存器中的ICRC位指示，并表示在数据传输过程中发生了损坏。到ATA / ATAPI-7为止，标准规定此位仅适用于UDMA传输，但ATA / ATAPI-8草案修订版1f表示该位可能适用于多字DMA和PIO。

**在数据传输或完成期间的ABRT错误**

到ATA / ATAPI-7为止，标准规定ABRT可以在ICRC错误以及设备无法完成命令的情况下设置。结合MWDMA和PIO传输错误在ATA / ATAPI-7之前不允许使用ICRC位的事实，似乎暗示ABRT位本身可能指示传输错误。

但是，ATA / ATAPI-8草案修订版1f删除了ICRC错误可能打开ABRT的部分。因此，这是某种灰色地带。这里需要一些启发式方法。

ATA / ATAPI设备错误可以进一步分类如下。

**介质错误**

这由ERROR寄存器中的UNC位指示。 ATA设备仅在一定数量的重试无法恢复数据时报告UNC错误，因此除了通知上层之外，没有其他事情可做。

READ和WRITE命令报告第一个失败扇区的CHS或LBA，但ATA / ATAPI标准规定错误完成时传输的数据量是不确定的，因此不能假设失败扇区之前的扇区已传输，因此不能像SCSI那样成功地完成那些扇区。

**介质更改/请求介质更改错误**

<<待办事项：在此处填写>>

**地址错误**

这由ERROR寄存器中的IDNF位指示。报告给上层。

**其他错误**

这可能是由ABRT ERROR位或其他错误条件指示的无效命令或参数。请注意，ABRT位可以指示许多内容，包括ICRC和地址错误。需要启发式方法。

根据命令，不是所有STATUS / ERROR位都适用。这些不适用的位在输出说明中用“na”标记，但是在ATA / ATAPI-7之前找不到“na”的定义。但是，ATA / ATAPI-8草案修订版1f将“N / A”说明为：

"N/A"表示一个字段在此标准中没有定义的值，主机或设备不应检查它。N/A字段应清零。

因此，可以合理地假设设备将"na"位清零，因此不需要明确屏蔽它们。

#### ATAPI设备CHECK CONDITION

在传输PACKET命令的最后一个字节之后，在状态寄存器中设置CHK位（ERR位）表示ATAPI设备CHECK CONDITION错误。对于这种类型的错误，需要获取感知数据以收集关于错误的信息。应使用REQUEST SENSE命令获取感知数据。

一旦获取了感知数据，可以像处理其他SCSI错误一样处理这种类型的错误。请注意，感知数据可能会指示ATA总线错误（例如，感知键值04h HARDWARE ERROR&&ASC/ASCQ 47h/00h SCSI PARITY ERROR）。在这种情况下，应将错误视为ATA总线错误，并根据ATA总线错误处理。

#### ATA设备错误（NCQ）

在NCQ命令阶段（一个或多个NCQ命令待处理）内，清除BSY并设置ERR位表示NCQ命令错误。虽然STATUS和ERROR寄存器将包含说明错误的有效值，但需要使用READ LOG EXT以清除错误状态，确定哪个命令失败以及获取更多信息。

读取LOG EXT日志页面10h报告已失败的标记和说明错误的任务文件寄存器值。有了这些信息，可以像处理ATA/ATAPI设备错误（非NCQ/非CHECK CONDITION）中的普通ATA命令错误一样处理失败的命令，并必须重试所有其他正在进行的命令。请注意，不应计算此重试-如果没有失败的命令，以这种方式重试的命令很可能已正常完成。

请注意，ATA总线错误可能会作为ATA设备NCQ错误报告。应按照ATA总线错误中说明的方式处理此问题。

如果读取LOG EXT日志页面10h失败或报告了NQ，则我们非常糟糕。应根据HSM违规进行处理。

#### ATA总线错误

ATA总线错误表示在ATA总线（SATA或PATA）上传输期间发生了数据损坏。此类错误可以由以下指示：

根据ATA/ATAPI设备错误（非NCQ/非CHECK CONDITION）中说明的ICRC或ABRT错误指示。

具有指示传输错误的错误信息的控制器特定错误完成。

在某些控制器上，命令超时。在这种情况下，可能有一种机制来确定超时是否由传输错误引起。

未知/随机错误、超时和各种奇思妙想。

如上所述，传输错误可能会导致各种症状，从设备ICRC错误到随机设备死机。对于许多情况，没有办法确定错误条件是否由传输错误引起；因此，在处理错误和超时时，有必要采用某种启发式方法。例如，对于已知支持指令的重复ABRT错误，很可能会表示ATA总线错误。

确定可能发生ATA总线错误后，降低ATA总线传输速度是可以缓解问题的操作之一。有关更多信息，请参见重新配置传输。

#### PCI总线错误

在PCI（或其他系统总线）上传输期间发生数据损坏或其他故障。对于标准BMDMA，这由BMDMA状态寄存器中的错误位表示。此类错误必须记录下来，因为它表明系统出现了重大问题。建议重置主机控制器。

#### 延迟完成

这当超时发生时，超时处理程序发现已成功地或带有错误地完成了已超时的命令时发生。这通常是由于中断丢失引起的。必须记录此类错误。建议重置主机控制器。

#### 未知错误（超时）

当超时发生并且命令仍在处理中或主机和设备处于未知状态时，就会发生这种情况。当出现这种情况时，HSM可能处于任何有效或无效状态。为了使设备处于已知状态并使其忘记已超时的命令，需要进行重置。可以重试已超时的命令。

超时也可能是由于传输错误引起的。有关详细信息，请参见ATA总线错误。

#### 热插拔和电源管理异常

<<待办事项：在此处填写>>

### EH恢复操作

此部分讨论了几个重要的恢复操作。

#### 清除错误条件

许多控制器要求其错误寄存器由错误处理程序清除。不同的控制器可能具有不同的要求。

对于SATA，强烈建议在错误处理期间至少清除SError寄存器。

#### 重置

在EH期间，以下情况需要进行重置。

HSM处于未知或无效状态

HBA处于未知或无效状态

EH需要让HBA/设备忘记关于在飞行中的命令

HBA/设备的行为异常

不管是否存在错误情况，在EH期间进行重置可能是一种改善EH健壮性的好主意。重置HBA和设备中的哪一个或哪两个，取决于情况，但建议使用以下方案。

当HBA处于就绪状态，但ATA/ATAPI设备处于未知状态时，只重置设备。

如果HBA处于未知状态，则重置HBA和设备。

HBA重置是实现特定的。对于符合任务文件/BMDMA PCI IDE的控制器而言，停止活动的DMA事务可能足以满足，前提是BMDMA状态是唯一的HBA上下文。但即使大多数任务文件/BMDMA PCI IDE符合的控制器可能也有实现特定的要求和机制来重置自己。这必须由特定的驱动程序解决。

另一方面，ATA/ATAPI标准详细说明了重置ATA/ATAPI设备的方法。

**PATA硬件重置**

这是通过断言PATA RESET-信号发出的硬件启动设备重置。虽然有些硬件提供允许驱动程序直接调整RESET-信号的寄存器，但没有标准的软件启动硬件重置的方式。

**软件重置**

这是通过将CONTROL SRST位打开至少5us来实现的。PATA和SATA都支持它，但在SATA的情况下，这可能需要控制器特定的支持，因为在BSY位仍然设置时应该传输第二个寄存器FIS以清除SRST。请注意，对于PATA，这会重置一个通道上的主设备和从设备。

**执行设备诊断命令**

尽管ATA/ATAPI标准没有明确定义，但EDD意味着某种级别的重置，可能与软件重置类似。主机侧EDD协议可以通过正常的命令处理来处理，大多数SATA控制器应该能够像处理其他命令一样处理EDD。与软件重置一样，在PATA总线上影响两个设备的EDD也会影响到。

尽管EDD确实会重置设备，但作为错误处理，这并不适用，因为无法在BSY设置时发出EDD，并且当设备处于未知/异常状态时，它将如何起作用尚不清楚。

**ATAPI 设备重置命令**

这与软件重置非常相似，只是重置可以限制在选定的设备上，而不影响共享电缆的其他设备。

**SATA物理重置**

这是重置SATA设备的首选方法。实际上，它与PATA硬件重置相同。请注意，这可以使用标准SCR控制寄存器完成。因此，它通常比软件重置更容易实现。

重置设备时要考虑的另一件事是，重置会清除某些配置参数，并且需要将它们设置为先前或新调整的值。

受影响的参数包括：

使用INITIALIZE DEVICE PARAMETERS设置的CHS

使用SET FEATURES设置的参数，包括传输模式设置

使用SET MULTIPLE MODE设置的块计数

其他参数（SET MAX、MEDIA LOCK...）

ATA/ATAPI标准指定某些参数必须在硬件或软件重置时保持不变，但并没有严格指定所有参数。始终需要在重置后重新配置所需的参数以保证鲁棒性。请注意，这也适用于从深度睡眠（断电）恢复。

此外，ATA/ATAPI标准要求在更新任何配置参数或进行硬件重置后发布IDENTIFY DEVICE/IDENTIFY PACKET DEVICE，并使用结果进行进一步操作。操作系统驱动程序需要实现重新验证机制以支持此功能。

#### 重新配置传输

对于PATA和SATA，许多便宜的连接器、电缆或控制器都采用了砍掉成本的方法，很常见的情况是出现高传输错误率。可以通过降低传输速度来减轻这种情况。

以下是Jeff Garzik建议的可能方案。

如果在15分钟内出现超过$N（3？）个传输错误，

如果是SATA，则降低SATA PHY速度。如果速度无法降低，

降低UDMA传输速度。如果在UDMA0，切换到PIO4，

降低PIO传输速度。如果在PIO3，投诉，但继续

## ata\_piix内部结构

### int ich\_pata\_cable\_detect(struct ata\_port \* ap)

探测主机控制器电缆检测信息

**参数**

struct ata\_port \* ap

需要电缆检测信息的端口

**说明**

从ATA PCI设备的PCI配置寄存器中读取80c电缆指示器。该寄存器通常由固件（BIOS）设置。

锁定：无（继承自调用方）。

### int piix\_pata\_prereset(struct ata\_link \* link，unsigned long deadline)

针对PATA主机控制器的prereset

**参数**

struct ata\_link \* link

目标链接

unsigned long deadline

**说明**

操作期限jiffies

锁定：无（继承自调用方）。

### void piix\_set\_piomode(struct ata\_port \*ap, struct ata\_device \*adev)

初始化主机控制器PATA PIO时间

**参数**

struct ata\_port \*ap

正在配置时间的端口

struct ata\_device \*adev

需要设置的驱动器

**说明**

在主机控制器PCI配置空间中设置设备的PIO模式。

锁定：无（继承自调用者）。

### void do\_pata\_set\_dmamode(struct ata\_port \*ap, struct ata\_device \*adev, int isich)

初始化主机控制器PATA PIO时间

**参数**

struct ata\_port \*ap

正在配置时间的端口

struct ata\_device \*adev

需要设置的驱动器

int isich

如果芯片是ICH设备，则设置

**说明**

在主机控制器PCI配置空间中设置设备的UDMA模式。

锁定：无（继承自调用者）。

### void piix\_set\_dmamode(struct ata\_port \*ap, struct ata\_device \*adev)

初始化主机控制器PATA DMA时间

**参数**

struct ata\_port \*ap

正在配置时间的端口

struct ata\_device \*adev

um

**说明**

在主机控制器PCI配置空间中设置设备的MW/UDMA模式。

锁定：无（继承自调用者）。

### void ich\_set\_dmamode(struct ata\_port \*ap, struct ata\_device \*adev)

初始化主机控制器PATA DMA时间

**参数**

struct ata\_port \*ap

正在配置时间的端口

struct ata\_device \*adev

um

**说明**

在主机控制器PCI配置空间中设置设备的MW/UDMA模式。

锁定：无（继承自调用者）。

### int piix\_check\_450nx\_errata(struct pci\_dev \*ata\_dev)

检查450NX设置的问题

**参数**

struct pci\_dev \*ata\_dev

要检查的PCI设备

**说明**

检查450NX勘误#19和勘误#25是否存在。如果发现它们，返回一个错误代码，以便我们可以关闭DMA。

### int piix\_init\_one(struct pci\_dev \*pdev, const struct pci\_device\_id \*ent)

向内核服务注册PIIX ATA PCI设备

**参数**

struct pci\_dev \*pdev

要注册的PCI设备

const struct pci\_device\_id \*ent

与pdev匹配的piix\_pci\_tbl中的条目

**说明**

从内核PCI层调用。我们探测组合模式（叹息），然后将控制权交给libata，让它完成其余操作。

锁定：从PCI层继承（可能会休眠）。

**返回**

成功时为零，或-ERRNO值。

## sata\_sil内部

### int sil\_set\_mode(struct ata\_link \*link, struct ata\_device \*\*r\_failed)

包装set\_mode函数

**参数**

struct ata\_link \*link

要设置的链接

struct ata\_device \*\*r\_failed

当我们失败时返回的设备

**说明**

将设备设置的libata方法包装，因为设置后我们需要检查结果并做一些配置工作。

### void sil\_dev\_config(struct ata\_device \*dev)

应用设备/主机特定的勘误修复

**参数**

struct ata\_device \*dev

要检查的设备

**说明**

完成IDENTIFY [PACKET] DEVICE步骤后，并且已知存在设备，将调用此函数。我们应用两个特定于Silicon Image的勘误修复：Seagate和Maxtor修复。

对于某些Seagate设备，我们必须将最大扇区数限制在8K以下。

对于某些Maxtor设备，我们不得超出udma5来编程驱动器。

这两个修复非常悲观。一旦我获得有关这些勘误的更多信息，我将创建更详尽的列表，并将修复应用于仅需要它的特定设备/主机/固件。

20040111-受Mod15Write错误影响的Seagate驱动器被列入黑名单。Maxtor诡计在黑名单中，但出于以下原因，我仍然保留原始悲观的解决方案...-似乎少有关于它的信息，从Windows驱动程序中仅获取了一个设备，可能只有一个受到影响。任何更多的信息将不胜感激。-不过，UDMA5几乎没有什么好抱怨的

## 感谢

ATA知识的大部分归功于与Andre Hedrick（www.linux-ide.org）的漫长交谈以及研究ATA和SCSI规范的漫长时间。

感谢Alan Cox指出SATA和SCSI之间的相似之处，并一般上激励我们对libata进行黑客攻击。

libata的设备检测方法ata\_pio\_devchk以及一般上所有的早期探测都是基于仔细研究Hale Landis在他的ATADRVR驱动程序（www.ata-atapi.com）中的探测/重置代码。

# 目标和iSCSI接口指南

## 介绍和概述

待定

## 目标核心设备接口

## 目标核心传输接口

### int transport\_init\_session(struct se\_session \*se\_sess)

初始化会话对象

**参数**

struct se\_session \*se\_sess

会话对象指针。

**说明**

在调用此函数之前，调用方必须将se\_sess初始化为零。

### struct se\_session \*transport\_alloc\_session(enum target\_prot\_op sup\_prot\_ops)

分配会话对象并将其初始化

**参数**

enum target\_prot\_op sup\_prot\_ops

定义支持的T10-PI模式的位掩码。

### int transport\_alloc\_session\_tags(struct se\_session \*se\_sess, unsigned int tag\_num, unsigned int tag\_size)

分配目标驱动程序私有数据

**参数**

struct se\_session \*se\_sess

会话指针。

unsigned int tag\_num

发起方和目标之间最大的在飞命令数。

unsigned int tag\_size

目标驱动程序与每个命令关联的私有数据的大小（以字节为单位）。

### int target\_submit\_cmd\_map\_sgls(struct se\_cmd \*se\_cmd, struct se\_session \*se\_sess, unsigned char \*cdb, unsigned char \*sense, u64 unpacked\_lun, u32 data\_length, int task\_attr, int data\_dir, int flags, struct scatterlist \*sgl, u32 sgl\_count, struct scatterlist \*sgl\_bidi, u32 sgl\_bidi\_count, struct scatterlist \*sgl\_prot, u32 sgl\_prot\_count)

查找解包后的LUN并提交未初始化的se\_cmd + 使用预分配的SGL内存。

**参数**

struct se\_cmd\* se\_cmd

要提交的命令说明符

struct se\_session\* se\_sess

关联的se\_sess端点

unsigned char\* cdb

指向SCSI CDB的指针

unsigned char\* sense

指向SCSI sense缓冲区的指针

u64 unpacked\_lun

引用struct se\_lun的解包后的LUN

u32 data\_length

预期的数据传输长度

int task\_attr

SAM任务属性

int data\_dir

DMA数据方向

int flags

来自target\_sc\_flags\_tables的命令提交标志

struct scatterlist\* sgl

单向映像的结构scatterlist内存

u32 sgl\_count

单向映像的scatterlist计数

struct scatterlist\* sgl\_bidi

双向读取映像的结构scatterlist内存

u32 sgl\_bidi\_count

双向读取映像的scatterlist计数

struct scatterlist\* sgl\_prot

结构scatterlist内存保护信息

u32 sgl\_prot\_count

保护信息的scatterlist计数

**说明**

如果调用者设置了se\_cmd->tag，则支持任务标记。

返回非零以发出活动I/O关闭失败的信号。所有其他设置异常将被返回为SCSI CHECK\_CONDITION响应，但在此处仍返回零。

这只能从进程上下文中调用，并且目前假定由目标核心对传输载荷缓冲区进行内部分配。

### int target\_submit\_cmd（struct se\_cmd \* se\_cmd，struct se\_session \* se\_sess，unsigned char \* cdb，unsigned char \* sense，u64 unpacked\_lun，u32 data\_length，int task\_attr，int data\_dir，int flags）

查找解包后的lun并提交未初始化的se\_cmd

**参数**

struct se\_cmd\* se\_cmd

要提交的命令说明符

struct se\_session\* se\_sess

关联的se\_sess端点

unsigned char\* cdb

指向SCSI CDB的指针

unsigned char\* sense

指向SCSI sense缓冲区的指针

u64 unpacked\_lun

引用struct se\_lun的解包后的LUN

u32 data\_length

预期的数据传输长度

int task\_attr

SAM任务属性

int data\_dir

DMA数据方向

int flags

来自target\_sc\_flags\_tables的命令提交标志

**说明**

如果调用者设置了se\_cmd->tag，则支持任务标记。

返回非零以发出活动I/O关闭失败的信号。所有其他设置异常将被返回为SCSI CHECK\_CONDITION响应，但在此处仍返回零。

这只能从进程上下文中调用，并且目前假定由目标核心对传输载荷缓冲区进行内部分配。

它还假定内部目标核心SGL内存分配。

### int target\_submit\_tmr（struct se\_cmd \* se\_cmd，struct se\_session \* se\_sess，unsigned char \* sense，u64 unpacked\_lun，void \* fabric\_tmr\_ptr，unsigned char tm\_type，gfp\_t gfp，u64 tag，int flags）

查找解包后的LUN并提交未初始化的se\_cmd以获取TMR CDB

**参数**

struct se\_cmd\* se\_cmd

要提交的命令**说明**符

struct se\_session\* se\_sess

关联的se\_sess端点

unsigned char\* sense

指向SCSI sense缓冲区的指针

u64 unpacked\_lun

引用struct se\_lun的解包后的LUN

void\* fabric\_tmr\_ptr

TMR req的fabric上下文

unsigned char tm\_type

TM请求类型

gfp\_t gfp

调用者的gfp类型

u64 tag

引用的任务标记以进行TMR\_ABORT\_TASK

int flags

提交cmd标志

**说明**

从所有上下文中调用。

### int target\_get\_sess\_cmd（struct se\_cmd \* se\_cmd，bool ack\_kref）

将命令添加到活动->sess\_cmd\_list

**参数**

struct se\_cmd\* se\_cmd

要添加的命令说明符

bool ack\_kref

发出信号，表明fabric将执行一个ack target\_put\_sess\_cmd（）

### int target\_put\_sess\_cmd（struct se\_cmd \* se\_cmd）

减少命令引用计数

**参数**

struct se\_cmd\* se\_cmd

要从中降低参考的命令

**说明**

仅当此target\_put\_sess\_cmd（）调用将引用计数降至零时，才返回1。否则返回零。

### void target\_sess\_cmd\_list\_set\_waiting（struct se\_session \* se\_sess）

设置sess\_tearing\_down，因此不会排队新命令。

**参数**

struct se\_session\* se\_sess

要标记的会话

### void target\_wait\_for\_sess\_cmds（struct se\_session \* se\_sess）

等待未完成的命令

**参数**

struct se\_session\* se\_sess

等待活动I/O的会话

### bool transport\_wait\_for\_tasks（struct se\_cmd \* cmd）

设置CMD\_T\_STOP并等待t\_transport\_stop\_comp

**参数**

struct se\_cmd\* cmd

要等待的命令

## 目标支持的用户空间I / O

### 用户空间I / O

为LIO定义共享内存接口，以传递SCSI命令和数据到用户空间进行处理。这是为了允许过于复杂以无法在内核中支持的后端成为可能。

它使用UIO框架为我们执行大量设备创建和自省工作。

有关环形的布局方式，请参见.h文件。请注意，虽然定义了命令环，但数据区域的细节尚未确定。命令条目中的偏移值指向映射到mmap的区域内部的其他位置。在命令环外部还有单独的空间用于数据缓冲区。这样可以最大限度地灵活地移动缓冲区分配，甚至可以进行页面翻转或其他分配技术，而无需更改命令环布局。

安全性：必须假定用户进程是恶意的。如果它愿意，没有办法阻止它破坏命令环协议，但为了防止其他问题，我们必须始终只从共享内存区域中读取数据，而不会读取偏移量或大小。这适用于命令环条目以及邮箱。所需的附加代码可能具有“UAM”注释。

### 环设计

mmaped区域分为三个部分：1）邮箱（结构tcmu\_mailbox，下文）；2）命令环；3）命令环之外的一切（数据）。

邮箱告诉用户空间命令环距共享内存区域开始的偏移量以及命令环的大小。

内核通过向该环中放置tcmu\_cmd\_entry结构、更新mailbox->cmd\_head和通过UIO的中断机制触发用户空间来将SCSI命令传递给用户空间。

tcmu\_cmd\_entry包含一个头。如果头类型是填充（PAD），则用户空间应跳过hdr->长度字节（mod cmdr\_size）以找到下一个cmd\_entry。否则，该条目将包含指向包含cdb和数据缓冲区的mmaped区域中的偏移量 - 后者可以通过iov数组访问。iov地址也是共享区域中的偏移量。当用户空间完成处理命令时，设置entry->rsp.scsi\_status、如果适用填写rsp.sense\_buffer，还将mailbox->cmd\_tail设置为旧的cmd\_tail加上hdr->length，mod cmdr\_size。如果cmd\_tail不等于cmd\_head，则应以同样的方式处理下一个数据包，以此类推。

## iSCSI辅助函数

### void iscsi\_prep\_data\_out\_pdu(struct iscsi\_task \*task, struct iscsi\_r2t\_info \*r2t, struct iscsi\_data \*hdr)

初始化数据输出

**参数**

struct iscsi\_task \*task

SCSI命令任务

struct iscsi\_r2t\_info \*r2t

R2T信息

struct iscsi\_data \*hdr

PDU中的iSCSI数据

**备注**

初始化此R2T序列中的Data-Out，并找到此SCSI命令中的适当的data\_offset。

此函数在连接锁定的情况下调用。

### void iscsi\_complete\_scsi\_task(struct iscsi\_task \*task, uint32\_t exp\_cmdsn, uint32\_t max\_cmdsn)

正常完成SCSI任务

**参数**

struct iscsi\_task \*task

iscsi任务用于SCSI命令

uint32\_t exp\_cmdsn

预期的cmd sn（CPU格式）

uint32\_t max\_cmdsn

最大的cmd sn（CPU格式）

**说明**

当驱动程序不需要或无法执行较低级别的pdu处理时使用此函数。

使用session back\_lock调用

### struct iscsi\_task \*iscsi\_itt\_to\_task(struct iscsi\_conn \*conn, itt\_t itt)

按itt查找任务

**参数**

struct iscsi\_conn \*conn

iSCSI连接

itt\_t itt

itt

**说明**

这应该用于像登录和NOP这样的管理任务，或者LDD的itt空间不包括会话年龄时。

必须持有会话 back\_lock。

### int \_\_iscsi\_complete\_pdu(struct iscsi\_conn \*conn, struct iscsi\_hdr \*hdr, char \*data, int datalen)

完整的pdu

**参数**

struct iscsi\_conn \*conn

iSCSI连接

struct iscsi\_hdr \*hdr

iSCSI头

char \*data

数据缓冲区

int datalen

数据缓冲区的长度

**说明**

通过释放在queuecommand或send generic时分配的任何资源完成pdu处理。会话back\_lock必须被持有且必须验证已调用itt。

### struct iscsi\_task \*iscsi\_itt\_to\_ctask(struct iscsi\_conn \*conn, itt\_t itt)

按itt查找任务

**参数**

struct iscsi\_conn \*conn

iSCSI连接

itt\_t itt

itt

**说明**

这应该用于cmd任务。

必须持有session back\_lock。

### void iscsi\_requeue\_task(struct iscsi\_task \*task)

重新将任务排队以从会话工作队列运行

**参数**

struct iscsi\_task \*task

重新排队的任务

**说明**

调用者必须对将重新排队的任务引用。

### void iscsi\_suspend\_queue(struct iscsi\_conn \*conn)

挂起iscsi\_queue命令

**参数**

struct iscsi\_conn \*conn

要停止在其上排队IO的iSCSI连接

**说明**

它抓住session frwd\_lock，以确保没有人在xmit\_task/queuecommand中，并设置挂起以防止排队新命令。这仅需要由需要将路径与iscsi\_queuecommand/xmit\_task同步的卸载驱动程序调用，例如ep断开连接。要再次启动IO，则libiscsi将在FFP中调用iscsi\_start\_tx和iscsi\_unblock\_session。

### void iscsi\_suspend\_tx(struct iscsi\_conn \*conn)

暂停iscsi\_data\_xmit

**参数**

struct iscsi\_conn \*conn

要在其上停止处理IO的iSCSI连接。

**说明**

此函数将挂起位设置为防止iscsi\_data\_xmit发送新IO，如果xmit线程上有工作，则会等待其完成。

### int iscsi\_eh\_session\_reset(struct scsi\_cmnd \* sc)

放弃会话并尝试重新登录

**参数**

struct scsi\_cmnd \* sc

SCSI命令

**说明**

此函数将等待重新登录、来自用户空间的会话终止或恢复/替换超时。

### int iscsi\_eh\_recover\_target(struct scsi\_cmnd \* sc)

重置目标，可能还包括会话

**参数**

struct scsi\_cmnd \* sc

SCSI命令

**说明**

这将尝试发送一个温暖的目标重置。如果失败，我们将升级到ERL0会话恢复。

### int iscsi\_host\_add(struct Scsi\_Host \* shost, struct device \* pdev)

将主机添加到系统

**参数**

struct Scsi\_Host \* shost

SCSI主机

struct device \* pdev

父设备

**说明**

这应该由部分卸载和软件iSCSI驱动程序调用，以将主机添加到系统中。

### struct Scsi\_Host \* iscsi\_host\_alloc(struct scsi\_host\_template \* sht, int dd\_data\_size, bool xmit\_can\_sleep)

分配主机和驱动程序数据

**参数**

struct scsi\_host\_template \* sht

SCSI主机模板

int dd\_data\_size

驱动程序主机数据大小

bool xmit\_can\_sleep

表示LLD是否将从工作队列排队IO

**说明**

这应该被称为部分卸载和软件iscsi驱动程序。使用iscsi\_host\_priv（）宏访问驱动程序特定的内存。

### void iscsi\_host\_remove（struct Scsi\_Host \* shost）

删除主机和会话

**参数**

struct Scsi\_Host \* shost

scsi主机

**说明**

如果还有剩余的会话，这将启动删除并等待完成。

### struct iscsi\_cls\_session \* iscsi\_session\_setup（struct iscsi\_transport \* iscsit，struct Scsi\_Host \* shost，uint16\_t cmds\_max，int dd\_size，int cmd\_task\_size，uint32\_t initial\_cmdsn，unsigned int id）

创建iscsi cls会话、主机和会话

**参数**

struct iscsi\_transport \* iscsit

iscsi传输模板

struct Scsi\_Host \* shost

scsi主机

uint16\_t cmds\_max

会话可以排队

int dd\_size

私有驱动程序数据大小，添加到会话分配大小

int cmd\_task\_size

LLD任务私有数据大小

uint32\_t initial\_cmdsn

初始CmdSN

unsigned int id

要添加到此会话的目标ID

**说明**

这可以被软件iscsi\_transports使用，每个scsi主机分配一个会话。

调用者应将cmds\_max设置为他们支持的最大任务总数（mgmt + scsi）。iscsi层为nop处理和登录/注销请求保留iscsi\_mgmt\_cmds\_max任务。

### void iscsi\_session\_teardown（struct iscsi\_cls\_session \* cls\_session）

销毁会话、主机和cls\_session

**参数**

struct iscsi\_cls\_session \* cls\_session

iscsi会话

### struct iscsi\_cls\_conn \* iscsi\_conn\_setup（struct iscsi\_cls\_session \* cls\_session，int dd\_size，uint32\_t conn\_idx）

创建iscsi\_cls\_conn和iscsi\_conn

**参数**

struct iscsi\_cls\_session \* cls\_session

iscsi\_cls会话

int dd\_size

私有驱动程序数据大小

uint32\_t conn\_idx

cid

### void iscsi\_conn\_teardown（struct iscsi\_cls\_conn \* cls\_conn）

拆除iscsi连接

**参数**

struct iscsi\_cls\_conn \* cls\_conn

iscsi类连接

**说明**

TODO：我们可能需要将其变成类似于scsi-mls remove +put host的两个步骤。

## iSCSI引导信息

### struct iscsi\_boot\_kobj \*iscsi\_boot\_create\_target（struct iscsi\_boot\_kset \* boot\_kset，int index，void \* data，ssize\_t（\* show）（void \* data，int type，char \* buf），umode\_t（\* is\_visible）（void \* data，int type），void（\* release）（void \* data））

创建引导目标sysfs目录

**参数**

struct iscsi\_boot\_kset \* boot\_kset

引导kset

int index

目标ID

void \* data

目标特定数据

ssize\_t（\* show）（void \* data，int type，char \* buf）

attr show函数

umode\_t（\* is\_visible）（void \* data，int type）

attr可见性函数

void（\* release）（void \* data）

释放函数

**注**

当所有对目标kobject的引用都被释放时，引导sysfs lib将释放为调用者传递的数据。

### struct iscsi\_boot\_kobj \*iscsi\_boot\_create\_initiator（struct iscsi\_boot\_kset \* boot\_kset，int index，void \* data，ssize\_t（\* show）（void \* data，int type，char \* buf），umode\_t（\* is\_visible）（void \* data，int type），void（\* release）（void \* data））

创建引导初始化程序sysfs目录

**参数**

struct iscsi\_boot\_kset \* boot\_kset

引导kset

int index

发起者ID

void \* data

驱动程序特定数据

ssize\_t（\* show）（void \* data，int type，char \* buf）

attr show函数

umode\_t（\* is\_visible）（void \* data，int type）

attr可见性函数

void（\* release）（void \* data）

释放函数

注

当所有对初始化程序kobject的引用都被释放时，引导sysfs lib将释放为调用者传递的数据。

### struct iscsi\_boot\_kobj \*iscsi\_boot\_create\_ethernet（struct iscsi\_boot\_kset \* boot\_kset，int index，void \* data，ssize\_t（\* show）（void \* data，int type，char \* buf），umode\_t（\* is\_visible）（void \* data，int type），void（\* release）（void \* data））

创建引导以太网sysfs目录

**参数**

struct iscsi\_boot\_kset \* boot\_kset

引导kset

int index

以太网设备ID

void \* data

驱动程序特定数据

ssize\_t（\* show）（void \* data，int type，char \* buf）

attr show函数

umode\_t（\* is\_visible）（void \* data，int type）

attr可见性函数

void（\* release）（void \* data）

释放函数

**注**

当所有对以太网kobject的引用都被释放时，引导sysfs lib将释放为调用者传递的数据。

### struct iscsi\_boot\_kobj \*iscsi\_boot\_create\_acpitbl（struct iscsi\_boot\_kset \* boot\_kset，int index，void \* data，ssize\_t（\* show）（void \* data，int type，char \* buf），umode\_t（\* is\_visible）（void \* data，int type），void（\* release）（void \* data））

创建引导acpi表sysfs目录

**参数**

struct iscsi\_boot\_kset \* boot\_kset

引导kset

int index

未使用

void \* data

驱动程序特定数据

ssize\_t（\* show）（void \* data，int type，char \* buf）

attr show函数

umode\_t（\* is\_visible）（void \* data，int type）

attr可见性函数

void（\* release）（void \* data）

释放函数

**注**

当所有对acpitbl kobject的引用都被释放时，引导sysfs lib将释放为调用者传递的数据。

### struct iscsi\_boot\_kset \*iscsi\_boot\_create\_kset（const char \* set\_name）

创建根sysfs树

**参数**

const char \* set\_name

根目录名

### struct iscsi\_boot\_kset \*iscsi\_boot\_create\_host\_kset（unsigned int hostno）

为scsi主机创建根sysfs树

**参数**

unsigned int hostno

scsi主机的主机号

### void iscsi\_boot\_destroy\_kset（struct iscsi\_boot\_kset \* boot\_kset）

销毁kset及其下面的kobject

**参数**

struct iscsi\_boot\_kset \* boot\_kset

引导kset

**说明**

这将删除kset和kobject及其下面的属性。

## iSCSI传输类

文件drivers/scsi/scsi\_transport\_iscsi.c定义了iSCSI类的传输属性，用于在TCP/IP连接上发送SCSI数据包。

### struct iscsi\_bus\_flash\_session \* iscsi\_create\_flashnode\_sess（struct Scsi\_Host \* shost，int index，struct iscsi\_transport \* transport，int dd\_size）

在sysfs中添加flashnode会话条目。

**参数**

struct Scsi\_Host \* shost

主机数据指针

int index

在sysfs中添加的flashnode索引

struct iscsi\_transport \* transport

传输数据指针

int dd\_size

要分配的总大小

**说明**

添加flashnode会话属性的sysfs条目

**返回**

成功时分配的flashnode会话指针 失败时返回 NULL

### struct iscsi\_bus\_flash\_conn \* iscsi\_create\_flashnode\_conn(struct Scsi\_Host \* shost, struct iscsi\_bus\_flash\_session \* fnode\_sess, struct iscsi\_transport \* transport, int dd\_size)

在sysfs中添加flashnode连接条目

**参数**

struct Scsi\_Host \* shost

主机数据指针

struct iscsi\_bus\_flash\_session \* fnode\_sess

父级flashnode会话条目指针

struct iscsi\_transport \* transport

传输数据指针

int dd\_size

要分配的总大小

**说明**

添加flashnode连接属性的sysfs条目

**返回**

成功时分配的flashnode连接指针 失败时返回 NULL

### struct device \* iscsi\_find\_flashnode\_sess(struct Scsi\_Host \* shost, void \* data, int (\*fn) (struct device \*dev, void \*data)

查找flashnode会话条目

**参数**

struct Scsi\_Host \* shost

主机数据指针

void \* data

包含要用于比较的值的数据指针

int (\*)(struct device \*dev, void \*data) fn

执行实际比较的函数指针

**说明**

使用传递的函数指针中定义的逻辑来比较数据，找到flashnode会话对象

**返回**

成功时找到的flashnode会话设备对象指针 失败时返回 NULL

### struct device \* iscsi\_find\_flashnode\_conn(struct iscsi\_bus\_flash\_session \* fnode\_sess)

查找flashnode连接条目

**参数**

struct iscsi\_bus\_flash\_session \* fnode\_sess

父级flashnode会话条目指针

**说明**

使用传递的函数指针中定义的逻辑来比较数据，找到flashnode连接对象

**返回**

成功时找到的flashnode连接设备对象指针 失败时返回 NULL

### void iscsi\_destroy\_flashnode\_sess(struct iscsi\_bus\_flash\_session \* fnode\_sess)

删除flashnode会话条目

**参数**

struct iscsi\_bus\_flash\_session \* fnode\_sess

要删除的flashnode会话条目指针

**说明**

从sysfs中删除flashnode会话条目以及所有子flashnode连接条目

### void iscsi\_destroy\_all\_flashnode(struct Scsi\_Host \* shost)

删除所有flashnode会话条目

**参数**

struct Scsi\_Host \* shost

主机数据指针

**说明**

从sysfs中删除所有flashnode会话条目以及相应的子flashnode连接条目

### int iscsi\_scan\_finished(struct Scsi\_Host \* shost, unsigned long time)

帮助报告运行扫描的完成时间

**参数**

struct Scsi\_Host \* shost

SCSI主机

unsigned long time

扫描运行时长

**说明**

此函数可用于驱动程序，如qla4xxx，报告扫描启动时完成的时间。

### int iscsi\_block\_scsi\_eh(struct scsi\_cmnd \* cmd)

阻止SCSI eh直到会话状态转换完毕

**参数**

struct scsi\_cmnd \* cmd

传递给SCSI eh处理器的SCSI命令

**说明**

如果会话已关闭，此函数将等待恢复计时器触发或会话重新登录。如果恢复计时器触发，则返回FAST\_IO\_FAIL。调用者应将此错误值传递给SCSI eh。

### void iscsi\_unblock\_session(struct iscsi\_cls\_session \* session)

将会话设置为已登录并开始IO

**参数**

struct iscsi\_cls\_session \* session

iSCSI会话

**说明**

标记会话已准备好接受IO。

### struct iscsi\_cls\_session \* iscsi\_create\_session(struct Scsi\_Host \* shost, struct iscsi\_transport \* transport, int dd\_size, unsigned int target\_id)

创建iSCSI类会话

**参数**

struct Scsi\_Host \* shost

SCSI主机

struct iscsi\_transport \* transport

iSCSI传输

int dd\_size

私有驱动程序数据大小

unsigned int target\_id

目标ID

**说明**

这可以从LLD或iscsi\_transport调用。

### struct iscsi\_cls\_conn \* iscsi\_create\_conn(struct iscsi\_cls\_session \* session, int dd\_size, uint32\_t cid)

创建iSCSI类连接

**参数**

struct iscsi\_cls\_session \* session

iSCSI类会话

int dd\_size

私有驱动程序数据大小

uint32\_t cid

连接ID

**说明**

这可以从LLD或iscsi\_transport调用。连接是会话的子元，因此cid必须对会话上的所有连接都是唯一的。

由于我们不支持MCS，cid通常为零。在某些情况下，对于软件iSCSI，我们可能会尝试预分配连接结构，在这种情况下可能会有两个连接结构，cid将为非零。

### int iscsi\_destroy\_conn(struct iscsi\_cls\_conn \* conn)

销毁iSCSI类连接

**参数**

struct iscsi\_cls\_conn \* conn

iSCSI类会话

**说明**

这可以从LLD或iscsi\_transport调用。

### int iscsi\_session\_event(struct iscsi\_cls\_session \* session, enum iscsi\_uevent\_e event)

发送会话销毁完成事件

**参数**

struct iscsi\_cls\_session \* session

iSCSI类会话

enum iscsi\_uevent\_e event

事件类型

## iSCSI TCP接口

### int iscsi\_sw\_tcp\_recv(read\_descriptor\_t \*rd\_desc, struct sk\_buff \*skb, unsigned int offset, size\_t len)

以sendfile方式接收TCP

**参数**

read\_descriptor\_t \*rd\_desc

读取**说明**符

struct sk\_buff \*skb

套接字缓冲区

unsigned int offset

skb中的偏移量

size\_t len

skb->len - offset

### int iscsi\_sw\_sk\_state\_check(struct sock \*sk)

检查套接字状态

**参数**

struct sock \*sk

套接字

**说明**

如果套接字处于CLOSE或CLOSE\_WAIT状态，则如果仍有一些数据未处理完毕，我们不应关闭连接。

必须使用sk\_callback\_lock调用。

### void iscsi\_sw\_tcp\_write\_space(struct sock \*sk)

当有更多输出缓冲区空间可用时调用

**参数**

struct sock \*sk

可用于套接字空间的套接字

### int iscsi\_sw\_tcp\_xmit\_segment(struct iscsi\_tcp\_conn \*tcp\_conn, struct iscsi\_segment \*segment)

传输段

**参数**

struct iscsi\_tcp\_conn \*tcp\_conn

iSCSI TCP 连接

struct iscsi\_segment \*segment

要传输的缓冲区

**说明**

此函数将尽量传输网络层接受的缓冲区，并返回传输的字节数。如果启用了 CRC 哈希，则此函数将在传输时计算哈希值。当整个段已传输时，它将检索哈希值并将其发送。

### int iscsi\_sw\_tcp\_xmit(struct iscsi\_conn \*conn)

TCP 传输

**参数**

struct iscsi\_conn \*conn

iSCSI 连接

### int iscsi\_sw\_tcp\_xmit\_qlen(struct iscsi\_conn \*conn)

返回排队等待传输的字节数

**参数**

struct iscsi\_conn \*conn

iSCSI 连接

### int iscsi\_tcp\_segment\_done(struct iscsi\_tcp\_conn \*tcp\_conn, struct iscsi\_segment \*segment, int recv, unsigned copied)

检查段是否完成

**参数**

struct iscsi\_tcp\_conn \*tcp\_conn

iSCSI TCP 连接

struct iscsi\_segment \*segment

要检查的 iSCSI 段

int recv

如果从接收路径中调用，则设置为 1

unsigned copied

复制的字节数

**说明**

检查我们是否已完成接收此段。如果接收缓冲区已满但我们还需要更多数据，则继续移动到散列列表中的下一个条目。如果我们接收的数据量不是 4 的倍数，则我们还将透明地接收填充字节。此函数必须是可重入的。

### void iscsi\_tcp\_hdr\_recv\_prep(struct iscsi\_tcp\_conn \*tcp\_conn)

为 hdr 接收准备段

**参数**

struct iscsi\_tcp\_conn \*tcp\_conn

要准备的 iSCSI 连接

**说明**

此函数始终针对散列参数传递 NULL，因为在调用此函数时，我们尚不知道标头的最终大小，并希望延迟摘要处理直到知道为止。

### void iscsi\_tcp\_cleanup\_task(struct iscsi\_task \*task)

释放 tcp\_task 资源

**参数**

struct iscsi\_task \*task

iSCSI 任务

**说明**

必须使用 session back\_lock 调用

### int iscsi\_tcp\_recv\_segment\_is\_hdr(struct iscsi\_tcp\_conn \*tcp\_conn)

测试是否正在读取标头

**参数**

struct iscsi\_tcp\_conn \*tcp\_conn

iSCSI TCP 连接

**说明**

如果当前正在处理或设置以处理标头，则返回非零。

### int iscsi\_tcp\_recv\_skb(struct iscsi\_conn \*conn, struct sk\_buff \*skb, unsigned int offset, bool offloaded, int \*status)

处理 skb

**参数**

struct iscsi\_conn \*conn

iSCSI 连接

struct sk\_buff \*skb

网络缓冲区，其中包含标头和/或数据段

unsigned int offset

skb 中的偏移量

bool offloaded

表示传输是否已卸载的布尔值

int \*status

iSCSI TCP 状态结果

**说明**

将返回状态传输给 status。并将返回已复制的字节数。

### int iscsi\_tcp\_task\_init(struct iscsi\_task \*task)

初始化 iSCSI SCSI\_READ 或 SCSI\_WRITE 命令

**参数**

struct iscsi\_task \*task

SCSI 命令任务

### int iscsi\_tcp\_task\_xmit(struct iscsi\_task \*task)

传输普通 PDU 任务

**参数**

struct iscsi\_task \*task

iSCSI 命令任务

**说明**

当一切顺利传输时，我们预期返回 0，如果队列中仍有数据，则返回 -EAGAIN，否则返回 != 0 的任何其他错误。

# MTD NAND 驱动程序编程接口

## 介绍

通用 NAND 驱动程序支持几乎所有 NAND 和 AG-AND 基于芯片，将其连接到 Linux 内核的 Memory Technology Devices (MTD) 子系统中。

为希望实现适用于 NAND 设备的板载驱动程序或文件系统驱动程序的开发人员提供此文档。

## 已知的错误和假设

无。

## 文档提示

函数和结构文档是自动生成的。每个函数和结构成员都具有简短的说明，并带有 [XXX] 标识符。以下章节解释了这些标识符的含义。

### 函数标识符 [XXX]

在短注释中，函数用 [XXX] 标识符标记。标识符解释了函数的使用和范围。使用以下标识符：

[MTD Interface]

这些函数提供对 MTD 内核 API 的接口。它们不可替换并提供硬件上完全独立的功能。

[NAND Interface]

这些函数是导出的，并提供对 NAND 内核 API 的接口。

[GENERIC]

通用函数不可替换并提供完全独立于硬件的功能。

[DEFAULT]

默认函数提供适用于大多数实现的硬件相关功能。如果必要，板载驱动程序可以替换这些函数。这些函数通过 NAND 芯片说明结构中的指针调用。在调用 nand\_scan() 之前，板载驱动程序可以设置应由板依赖函数替换的函数。如果函数指针在进入 nand\_scan() 时为 NULL，则指针设置为适用于检测到的芯片类型的默认函数。

### 结构成员标识符 [XXX]

在注释中，结构成员用 [XXX] 标识符标记。标识符解释了成员的使用和范围。使用以下标识符：

[INTERN]

这些成员仅限 NAND 驱动程序内部使用，不得修改。这些值中的大多数根据在 nand\_scan() 中评估的芯片几何信息计算。

[REPLACEABLE]

可替换的成员包含与硬件有关的功能，这些功能可以由板载驱动程序提供。在调用nand\_scan()之前，板载驱动程序可以设置应替换为板依赖功能的功能。如果函数指针在进入nand\_scan()时为NULL，则指针将设置为默认函数，适用于检测到的芯片类型。

[BOARDSPECIFIC]

板特定的成员包含必须由板载驱动程序提供的与硬件有关的信息。板载驱动程序必须在调用nand\_scan()之前设置函数指针和数据字段。

[OPTIONAL]

可选成员可能包含与板载驱动程序有关的信息。通用NAND驱动程序代码不使用此信息。

## 基本板载驱动程序

对于大多数板，只需提供基本功能并在nand芯片说明结构中填写一些真正依赖于板子的成员即可。

### 基本定义

至少需要提供一个nand\_chip结构和一个存储ioremap化的芯片地址的存储器。您可以使用kmalloc分配nand\_chip结构，也可以静态分配它。NAND芯片结构嵌入了一个将注册到MTD子系统中的mtd结构。您可以使用nand\_to\_mtd()助手从nand\_chip指针中提取指向mtd结构的指针。

基于kmalloc的示例

Kmalloc based example

static struct mtd\_info \*board\_mtd;

static void \_\_iomem \*baseaddr;

静态示例

static struct nand\_chip board\_chip;

static void \_\_iomem \*baseaddr;

### 分区定义

如果要将设备分成分区，则定义适合您的板的分区方案。

#define NUM\_PARTITIONS 2

static struct mtd\_partition partition\_info[] = {

{ .name = "Flash partition 1",

.offset = 0,

.size = 8 \* 1024 \* 1024 },

{ .name = "Flash partition 2",

.offset = MTDPART\_OFS\_NEXT,

.size = MTDPART\_SIZ\_FULL },

};

### 硬件控制函数

硬件控制函数提供对NAND芯片的控制引脚的访问。可以通过GPIO引脚或地址线进行访问。如果使用地址线，请确保符合时序要求。

基于GPIO的示例

static void board\_hwcontrol(struct mtd\_info \*mtd, int cmd)

{

switch(cmd){

case NAND\_CTL\_SETCLE: /\* Set CLE pin high \*/ break;

case NAND\_CTL\_CLRCLE: /\* Set CLE pin low \*/ break;

case NAND\_CTL\_SETALE: /\* Set ALE pin high \*/ break;

case NAND\_CTL\_CLRALE: /\* Set ALE pin low \*/ break;

case NAND\_CTL\_SETNCE: /\* Set nCE pin low \*/ break;

case NAND\_CTL\_CLRNCE: /\* Set nCE pin high \*/ break;

}

}

基于地址线的示例。假设nCE引脚由片选解码器驱动.

static void board\_hwcontrol(struct mtd\_info \*mtd, int cmd)

{

struct nand\_chip \*this = mtd\_to\_nand(mtd);

switch(cmd){

case NAND\_CTL\_SETCLE: this->legacy.IO\_ADDR\_W |= CLE\_ADRR\_BIT; break;

case NAND\_CTL\_CLRCLE: this->legacy.IO\_ADDR\_W &= ~CLE\_ADRR\_BIT; break;

case NAND\_CTL\_SETALE: this->legacy.IO\_ADDR\_W |= ALE\_ADRR\_BIT; break;

case NAND\_CTL\_CLRALE: this->legacy.IO\_ADDR\_W &= ~ALE\_ADRR\_BIT; break;

}

}

### 设备准备函数

如果硬件接口将NAND芯片的准备就绪引脚连接到GPIO或其他可访问的I/O引脚，则使用此函数读取引脚的状态。该函数没有参数，如果设备忙（R/B引脚为低电平），则应返回0；如果设备准备就绪（R/B引脚为高电平），则应返回1。如果硬件接口无法访问准备就绪引脚，则不得定义该函数并将函数指针this->legacy.dev\_ready设置为NULL。

### 初始化函数

初始化函数分配内存并设置所有板特定参数和函数指针。当一切都设置好后，调用nand\_scan()。此函数尝试检测和识别芯片。如果找到芯片，则所有内部数据字段都将相应地初始化。struct 必须先清零，然后用关于设备的必要信息填充。

static int \_\_init board\_init (void)

{

struct nand\_chip \*this;

int err = 0;

/\* Allocate memory for MTD device structure and private data \*/

this = kzalloc(sizeof(struct nand\_chip), GFP\_KERNEL);

if (!this) {

printk ("Unable to allocate NAND MTD device structure.\n");

err = -ENOMEM;

goto out;

}

board\_mtd = nand\_to\_mtd(this);

/\* map physical address \*/

baseaddr = ioremap(CHIP\_PHYSICAL\_ADDRESS, 1024);

if (!baseaddr) {

printk("Ioremap to access NAND chip failed\n");

err = -EIO;

goto out\_mtd;

}

/\* Set address of NAND IO lines \*/

this->IO\_ADDR\_R = baseaddr;

this->IO\_ADDR\_W = baseaddr;

/\* Reference hardware control function \*/

this->hwcontrol = board\_hwcontrol;

/\* Set command delay time, see datasheet for correct value \*/

this->chip\_delay = CHIP\_DEPENDEND\_COMMAND\_DELAY;

/\* Assign the device ready function, if available \*/

this->dev\_ready = board\_dev\_ready;

this->eccmode = NAND\_ECC\_SOFT;

/\* Scan to find existence of the device \*/

if (nand\_scan (board\_mtd, 1)) {

err = -ENXIO;

goto out\_ior;

}

add\_mtd\_partitions(board\_mtd, partition\_info, NUM\_PARTITIONS);

goto out;

out\_ior:

iounmap(baseaddr);

out\_mtd:

kfree (this);

out:

return err;

}

module\_init(board\_init);

### 退出函数

如果驱动程序被编译为模块，则仅在此需要退出函数。它释放芯片驱动程序持有的所有资源，并在MTD层中注销分区。

#ifdef MODULE

static void \_\_exit board\_cleanup (void)

{

/\* Unregister device \*/

WARN\_ON(mtd\_device\_unregister(board\_mtd));

/\* Release resources \*/

nand\_cleanup(mtd\_to\_nand(board\_mtd));

/\* unmap physical address \*/

iounmap(baseaddr);

/\* Free the MTD device structure \*/

kfree (mtd\_to\_nand(board\_mtd));

}

module\_exit(board\_cleanup);

#endif

## 高级板级驱动程序功能

本章介绍了NAND驱动程序的高级功能。需要重写的板级驱动程序的函数列表请参阅nand\_chip结构的文档。

### 多芯片控制

NAND驱动程序可以控制芯片阵列。因此，板级驱动程序必须提供自己的select\_chip函数。此功能必须选择（取消选择）要请求的芯片。在调用nand\_scan（）之前，必须设置nand\_chip结构中的函数指针。 nand\_scan（）的maxchip参数定义了要扫描的芯片的最大数量。确保select\_chip函数可以处理请求的芯片数。

NAND驱动程序将芯片连接成一个虚拟芯片，并将此虚拟芯片提供给MTD层。

注意：驱动程序只能处理等大小芯片的线性芯片阵列。不支持扩展总线宽度的并行阵列。

基于GPIO的示例

static void board\_select\_chip (struct mtd\_info \*mtd, int chip)

{

/\* Deselect all chips, set all nCE pins high \*/

GPIO(BOARD\_NAND\_NCE) |= 0xff;

if (chip >= 0)

GPIO(BOARD\_NAND\_NCE) &= ~ (1 << chip);

}

基于地址线的示例。假设nCE引脚连接到地址解码器。

static void board\_select\_chip (struct mtd\_info \*mtd, int chip)

{

struct nand\_chip \*this = mtd\_to\_nand(mtd);

/\* Deselect all chips \*/

this->legacy.IO\_ADDR\_R &= ~BOARD\_NAND\_ADDR\_MASK;

this->legacy.IO\_ADDR\_W &= ~BOARD\_NAND\_ADDR\_MASK;

switch (chip) {

case 0:

this->legacy.IO\_ADDR\_R |= BOARD\_NAND\_ADDR\_CHIP0;

this->legacy.IO\_ADDR\_W |= BOARD\_NAND\_ADDR\_CHIP0;

break;

....

case n:

this->legacy.IO\_ADDR\_R |= BOARD\_NAND\_ADDR\_CHIPn;

this->legacy.IO\_ADDR\_W |= BOARD\_NAND\_ADDR\_CHIPn;

break;

}

}

### 硬件ECC支持

#### 函数和常量

NAND驱动程序支持三种不同类型的硬件ECC。

1）NAND\_ECC\_HW3\_256

提供每256个字节3个字节的ECC的硬件ECC生成器。

2）NAND\_ECC\_HW3\_512

提供每512个字节3个字节的ECC的硬件ECC生成器。

3）NAND\_ECC\_HW6\_512

提供每512个字节6个字节的ECC的硬件ECC生成器。

4）NAND\_ECC\_HW8\_512

提供每512个字节8个字节的ECC的硬件ECC生成器。

如果您的硬件生成器具有不同的功能，请在nand\_base.c的适当位置添加它。

板级驱动程序必须提供以下功能：

1）enable\_hwecc

在读/写芯片之前调用此函数。在此功能中重置或初始化硬件生成器。该函数是使用可让您区分读取和写入操作的参数调用的。

2）calculate\_ecc

在从/写入芯片后调用此函数。将ECC从硬件传输到缓冲区。如果设置了NAND\_HWECC\_SYNDROME选项，则仅在写入时调用该函数。详见下文。

3）correct\_data

在ECC出现错误的情况下调用此函数进行错误检测和更正。如果可以更正错误，则返回1或2。如果错误不可更正，则返回-1。如果您的硬件生成器与nand\_ecc软件生成器的默认算法匹配，则使用nand\_ecc提供的校正函数而不是实现重复代码。

#### 具有综合计算的硬件ECC

许多硬件ECC实现提供Reed-Solomon代码并在读取时计算错误综合。在调用通用Reed-Solomon库中的错误更正代码之前，必须将该综合转换为标准Reed-Solomon综合。

在数据字节之后立即放置ECC字节，以使综合发生器工作。这与软件ECC通常使用的布局相反。不再可以分离数据和带外区域。NAND驱动程序代码处理此布局，而剩余的自由字节在autoplacement代码中管理。在这种情况下提供匹配的oob-layout。有关实现参考，请参见rts\_from4.c和diskonchip.c。在这些情况下，我们还必须在FLASH上使用坏块表，因为ECC布局与坏块标记位置干扰。有关详细信息，请参阅坏块表支持。

#### 坏块表支持

大多数NAND芯片将坏块标记放在空闲区的特定位置。绝不能擦除这些坏块，因为坏块信息将丢失。可以通过读取块中第一页的空闲区域来每次访问块时检查坏块标记。这很耗时，因此使用坏块表。

NAND驱动程序支持各种类型的坏块表。

1）每个设备

坏块表包含设备的所有坏块信息，该设备可能由多个芯片组成。

2）按芯片

每个芯片使用一个坏块表，并包含该特定芯片的坏块信息。

3）固定偏移量

坏块表位于芯片（设备）的固定偏移量上。这适用于各种DiskOnChip设备。

4）自动放置

坏块表自动放置并检测到芯片（设备）的末尾或开头。

5）镜像表

坏块表在芯片（设备）上镜像，允许更新坏块表而不会丢失数据。

nand\_scan（）调用函数nand\_default\_bbt（）。nand\_default\_bbt（）根据nand\_scan（）检索到的芯片信息选择适当的默认坏块表说明符。

标准策略是扫描设备以查找坏块并构建基于RAM的坏块表，这样比始终检查闪存芯片本身的坏块信息更快。

#### 基于闪存的表格

可能希望或需要在闪存中保留坏块表。对于AG-AND芯片，这是强制性的，因为它们没有出厂标记的坏块。它们有出厂标记的好块。这个标记模式在块被擦除以便重新使用时被擦除。因此，如果在将模式写回芯片之前发生断电，该块将丢失并添加到坏块中。因此，当我们第一次检测到它们时扫描芯片以获取好块信息，并将此信息存储在坏块表中，然后才擦除任何块。

存储表格的块通过在内存坏块表中标记它们为坏块来防止意外访问。坏块表管理函数允许规避此保护。

激活FLASH基础坏块表支持的最简单方法是在调用nand\_scan（）之前将选项NAND\_BBT\_USE\_FLASH设置为nand芯片结构的bbt\_option字段中。对于AG-AND芯片，这是默认设置的。这会激活NAND驱动程序的默认FLASH基础坏块表功能。默认坏块表选项包括

每个芯片存储坏块表

每个块使用2位

在芯片末尾自动放置

使用带有版本号的镜像表

在芯片末尾保留4个块

#### 用户定义的表格

通过填充nand\_bbt\_descr结构并将指针存储在nand\_chip结构成员bbt\_td中创建用户定义的表格，然后再调用nand\_scan（）。如果需要镜像表，则必须创建第二个结构，并将指针存储在nand\_chip结构的bbt\_md中。如果将bbt\_md成员设置为NULL，则仅使用主表格，并且不执行对镜像表的扫描。

在nand\_bbt\_descr结构中，最重要的字段是选项字段。选项定义大多数表格属性。使用从rawnand.h预定义的常量来定义选项。

1）每个块的位数

支持的位数为1, 2, 4, 8。

2）每个芯片的表格

设置常量NAND\_BBT\_PERCHIP选择在芯片阵列中为每个芯片管理坏块表。如果未设置此选项，则使用每个设备的坏块表。

3）表位置是绝对的

使用选项常量NAND\_BBT\_ABSPAGE，并在字段pages中定义坏块表开始的绝对页码。如果选择了每个芯片的坏块表，并且您有一个多芯片阵列，则必须为每个芯片提供起始页码。注意：不执行表格识别模式的扫描，因此可以将pattern、veroffs、offs、len字段留空。

4）表位置自动检测

该表可以位于芯片（设备）的第一个或最后一个好块中。将NAND\_BBT\_LASTBLOCK设置为在芯片（设备）的末尾放置坏块表。坏块表因在包含坏块表的块的第一个页中的备用区域中存储的模式而被标记和识别。将指针存储到pattern字段中。此外，还必须在len中存储模式的长度，并在nand\_bbt\_descr结构的offs成员中给出备用区域的偏移量。对于镜像坏块表，必须使用不同的模式。

5）表格创建

设置选项NAND\_BBT\_CREATE以在扫描期间找不到表格时启用表格创建。通常，如果找到新芯片，则只进行一次操作。

6）表格写入支持

设置选项NAND\_BBT\_WRITE以启用表格写入支持。这允许更新坏块表以标记块由于磨损而变坏。MTD接口函数block\_markbad调用坏块表的更新函数。如果启用了写入支持，则表格将在FLASH上更新。

注意：版本控制应仅启用镜像表格，具有写支持。

7）表格版本控制

将选项NAND\_BBT\_VERSION设置为启用表格版本控制。强烈建议为具有写支持的镜像表启用此功能。这可以确保减少丢失损坏块表信息的风险，仅损失应标记为损坏的一个磨损块的信息。版本存储在设备的备用区域中的4个连续字节中。版本号的位置由损坏块表说明符中的成员veroffs定义。

8）写入时保存块内容

如果包含损坏块表的块含有其他有用的信息，请设置选项NAND\_BBT\_SAVECONTENT。将写入损坏块表时，将读取整个块以更新损坏块表，擦除块，然后将一切写回。如果未设置此选项，则仅写入损坏块表，块中的其他所有内容均被忽略并擦除。

9）保留块数

对于自动放置，必须保留一些块以用于损坏块表存储。保留块数在损坏块表说明结构的maxblocks成员中定义。为镜像表格保留4个块应是合理的数量。这还限制了扫描损坏块表标识模式的块数。

### 备用区域（自动）放置

nand驱动程序实现了存储系统数据的备用区域的不同放置可能性，

由fs驱动程序定义的放置

自动放置

默认放置函数是自动放置。nand驱动程序针对各种芯片类型内置默认放置方案。如果由于硬件ECC功能而默认放置不适合，则板驱动程序可以提供自己的放置方案。

文件系统驱动程序可以提供自己的放置方案，以代替默认放置方案。

放置方案由nand\_oobinfo结构定义

struct nand\_oobinfo {

int useecc;

int eccbytes;

int eccpos[24];

int oobfree[8][2];

};

1）useecc

useecc成员控制ecc和放置功能。头文件include / mtd / mtd-abi.h包含选择ecc和放置的常量。 MTD\_NANDECC\_OFF完全关闭ecc。这不是推荐的，仅用于测试和诊断。 MTD\_NANDECC\_PLACE选择主叫方定义的放置，MTD\_NANDECC\_AUTOPLACE选择自动放置。

2）eccbytes

eccbytes成员定义每页ecc字节数。

3）eccpos

eccpos数组保存备用区域中放置ecc代码的字节偏移量。

4）oobfree

oobfree数组定义备用区域中可用于自动放置的区域。信息以{offset，size}格式给出。offset定义可用区域的起始位置，size定义以字节为单位的长度。可以定义超过一个区域。列表由{0,0}条目终止。

#### 由fs驱动程序定义的放置

调用函数提供指向nand\_oobinfo结构的指针，该结构定义了ecc放置。对于写入，调用者必须同时提供备用区域缓冲区和数据缓冲区。备用区域缓冲区大小为（页数）\*（备用区域大小）。对于读取，缓冲区大小为（页数）\*（（备用区域大小）+（每页ecc步骤数）\* sizeof（int））。驱动程序为备用缓冲区中的每个元组存储ecc检查结果。存储顺序为：

<spare data page 0><ecc result 0>...<ecc result n>

...

<spare data page n><ecc result 0>...<ecc result n>

这是YAFFS1使用的旧模式。

如果备用区域缓冲区为NULL，则仅根据nand\_oobinfo结构中给定的方案进行ECC放置。

#### 自动放置

自动放置使用内置默认值在备用区域中放置ecc字节。如果必须将文件系统数据存储/读取到备用区域中，则调用函数必须提供缓冲区。每页的缓冲区大小由nand\_oobinfo结构中的oobfree数组确定。

如果备用区域缓冲区为NULL，则仅根据默认内置方案进行ECC放置。

### 备用区域自动放置默认方案

#### 256字节页面大小

| Offset | Content | Comment |
| --- | --- | --- |
| 0x00 | ECC byte 0 | Error correction code byte 0 |
| 0x01 | ECC byte 1 | Error correction code byte 1 |
| 0x02 | ECC byte 2 | Error correction code byte 2 |
| 0x03 | Autoplace 0 |  |
| 0x04 | Autoplace 1 |  |
| 0x05 | Bad block marker | 如果此字节中的任何位为零，则此块是坏块。这仅适用于块中的第一页。在剩余的页面中，该字节被保留 |
| 0x06 | Autoplace 2 |  |
| 0x07 | Autoplace 3 |  |

#### 512 byte pagesize

| Offset | Content | Comment |
| --- | --- | --- |
| 0x00 | ECC byte 0 | Error correction code byte 0 of the lower 256 Byte data in this page |
| 0x01 | ECC byte 1 | Error correction code byte 1 of the lower 256 Bytes of data in this page |
| 0x02 | ECC byte 2 | Error correction code byte 2 of the lower 256 Bytes of data in this page |
| 0x03 | ECC byte 3 | Error correction code byte 0 of the upper 256 Bytes of data in this page |
| 0x04 | reserved | reserved |
| 0x05 | Bad block marker | If any bit in this byte is zero, then this block is bad. This applies only to the first page in a block. In the remaining pages this byte is reserved |
| 0x06 | ECC byte 4 | Error correction code byte 1 of the upper 256 Bytes of data in this page |
| 0x07 | ECC byte 5 | Error correction code byte 2 of the upper 256 Bytes of data in this page |
| 0x08 - 0x0F | Autoplace 0 - 7 |  |

#### 2048 byte pagesize

| Offset | Content | Comment |
| --- | --- | --- |
| 0x00 | Bad block marker | If any bit in this byte is zero, then this block is bad. This applies only to the first page in a block. In the remaining pages this byte is reserved |
| 0x01 | Reserved | Reserved |
| 0x02-  0x27 | Autoplace 0 - 37 |  |
| 0x28 | ECC byte 0 | Error correction code byte 0 of the first 256 Byte data in this page |
| 0x29 | ECC byte 1 | Error correction code byte 1 of the first 256 Bytes of data in this page |
| 0x2A | ECC byte 2 | Error correction code byte 2 of the first 256 Bytes data in this page |
| 0x2B | ECC byte 3 | Error correction code byte 0 of the second 256 Bytes of data in this page |
| 0x2C | ECC byte 4 | Error correction code byte 1 of the second 256 Bytes of data in this page |
| 0x2D | ECC byte 5 | Error correction code byte 2 of the second 256 Bytes of data in this page |
| 0x2E | ECC byte 6 | Error correction code byte 0 of the third 256 Bytes of data in this page |
| 0x2F | ECC byte 7 | Error correction code byte 1 of the third 256 Bytes of data in this page |
| 0x30 | ECC byte 8 | Error correction code byte 2 of the third 256 Bytes of data in this page |
| 0x31 | ECC byte 9 | Error correction code byte 0 of the fourth 256 Bytes of data in this page |
| 0x32 | ECC byte 10 | Error correction code byte 1 of the fourth 256 Bytes of data in this page |
| 0x33 | ECC byte 11 | Error correction code byte 2 of the fourth 256 Bytes of data in this page |
| 0x34 | ECC byte 12 | Error correction code byte 0 of the fifth 256 Bytes of data in this page |
| 0x35 | ECC byte 13 | Error correction code byte 1 of the fifth 256 Bytes of data in this page |
| 0x36 | ECC byte 14 | Error correction code byte 2 of the fifth 256 Bytes of data in this page |
| 0x37 | ECC byte 15 | Error correction code byte 0 of the sixth 256 Bytes of data in this page |
| 0x38 | ECC byte 16 | Error correction code byte 1 of the sixth 256 Bytes of data in this page |
| 0x39 | ECC byte 17 | Error correction code byte 2 of the sixth 256 Bytes of data in this page |
| 0x3A | ECC byte 18 | Error correction code byte 0 of the seventh 256 Bytes of data in this page |
| 0x3B | ECC byte 19 | Error correction code byte 1 of the seventh 256 Bytes of data in this page |
| 0x3C | ECC byte 20 | Error correction code byte 2 of the seventh 256 Bytes of data in this page |
| 0x3D | ECC byte 21 | Error correction code byte 0 of the eighth 256 Bytes of data in this page |
| 0x3E | ECC byte 22 | Error correction code byte 1 of the eighth 256 Bytes of data in this page |
| 0x3F | ECC byte 23 | Error correction code byte 2 of the eighth 256 Bytes of data in this page |

## 文件系统支持

NAND驱动程序通过MTD接口提供了所有必要的文件系统函数。

文件系统必须了解NAND的特殊限制。 NAND Flash的一个主要限制是，您不能随意写入页面。在再次擦除之前，对一个页面的连续写入将被限制为1-3次，具体取决于制造商的规格。这同样适用于备用区域。

因此，支持NAND的文件系统必须以页面大小的块进行写入，或者持有写缓冲区，以收集小的写入，直至它们总和达到页面大小。可用的NAND感知文件系统有：JFFS2，YAFFS。

存储文件系统数据的备用区使用由备用区放置功能控制，该功能在早期章节中有说明。

## 工具

MTD项目提供了一些有用的工具来处理NAND闪存。

flasherase、flasheraseall：擦除和格式化FLASH分区

nandwrite：将文件系统映像写入NAND FLASH

nanddump：转储一个NAND FLASH分区的内容

这些工具都知道NAND的限制。请使用这些工具，而不是抱怨由非NAND感知访问方法引起的错误。

## 常量

本章说明了对驱动程序开发人员可能相关的常量。

### 芯片选项常量

#### 芯片ID表的常量

这些常量在raw nand.h中被定义。它们组合在一起来说明芯片的功能：

/\* Buswitdh is 16 bit \*/

#define NAND\_BUSWIDTH\_16 0x00000002

/\* Device supports partial programming without padding \*/

#define NAND\_NO\_PADDING 0x00000004

/\* Chip has cache program function \*/

#define NAND\_CACHEPRG 0x00000008

/\* Chip has copy back function \*/

#define NAND\_COPYBACK 0x00000010

/\* AND Chip which has 4 banks and a confusing page / block

\* assignment. See Renesas datasheet for further information \*/

#define NAND\_IS\_AND 0x00000020

/\* Chip has a array of 4 pages which can be read without

\* additional ready /busy waits \*/

#define NAND\_4PAGE\_ARRAY 0x00000040

#### 运行时选项的常量

这些常量在raw nand.h中被定义。它们组合在一起来说明功能：

/\* The hw ecc generator provides a syndrome instead a ecc value on read

\* This can only work if we have the ecc bytes directly behind the

\* data bytes. Applies for DOC and AG-AND Renesas HW Reed Solomon generators \*/

#define NAND\_HWECC\_SYNDROME 0x00020000

### ECC选择常量

使用这些常量选择ECC算法：

/\* No ECC. Usage is not recommended ! \*/

#define NAND\_ECC\_NONE 0

/\* Software ECC 3 byte ECC per 256 Byte data \*/

#define NAND\_ECC\_SOFT 1

/\* Hardware ECC 3 byte ECC per 256 Byte data \*/

#define NAND\_ECC\_HW3\_256 2

/\* Hardware ECC 3 byte ECC per 512 Byte data \*/

#define NAND\_ECC\_HW3\_512 3

/\* Hardware ECC 6 byte ECC per 512 Byte data \*/

#define NAND\_ECC\_HW6\_512 4

/\* Hardware ECC 8 byte ECC per 512 Byte data \*/

#define NAND\_ECC\_HW8\_512 6

### 硬件控制相关的常量

当调用开发板固件控制函数时，这些常量说明了所请求的硬件访问功能：

/\* Select the chip by setting nCE to low \*/

#define NAND\_CTL\_SETNCE 1

/\* Deselect the chip by setting nCE to high \*/

#define NAND\_CTL\_CLRNCE 2

/\* Select the command latch by setting CLE to high \*/

#define NAND\_CTL\_SETCLE 3

/\* Deselect the command latch by setting CLE to low \*/

#define NAND\_CTL\_CLRCLE 4

/\* Select the address latch by setting ALE to high \*/

#define NAND\_CTL\_SETALE 5

/\* Deselect the address latch by setting ALE to low \*/

#define NAND\_CTL\_CLRALE 6

/\* Set write protection by setting WP to high. Not used! \*/

#define NAND\_CTL\_SETWP 7

/\* Clear write protection by setting WP to low. Not used! \*/

#define NAND\_CTL\_CLRWP 8

### 与坏块表有关的常量

这些常量说明了用于坏块表说明符的选项：

/\* Options for the bad block table descriptors \*/

/\* The number of bits used per block in the bbt on the device \*/

#define NAND\_BBT\_NRBITS\_MSK 0x0000000F

#define NAND\_BBT\_1BIT 0x00000001

#define NAND\_BBT\_2BIT 0x00000002

#define NAND\_BBT\_4BIT 0x00000004

#define NAND\_BBT\_8BIT 0x00000008

/\* The bad block table is in the last good block of the device \*/

#define NAND\_BBT\_LASTBLOCK 0x00000010

/\* The bbt is at the given page, else we must scan for the bbt \*/

#define NAND\_BBT\_ABSPAGE 0x00000020

/\* bbt is stored per chip on multichip devices \*/

#define NAND\_BBT\_PERCHIP 0x00000080

/\* bbt has a version counter at offset veroffs \*/

#define NAND\_BBT\_VERSION 0x00000100

/\* Create a bbt if none axists \*/

#define NAND\_BBT\_CREATE 0x00000200

/\* Write bbt if necessary \*/

#define NAND\_BBT\_WRITE 0x00001000

/\* Read and write back block contents when writing bbt \*/

#define NAND\_BBT\_SAVECONTENT 0x00002000

## 结构

本章包含用于NAND驱动程序的结构的自动生成文档，并可能与驱动程序开发人员相关。每个结构成员都有一个短说明，其用[XXX]标识符标记。有关说明，请参阅章节“文档提示”。

### struct onfi\_params

将重新使用的ONFI特定参数。

**定义**

struct onfi\_params {

int version;

u16 tPROG;

u16 tBERS;

u16 tR;

u16 tCCS;

u16 async\_timing\_mode;

u16 vendor\_revision;

u8 vendor[88];

};

**成员**

version

ONFI版本（BCD编码），如果不支持ONFI，则为0

tPROG

页面编程时间

tBERS

块擦除时间

tR

页面读取时间

tCCS

更改列设置时间

async\_timing\_mode

支持的异步计时模式

vendor\_revision

供应商特定的修订号

vendor

供应商特定数据

### struct nand\_parameters

来自参数页面的NAND通用参数.

**定义**

struct nand\_parameters {

const char \*model;

bool supports\_set\_get\_features;

unsigned long set\_feature\_list[BITS\_TO\_LONGS(ONFI\_FEATURE\_NUMBER)];

unsigned long get\_feature\_list[BITS\_TO\_LONGS(ONFI\_FEATURE\_NUMBER)];

struct onfi\_params \*onfi;

};

**成员**

model

模型名称

supports\_set\_get\_features

NAND芯片支持设置/获取功能

set\_feature\_list

可设置的功能位图

get\_feature\_list

可获取的功能位图

onfi

ONFI特定参数

### struct nand\_id

NAND ID结构

**定义**

struct nand\_id {

u8 data[NAND\_MAX\_ID\_LEN];

int len;

};

**成员**

data

包含ID字节的缓冲区。

len

ID长度。

### struct nand\_controller\_ops

控制器操作

**定义**

struct nand\_controller\_ops {

int (\*attach\_chip)(struct nand\_chip \*chip);

void (\*detach\_chip)(struct nand\_chip \*chip);

};

**成员**

attach\_chip

此方法在NAND检测阶段之后调用，之后已设置闪存ID和MTD字段，例如擦除大小、页大小和OOB大小。如果由NAND芯片或设备树提供ECC要求，则可用。通常用于选择适当的ECC配置并分配相关资源。此钩子是可选的。

detach\_chip

释放在nand\_controller\_ops->attach\_chip（）中分配/申报的所有资源。此挂钩是可选的。

### struct nand\_controller

用于说明NAND控制器的结构

**定义**

struct nand\_controller {

spinlock\_t lock;

struct nand\_chip \*active;

wait\_queue\_head\_t wq;

const struct nand\_controller\_ops \*ops;

};

**成员**

lock

保护锁

active

当前持有控制器的MTD设备

wq

等待队列，在进行NAND操作时睡眠。如果有可用的硬件控制器，将使用代替每个芯片等待队列。

ops

NAND控制器操作。

### struct nand\_ecc\_step\_info

ECC引擎的ECC步骤信息

**定义**

struct nand\_ecc\_step\_info {

int stepsize;

const int \*strengths;

int nstrengths;

};

**成员**

stepsize

每个ECC步骤的数据字节数

strengths

支持的强度数组

nstrengths

支持的强度数量

### struct nand\_ecc\_caps

ECC引擎的能力

**定义**

struct nand\_ecc\_caps {

const struct nand\_ecc\_step\_info \*stepinfos;

int nstepinfos;

int (\*calc\_ecc\_bytes)(int step\_size, int strength);

};

**成员**

stepinfos

ECC步骤信息数组

nstepinfos

ECC步骤信息数量

calc\_ecc\_bytes

驱动程序的挂钩，用于计算每个步骤的ECC字节数

### struct nand\_ecc\_ctrl

ECC的控制结构

**定义**

struct nand\_ecc\_ctrl {

enum nand\_ecc\_engine\_type engine\_type;

enum nand\_ecc\_placement placement;

enum nand\_ecc\_algo algo;

int steps;

int size;

int bytes;

int total;

int strength;

int prepad;

int postpad;

unsigned int options;

u8 \*calc\_buf;

u8 \*code\_buf;

void (\*hwctl)(struct nand\_chip \*chip, int mode);

int (\*calculate)(struct nand\_chip \*chip, const uint8\_t \*dat, uint8\_t \*ecc\_code);

int (\*correct)(struct nand\_chip \*chip, uint8\_t \*dat, uint8\_t \*read\_ecc, uint8\_t \*calc\_ecc);

int (\*read\_page\_raw)(struct nand\_chip \*chip, uint8\_t \*buf, int oob\_required, int page);

int (\*write\_page\_raw)(struct nand\_chip \*chip, const uint8\_t \*buf, int oob\_required, int page);

int (\*read\_page)(struct nand\_chip \*chip, uint8\_t \*buf, int oob\_required, int page);

int (\*read\_subpage)(struct nand\_chip \*chip, uint32\_t offs, uint32\_t len, uint8\_t \*buf, int page);

int (\*write\_subpage)(struct nand\_chip \*chip, uint32\_t offset,uint32\_t data\_len, const uint8\_t \*data\_buf, int oob\_required, int page);

int (\*write\_page)(struct nand\_chip \*chip, const uint8\_t \*buf, int oob\_required, int page);

int (\*write\_oob\_raw)(struct nand\_chip \*chip, int page);

int (\*read\_oob\_raw)(struct nand\_chip \*chip, int page);

int (\*read\_oob)(struct nand\_chip \*chip, int page);

int (\*write\_oob)(struct nand\_chip \*chip, int page);

};

**成员**

mode

纠错码模式

algo

ECC算法

steps

每页 ECC 步数

size

每个 ECC 步骤的数据字节数

bytes

每步 ECC 字节数

total

每页的 ECC 字节总数

strength

每个 ECC 步骤的最大可纠正位数

prepad

基于综合征的 ECC 生成器的填充信息

postpad

基于综合征的 ECC 生成器的填充信息

options

ECC 特定选项（参见上面定义的 NAND\_ECC\_XXX 标志）

priv

指向私有 ECC 控制数据的指针

calc\_buf

用于计算 ECC 的缓冲区，大小为 oobsize。

code\_buf

用于从闪存读取 ECC 的缓冲区，大小为 oobsize。

hwctl

功能来控制硬件ECC生成器。仅当硬件 ECC 可用时才必须提供

calculate

ECC 计算或从 ECC 硬件回读的功能

correct

ECC校正功能，匹配ECC生成器（软件/硬件）。应返回一个正数，表示已纠正的位翻转数，如果位翻转数超过 ECC 强度，则返回 -EBADMSG，如果错误与纠正不直接相关，则返回任何其他错误代码。如果返回 -EBADMSG，则输入缓冲区应保持不变。

read\_page\_raw

函数读取没有 ECC 的原始页面。此函数应隐藏 ECC 控制器使用的特定布局，并始终返回连续的带内和带外数据，即使它们没有连续存储在 NAND 芯片上（例如 NAND\_ECC\_HW\_SYNDROME 交错带内和带外）波段数据）。

write\_page\_raw

函数来写一个没有 ECC 的原始页面。此函数应隐藏 ECC 控制器使用的特定布局，并将传递的数据视为连续的带内和带外数据。ECC 控制器负责进行适当的转换以适应其特定布局（例如，NAND\_ECC\_HW\_SYNDROME 交织带内和带外数据）。

read\_page

根据ECC生成器要求读取页面的功能；返回在任何单个 ECC 步骤中纠正的最大位翻转数，-EIO 硬件错误

read\_subpage

读取ECC覆盖的部分页面的功能；返回与read\_page()

write\_subpage

函数写入 ECC 覆盖的页面部分。

write\_page

函数根据 ECC 生成器的要求写入页面。

write\_oob\_raw

无ECC写芯片OOB数据功能

read\_oob\_raw

无ECC读取芯片OOB数据功能

read\_oob

读取芯片OOB数据的函数

write\_oob

写入芯片OOB数据的功能

### struct nand\_sdr\_timings

SDR NAND芯片时序

**定义**

struct nand\_sdr\_timings {

u64 tBERS\_max;

u32 tCCS\_min;

u64 tPROG\_max;

u64 tR\_max;

u32 tALH\_min;

u32 tADL\_min;

u32 tALS\_min;

u32 tAR\_min;

u32 tCEA\_max;

u32 tCEH\_min;

u32 tCH\_min;

u32 tCHZ\_max;

u32 tCLH\_min;

u32 tCLR\_min;

u32 tCLS\_min;

u32 tCOH\_min;

u32 tCS\_min;

u32 tDH\_min;

u32 tDS\_min;

u32 tFEAT\_max;

u32 tIR\_min;

u32 tITC\_max;

u32 tRC\_min;

u32 tREA\_max;

u32 tREH\_min;

u32 tRHOH\_min;

u32 tRHW\_min;

u32 tRHZ\_max;

u32 tRLOH\_min;

u32 tRP\_min;

u32 tRR\_min;

u64 tRST\_max;

u32 tWB\_max;

u32 tWC\_min;

u32 tWH\_min;

u32 tWHR\_min;

u32 tWP\_min;

u32 tWW\_min;

};

**成员**

tBERS\_max

块擦除时间

tCCS\_min

更改列设置时间

tPROG\_max

页编程时间

tR\_max

页面阅读时间

tALH\_min

ALE 保持时间

tADL\_min

ALE 到数据加载时间

tALS\_min

ALE 建立时间

tAR\_min

ALE 到 RE# 延迟

tCEA\_max

CE#访问时间

tCEH\_min

CE# 高保持时间

tCH\_min

CE#保持时间

tCHZ\_max

CE# high 输出 hi-Z

tCLH\_min

CLE保持时间

tCLR\_min

CLE 到 RE# 延迟

tCLS\_min

CLE建立时间

tCOH\_min

CE# 高到输出保持

tCS\_min

CE#建立时间

tDH\_min

数据保持时间

tDS\_min

数据建立时间

tFEAT\_max

Set Features 和 Get Features 的忙碌时间

tIR\_min

输出 hi-Z 到 RE# low

tITC\_max

接口和时序模式变化时间

tRC\_min

RE# 循环时间

tREA\_max

RE#访问时间

tREH\_min

RE# 高保持时间

tRHOH\_min

RE# 高到输出保持

tRHW\_min

RE# 高到 WE# 低

tRHZ\_max

RE# 高输出 hi-Z

tRLOH\_min

RE# 低到输出保持

tRP\_min

RE# 脉冲宽度

tRR\_min

准备好 RE# 低（仅数据）

tRST\_max

器件复位时间，从 R/B# 的下降沿到 R/B# 的上升沿测量。

tWB\_max

WE# 高到 SR[6] 低

tWC\_min

WE#循环时间

tWH\_min

WE#高保持时间

tWHR\_min

WE# 高到 RE# 低

tWP\_min

WE#脉冲宽度

tWW\_min

WP# 过渡到 WE# 低

**说明**

该结构定义了 SDR NAND 芯片的时序要求。这些信息可以在每个 NAND 数据表中找到，ONFI 规范中描述了时序含义：www.onfi.org/~/media/ONFI/specs/onfi\_3\_1\_spec.pdf（第 4.15 章时序参数）

所有这些时间都以皮秒表示。

### enum nand\_data\_interface\_type

NAND接口时序类型

**常量**

NAND\_SDR\_IFACE

单一数据速率接口

### struct nand\_data\_interface

NAND接口时序

**定义**

struct nand\_data\_interface {

enum nand\_data\_interface\_type type;

union {

struct nand\_sdr\_timings sdr;

} timings;

};

**成员**

type

计时类型

timings

时机、类型不同的类型

timings.sdr

当类型为 时使用它NAND\_SDR\_IFACE。

### const struct [nand\_sdr\_timings](https://www.kernel.org/doc/html/v4.19/driver-api/mtdnand.html" \l "c.nand_sdr_timings" \o "nand_sdr_timings) \* nand\_get\_sdr\_timings( const struct [nand\_data\_interface](https://www.kernel.org/doc/html/v4.19/driver-api/mtdnand.html" \l "c.nand_data_interface" \o "nand_data_interface) \*  conf )

从数据接口获取 SDR 时序

**参数**

const struct nand\_data\_interface \* conf

数据接口

### struct nand\_manufacturer\_ops

NAND 运营商

**定义**

struct nand\_manufacturer\_ops {

void (\*detect)(struct nand\_chip \*chip);

int (\*init)(struct nand\_chip \*chip);

void (\*cleanup)(struct nand\_chip \*chip);

void (\*fixup\_onfi\_param\_page)(struct nand\_chip \*chip, struct nand\_onfi\_params \*p);

};

**成员**

detect

检测 NAND 内存组织和功能

init

初始化所有供应商特定字段（如 ->:c:func: read\_retry() 实现）（如果有）。

cleanup

->:c:func: init()函数可能已经分配了资源， ->:c:func: cleanup() 在这里让供应商特定代码释放这些资源。

fixup\_onfi\_param\_page

将供应商特定的修正应用于 ONFI 参数页面。这是在验证校验和后调用的。

### struct nand\_op\_cmd\_instr

命令指令的定义

**定义**

struct nand\_op\_cmd\_instr {

u8 opcode;

};

**成员**

opcode

一个周期内发出的指令

### struct nand\_op\_addr\_instr

地址指令的定义

**定义**

struct nand\_op\_addr\_instr {

unsigned int naddrs;

const u8 \*addrs;

};

**成员**

naddrs

地址数组的长度

addrs

包含要发出的地址周期的数组

### struct nand\_op\_data\_instr

数据指令的定义

**定义**

struct nand\_op\_data\_instr {

unsigned int len;

union {

void \*in;

const void \*out;

} buf;

bool force\_8bit;

};

**成员**

len

要移动的数据字节数

buf

要填充的缓冲区

buf.in

从 NAND 芯片读取时要填充的缓冲区

buf.out

写入 NAND 芯片时读取的缓冲区

force\_8bit

强制 8 位访问

**说明**

请注意，“in”和“out”与 ONFI 规范相反，是从控制器的角度来看的，因此“in”是从 NAND 芯片读取，而“out”是写入 NAND 芯片。

### struct nand\_op\_waitrdy\_instr

等待就绪指令的定义

**定义**

struct nand\_op\_waitrdy\_instr {

unsigned int timeout\_ms;

};

**成员**

timeout\_ms

以 ms 为单位等待就绪/忙碌引脚时的最大延迟

### enum nand\_op\_instr\_type

所有指令类型的定义

**常量**

NAND\_OP\_CMD\_INSTR 命令指令

NAND\_OP\_ADDR\_INSTR 地址指令

NAND\_OP\_DATA\_IN\_INSTR 数据输入指令

NAND\_OP\_DATA\_OUT\_INSTR 数据输出指令

NAND\_OP\_WAITRDY\_INSTR 等待准备就绪指令

### struct nand\_op\_instr

指令对象

**定义**

struct nand\_op\_instr {

enum nand\_op\_instr\_type type;

union {

struct nand\_op\_cmd\_instr cmd;

struct nand\_op\_addr\_instr addr;

struct nand\_op\_data\_instr data;

struct nand\_op\_waitrdy\_instr waitrdy;

} ctx;

unsigned int delay\_ns;

};

**成员**

type

指令类型

ctx

指令相关的额外数据。需要根据type使用适当的元素

ctx.cmd

当type是NAND\_OP\_CMD\_INSTR时使用

ctx.addr

当type是NAND\_OP\_ADDR\_INSTR时使用

ctx.data

当type是NAND\_OP\_DATA\_IN\_INSTR或NAND\_OP\_DATA\_OUT\_INSTR时使用

ctx.waitrdy

当type是NAND\_OP\_WAITRDY\_INSTR时使用

delay\_ns

指令发出后控制器应该应用的延迟。现代控制器通常有内部时间控制逻辑，在这种情况下，控制器驱动程序可以忽略此字段。

### struct nand\_subop

子操作

**定义**

struct nand\_subop {

const struct nand\_op\_instr \*instrs;

unsigned int ninstrs;

unsigned int first\_instr\_start\_off;

unsigned int last\_instr\_end\_off;

};

**成员**

instrs

指令数组

ninstrs instrs

数组的长度

first\_instr\_start\_off

子操作的第一条指令的起始偏移量

last\_instr\_end\_off

子操作的最后一条指令的结束偏移量（排除）

**说明**

first\_instr\_start\_off和last\_instr\_end\_off仅适用于数据或地址指令。

当操作不能被NAND控制器处理时，解析器会将其分成子操作，并将其传递给控制器驱动程序。

### struct nand\_op\_parser\_addr\_constraints

地址指令的约束

**定义**

struct nand\_op\_parser\_addr\_constraints {

unsigned int maxcycles;

};

**成员**

maxcycles

控制器在单个步骤中可以发出的最大地址周期数

### struct nand\_op\_parser\_data\_constraints

数据指令的约束

**定义**

struct nand\_op\_parser\_data\_constraints {

unsigned int maxlen;

};

**成员**

maxlen

控制器可以在单个步骤中处理的最大数据长度

### struct nand\_op\_parser\_pattern\_elem

模式的一个元素

**定义**

struct nand\_op\_parser\_pattern\_elem {

enum nand\_op\_instr\_type type;

bool optional;

union {

struct nand\_op\_parser\_addr\_constraints addr;

struct nand\_op\_parser\_data\_constraints data;

} ctx;

};

**成员**

type

指令类型

optional

模式中此元素是可选还是必需的

ctx

地址或数据约束

ctx.addr

地址约束（周期数）

ctx.data

数据约束（数据长度）

### struct nand\_op\_parser\_pattern

NAND子操作模式说明符

**定义**

struct nand\_op\_parser\_pattern {

const struct nand\_op\_parser\_pattern\_elem \*elems;

unsigned int nelems;

int (\*exec)(struct nand\_chip \*chip, const struct nand\_subop \*subop);

};

**成员**

elems

模式元素数组

nelems elems

数组中模式元素的数量

exec

将发出一个子操作的函数

**说明**

模式是一个元素列表，每个元素表示一个带有其约束条件的指令。模式本身由核心用于将NAND芯片操作与NAND控制器操作匹配。一旦找到NAND控制器操作模式和NAND芯片操作（或NAND操作的子集）之间的匹配，将调用模式->：c：func：exec（）钩子，以便控制器驱动程序可以在总线上发出操作。

控制器驱动程序应声明其支持的模式并将此模式列表（使用以下宏的帮助创建）传递给nand\_op\_parser\_exec\_op（）助手。

### struct nand\_op\_parser

NAND控制器操作解析器说明符

**定义**

struct nand\_op\_parser {

const struct nand\_op\_parser\_pattern \*patterns;

unsigned int npatterns;

};

**成员**

patterns

支持的模式数组

npatterns patterns

数组的长度

**说明**

解析器说明符只是支持的模式数组，每当nand\_op\_parser\_exec\_op（）尝试执行NAND操作（或尝试确定是否支持特定操作）时，它将迭代该数组。

值得提到的是，模式将按其声明顺序进行测试，并且将采取第一个匹配，因此有必要适当地排序模式，以便简单/低效的模式放置在列表的末尾。通常，这是您放置单个指令模式的地方。

### struct nand\_operation

NAND操作说明符

**定义**

struct nand\_operation {

const struct nand\_op\_instr \*instrs;

unsigned int ninstrs;

};

**成员**

instrs

要执行的指令数组

ninstrs

instrs数组的长度

**说明**

实际的操作结构将传递给chip->：c：func：exec\_op（）。

### struct nand\_chip

NAND私有闪存芯片数据

**定义**

struct nand\_chip {

struct mtd\_info mtd;

void \_\_iomem \*IO\_ADDR\_R;

void \_\_iomem \*IO\_ADDR\_W;

uint8\_t (\*read\_byte)(struct mtd\_info \*mtd);

u16 (\*read\_word)(struct mtd\_info \*mtd);

void (\*write\_byte)(struct mtd\_info \*mtd, uint8\_t byte);

void (\*write\_buf)(struct mtd\_info \*mtd, const uint8\_t \*buf, int len);

void (\*read\_buf)(struct mtd\_info \*mtd, uint8\_t \*buf, int len);

void (\*select\_chip)(struct mtd\_info \*mtd, int chip);

int (\*block\_bad)(struct mtd\_info \*mtd, loff\_t ofs);

int (\*block\_markbad)(struct mtd\_info \*mtd, loff\_t ofs);

void (\*cmd\_ctrl)(struct mtd\_info \*mtd, int dat, unsigned int ctrl);

int (\*dev\_ready)(struct mtd\_info \*mtd);

void (\*cmdfunc)(struct mtd\_info \*mtd, unsigned command, int column, int page\_addr);

int(\*waitfunc)(struct mtd\_info \*mtd, struct nand\_chip \*this);

int (\*exec\_op)(struct nand\_chip \*chip,const struct nand\_operation \*op, bool check\_only);

int (\*erase)(struct mtd\_info \*mtd, int page);

int (\*set\_features)(struct mtd\_info \*mtd, struct nand\_chip \*chip, int feature\_addr, uint8\_t \*subfeature\_para);

int (\*get\_features)(struct mtd\_info \*mtd, struct nand\_chip \*chip, int feature\_addr, uint8\_t \*subfeature\_para);

int (\*setup\_read\_retry)(struct mtd\_info \*mtd, int retry\_mode);

int (\*setup\_data\_interface)(struct mtd\_info \*mtd, int chipnr, const struct nand\_data\_interface \*conf);

int chip\_delay;

unsigned int options;

unsigned int bbt\_options;

int page\_shift;

int phys\_erase\_shift;

int bbt\_erase\_shift;

int chip\_shift;

int numchips;

uint64\_t chipsize;

int pagemask;

u8 \*data\_buf;

int pagebuf;

unsigned int pagebuf\_bitflips;

int subpagesize;

uint8\_t bits\_per\_cell;

uint16\_t ecc\_strength\_ds;

uint16\_t ecc\_step\_ds;

int onfi\_timing\_mode\_default;

int badblockpos;

int badblockbits;

struct nand\_id id;

struct nand\_parameters parameters;

u16 max\_bb\_per\_die;

u32 blocks\_per\_die;

struct nand\_data\_interface data\_interface;

int read\_retries;

flstate\_t state;

uint8\_t \*oob\_poi;

struct nand\_controller \*controller;

struct nand\_ecc\_ctrl ecc;

unsigned long buf\_align;

struct nand\_controller dummy\_controller;

uint8\_t \*bbt;

struct nand\_bbt\_descr \*bbt\_td;

struct nand\_bbt\_descr \*bbt\_md;

struct nand\_bbt\_descr \*badblock\_pattern;

void \*priv;

struct {

const struct nand\_manufacturer \*desc;

void \*priv;

} manufacturer;

};

**成员**

mtd

MTD 设备注册到 MTD 框架

IO\_ADDR\_R

[BOARDSPECIFIC] 读取闪存设备 8 条 I/O 线的地址

IO\_ADDR\_W

[BOARDSPECIFIC] 地址写入闪存设备的 8 个 I/O 线。

read\_byte

[可替换] 从芯片读取一个字节

read\_word

[可替换] 从芯片中读取一个字

write\_byte

[REPLACEABLE] 在低 8 条 I/O 线上向芯片写入一个字节

write\_buf

[REPLACEABLE] 将缓冲区中的数据写入芯片

read\_buf

[REPLACEABLE] 从芯片读取数据到缓冲区

select\_chip

[REPLACEABLE] 选择芯片编号

block\_bad

[REPLACEABLE] 检查块是否坏，使用 OOB 标记

block\_markbad

[REPLACEABLE] 标记块坏

cmd\_ctrl

[BOARDSPECIFIC] 用于控制 ALE/CLE/nCE 的硬件特定函数。也用来写命令和地址

dev\_ready

[BOARDSPECIFIC] 用于访问设备就绪/忙线的硬件特定函数。如果设置为 NULL，则无法访问就绪/忙碌，并且就绪/忙碌信息将从芯片状态寄存器中读取。

cmdfunc

[REPLACEABLE] 用于向芯片写入命令的硬件特定函数。

waitfunc

[REPLACEABLE] 等待就绪的硬件特定功能。

exec\_op

执行 NAND 操作的控制器特定方法。此方法替换 ->:c:func: cmdfunc()、 ->{read,write}\_{buf,byte,word}()、 ->:c:func: dev\_ready ()和 ->:c:func:等待函数（）。

erase

[REPLACEABLE] 擦除功能

set\_features

[REPLACEABLE] 设置NAND芯片特性

get\_features

[REPLACEABLE] 获取NAND芯片特性

setup\_read\_retry

[FLASHSPECIFIC] 用于设置读取重试模式的闪存（供应商）特定功能。MLC NAND 最需要。

setup\_data\_interface

[可选] 设置数据接口和时序。如果 chipnr 设置为NAND\_DATA\_IFACE\_CHECK\_ONLY这意味着不应应用该配置，而应仅对其进行检查。

chip\_delay

[BOARDSPECIFIC] 将数据从阵列传输到读取寄存器 (tR) 的芯片相关延迟。

options

[BOARDSPECIFIC] 各种芯片选项。它们可以部分设置为通知 nand\_scan 特殊功能。请参阅定义以获取更多说明。

bbt\_options

[实习生] 坏块特定选项。此处使用的所有选项都必须来自 bbm.h。默认情况下，这些选项将被复制到适当的 nand\_bbt\_descr。

page\_shift

[INTERN] 页中的地址位数（列地址位）。

phys\_erase\_shift

[INTERN] 物理擦除块中的地址位数

bbt\_erase\_shift

[INTERN] bbt 条目中的地址位数

chip\_shift

[INTERN] 一个芯片的地址位数

numchips

[实习生] 物理芯片数量

chipsize

[实习生] 多芯片阵列的一个芯片的大小

pagemask

[INTERN] 页码掩码 = 数（页/片）- 1

data\_buf

[INTERN] 数据缓冲区，大小为（页面大小 + oobsize）。

pagebuf

[INTERN] 保存当前在 data\_buf 中的页码。

pagebuf\_bitflips

[INTERN] 保存当前在 data\_buf 中的页面的位翻转计数。

subpagesize

[INTERN] 持有 subpagesize

bits\_per\_cell

[INTERN] 每个单元格的位数。即，1 表示 SLC。

ecc\_strength\_ds

[实习生] 数据表中的 ECC 可纠正性。保证可纠正的每个ecc\_step\_ds的最小位错误量。如果未知，则设置为零。

ecc\_step\_ds

[实习生] ecc\_strength\_ds所需的 ECC 步骤，也来自数据表。如果已知，它是推荐的 ECC 步长；如果未知，则设置为零。

onfi\_timing\_mode\_default

[INTERN] 默认 ONFI 计时模式。如果芯片符合 ONFI 标准，则此字段设置为实际使用的 ONFI 模式；如果 NAND 芯片不符合 ONFI 标准，则从数据表中推断。

badblockpos

[INTERN] oob 区域中坏块标记的位置。

badblockbits

[INTERN] 好块的坏块标记位置中设置的最小位数；即，当 badblockbits == 7 时，BBM == 11110111b 还不错

id

[实习生]持有NAND ID

parameters

[INTERN] 以易于阅读的形式保存通用参数。

max\_bb\_per\_die

[实习生] 这个 nand 设备的每个死块的最大坏块数将遇到它们的生命周期。

blocks\_per\_die

[实习生] 芯片中 PEB 的数量

data\_interface

[INTERN] NAND接口时序信息

read\_retries

[实习生] 支持的读取重试模式数

state

[INTERN] NAND设备的当前状态

oob\_poi

“毒值缓冲区”，用于在写入前对 OOB 数据进行布局

controller

[REPLACEABLE] 指向在多个独立设备之间共享的硬件控制器结构的指针。

ecc

[BOARDSPECIFIC] ECC 控制结构

buf\_align

平台要求的最小缓冲区对齐

dummy\_controller

只能控制单个芯片的驱动程序的虚拟控制器实现

bbt

[实习生] 坏块表指针

bbt\_td

[可替换] 用于闪存查找的坏块表描述符。

bbt\_md

[可替换] 坏块表镜像描述符

badblock\_pattern

[可替换] 用于初始坏块扫描的坏块扫描模式。

priv

[可选] 指向私有芯片数据的指针

manufacturer

[实习生] 包含制造商信息

manufacturer.desc

[实习生] 包含制造商的描述

manufacturer.priv

[实习生] 包含制造商私人信息

### struct nand\_flash\_dev

NAND闪存设备ID结构

**定义**

struct nand\_flash\_dev {

char \*name;

union {

struct {

uint8\_t mfr\_id;

uint8\_t dev\_id;

};

uint8\_t id[NAND\_MAX\_ID\_LEN];

};

unsigned int pagesize;

unsigned int chipsize;

unsigned int erasesize;

unsigned int options;

uint16\_t id\_len;

uint16\_t oobsize;

struct {

uint16\_t strength\_ds;

uint16\_t step\_ds;

} ecc;

int onfi\_timing\_mode\_default;

};

**成员**

name

NAND芯片的可读名称

{unnamed\_union}

匿名

{unnamed\_struct}

匿名

mfr\_id

完整芯片ID数组的制造商ID部分（引用与\*\*id\*\*[0]相同的内存地址）

dev\_id

完整芯片ID数组的设备ID部分（引用与\*\*id\*\*[1]相同的内存地址）

id

完整设备ID数组

pagesize

NAND页的大小（以字节为单位）；如果为0，则从扩展NAND芯片ID数组确定实际页面大小（以及擦除块大小）

chipsize

MiB的总芯片大小

erasesize

擦除块大小（如果为0，则从扩展ID确定）

options

存储各种芯片位选项

id\_len

id的有效长度。

oobsize

OOB大小

ecc

来自数据表的ECC可校正性和步骤信息。

ecc.strength\_ds

来自数据表的ECC可校正性，与nand\_chip{}中的ecc\_strength\_ds相同。

ecc.step\_ds

由ecc.strength\_ds要求的ECC步骤，与nand\_chip{}中的ecc\_step\_ds相同，也来自数据表。例如，“每512字节的4位ECC”可以使用NAND\_ECC\_INFO（4，512）设置。

onfi\_timing\_mode\_default

重置NAND后输入的默认ONFI时序模式。应从数据表中说明的时序中推导出。

### struct nand\_manufacturer

NAND闪存制造商结构

**定义**

struct nand\_manufacturer {

int id;

char \*name;

const struct nand\_manufacturer\_ops \*ops;

};

**成员**

id

设备的制造商ID代码。

name

制造商名称

ops

制造商操作

### struct platform\_nand\_chip

芯片级设备结构

**定义**

struct platform\_nand\_chip {

int nr\_chips;

int chip\_offset;

int nr\_partitions;

struct mtd\_partition \*partitions;

int chip\_delay;

unsigned int options;

unsigned int bbt\_options;

const char \*\*part\_probe\_types;

};

**成员**

nr\_chips

要扫描的最大芯片数

chip\_offset

芯片号偏移量

nr\_partitions

由分区指向的分区数（或为零）

partitions

mtd分区列表

chip\_delay

us中的R/B延迟值

options

选项标志，例如16位总线宽度

bbt\_options

BBT选项标志，例如NAND\_BBT\_USE\_FLASH

part\_probe\_types

探测类型的以NULL结尾的数组

### struct platform\_nand\_ctrl

控制器级设备结构

**定义**

struct platform\_nand\_ctrl {

int (\*probe)(struct platform\_device \*pdev);

void (\*remove)(struct platform\_device \*pdev);

int (\*dev\_ready)(struct mtd\_info \*mtd);

void (\*select\_chip)(struct mtd\_info \*mtd, int chip);

void (\*cmd\_ctrl)(struct mtd\_info \*mtd, int dat, unsigned int ctrl);

void (\*write\_buf)(struct mtd\_info \*mtd, const uint8\_t \*buf, int len);

void (\*read\_buf)(struct mtd\_info \*mtd, uint8\_t \*buf, int len);

void \*priv;

};

**成员**

probe

平台特定的功能来探测/设置硬件

remove

平台特定的函数来移除/拆卸硬件

dev\_ready

平台特定函数读取就绪/忙碌引脚

select\_chip

平台特定芯片选择函数

cmd\_ctrl

用于控制ALE/CLE/nCE的平台特定函数。也用于写入命令和地址

write\_buf

用于写缓冲器的平台特定函数

read\_buf

用于读取缓冲器的平台特定函数

priv

传输驱动程序特定的设置的私有数据

**说明**

所有字段都是可选的，并且取决于硬件驱动程序的要求

### struct platform\_nand\_data

平台特定数据的容器结构

**定义**

struct platform\_nand\_data {

struct platform\_nand\_chip chip;

struct platform\_nand\_ctrl ctrl;

};

**成员**

chip

芯片级芯片结构

ctrl

控制器级设备结构

### int nand\_opcode\_8bits(unsigned int command)

**参数**

unsigned int command

要检查的操作码

## 提供公共函数

本章包含导出的NAND内核API函数的自动生成文档。每个函数都有一个短说明，标有[XXX]标识符。有关说明，请参见“文档提示”章节。

### void nand\_wait\_ready(struct mtd\_info \* mtd)

[通用]在命令之后等待就绪引脚。

**参数**

struct mtd\_info \* mtd

MTD设备结构

**说明**

等待命令后的就绪引脚，并在超时发生时发出警告。

### int nand\_soft\_waitrdy(struct nand\_chip \* chip, unsigned long timeout\_ms)

轮询STATUS reg，直到RDY位设置为1

**参数**

struct nand\_chip \* chip

NAND芯片struct

unsigned long timeout\_ms

超时时间（毫秒）

**说明**

使用->:c:func:exec\_op()来轮询STATUS寄存器，直到RDY位为1。如果在指定的超时时间内未发生，则返回-ETIMEDOUT。

当控制器无法访问NAND R/B引脚时，此助手函数旨在使用。

请注意，从->:c:func:exec\_op()实现调用此助手函数意味着->:c:func:exec\_op()必须是可重入的。

如果NAND芯片准备就绪，则返回0，否则返回负错误。

### int nand\_get\_features（struct nand\_chip \* chip，int addr，u8 \* subfeature\_param）

执行GET\_FEATURE的包装器

**参数**

struct nand\_chip \* chip

NAND芯片信息struct

int addr

功能地址

u8 \* subfeature\_param

子功能参数，四个字节的数组

**说明**

成功返回0，否则返回负错误。如果无法执行此操作，则返回-ENOTSUPP。

### int nand\_set\_features（struct nand\_chip \* chip，int addr，u8 \* subfeature\_param）

执行SET\_FEATURE的包装器

**参数**

struct nand\_chip \* chip

NAND芯片信息struct

int addr

功能地址

u8 \* subfeature\_param

子功能参数，四个字节的数组

**说明**

成功返回0，否则返回负错误。如果无法执行此操作，则返回-ENOTSUPP。

### int nand\_read\_page\_op（struct nand\_chip \* chip，unsigned int page，unsigned int offset\_in\_page，void \* buf，unsigned int len）

执行READ PAGE操作

**参数**

struct nand\_chip \* chip

NAND芯片

unsigned int page

要读取的页

unsigned int offset\_in\_page

页面内偏移量

void \* buf

用于存储数据的缓冲区

unsigned int len

缓冲区的长度

**说明**

该函数发出READ PAGE操作。此函数不会选择/取消选择CS线。

成功返回0，否则返回负错误代码。

### int nand\_change\_read\_column\_op（struct nand\_chip \* chip，unsigned int offset\_in\_page，void \* buf，unsigned int len，bool force\_8bit）

执行CHANGE\_READ\_COLUMN操作

**参数**

struct nand\_chip \* chip

NAND芯片

unsigned int offset\_in\_page

页面内偏移量

void \* buf

用于存储数据的缓冲区

unsigned int len

缓冲区的长度

bool force\_8bit

强制使用8位总线访问

**说明**

该函数发出CHANGE READ COLUMN操作。此函数不会选择/取消选择CS线。

成功返回0，否则返回负错误代码。

### int nand\_read\_oob\_op（struct nand\_chip \* chip，unsigned int page，unsigned int offset\_in\_oob，void \* buf，unsigned int len）

执行READ OOB操作

**参数**

struct nand\_chip \* chip

NAND芯片

unsigned int page

要读取的页

unsigned int offset\_in\_oob

OOB区内的偏移量

void \* buf

用于存储数据的缓冲区

unsigned int len

缓冲区的长度

**说明**

该函数发出READ OOB操作。此函数不会选择/取消选择CS线。

成功返回0，否则返回负错误代码。

### int nand\_prog\_page\_begin\_op（struct nand\_chip \* chip，unsigned int page，unsigned int offset\_in\_page，const void \* buf，unsigned int len）

启动PROG PAGE操作

**参数**

struct nand\_chip \* chip

NAND芯片

unsigned int page

要写入的页

unsigned int offset\_in\_page

页面内偏移量

const void \* buf

包含要写入页面的数据的缓冲区

unsigned int len

缓冲区的长度

**说明**

该函数发出PROG PAGE操作的前半部分。此函数不会选择/取消选择CS线。

成功返回0，否则返回负错误代码。

### int nand\_prog\_page\_end\_op（struct nand\_chip \* chip）

结束PROG PAGE操作

**参数**

struct nand\_chip \* chip

NAND芯片

**说明**

该函数发出PROG PAGE操作的后半部分。此函数不会选择/取消选择CS线。

成功返回0，否则返回负错误代码。

### int nand\_prog\_page\_op（struct nand\_chip \* chip，unsigned int page，unsigned int offset\_in\_page，const void \* buf，unsigned int len）

执行完整的PROG PAGE操作

**参数**

struct nand\_chip \* chip

NAND芯片

unsigned int page

要写入的页

unsigned int offset\_in\_page

页面内偏移量

const void \* buf

包含要写入页面的数据的缓冲区

unsigned int len

缓冲区的长度

**说明**

该函数发出完整的PROG PAGE操作。此函数不会选择/取消选择CS线。

成功返回0，否则返回负错误代码。

### int nand\_change\_write\_column\_op（struct nand\_chip \* chip，unsigned int offset\_in\_page，const void \* buf，unsigned int len，bool force\_8bit)

执行CHANGE WRITE COLUMN操作

**参数**

struct nand\_chip \* chip

NAND芯片

unsigned int offset\_in\_page

页面内偏移量

const void \* buf

包含要发送到NAND的数据的缓冲区

unsigned int len

缓冲区的长度

bool force\_8bit

强制使用8位总线访问

**说明**

该函数发出CHANGE WRITE COLUMN操作。此函数不会选择/取消选择CS线。

成功返回0，否则返回负错误代码。

### int nand\_readid\_op（struct nand\_chip \* chip，u8 addr，void \* buf，unsigned int len）

执行READID操作

**参数**

struct nand\_chip \*chip

NAND芯片

u8 addr

在READID命令之后传递的地址周期

void \*buf

用于存储ID的缓冲区

unsigned int len

缓冲区的长度

**说明**

此函数发送READID命令并读取NAND返回的ID。此函数不选中/取消选中CS线。

成功返回0，否则返回负错误代码。

### int nand\_status\_op(struct nand\_chip \*chip, u8 \*status)

执行STATUS操作

**参数**

struct nand\_chip \*chip

NAND芯片

u8 \*status

存储NAND状态的输出变量

**说明**

此函数发送STATUS命令并读取NAND返回的状态。此函数不选中/取消选中CS线。

成功返回0，否则返回负错误代码。

### int nand\_exit\_status\_op(struct nand\_chip \*chip)

退出STATUS操作

**参数**

struct nand\_chip \*chip

NAND芯片

**说明**

此函数发送READ0命令以取消STATUS命令的效果，以避免只读取状态直到发送新的读取命令。

此函数不选中/取消选中CS线。

成功返回0，否则返回负错误代码。

### int nand\_erase\_op(struct nand\_chip \*chip, unsigned int eraseblock)

执行擦除操作

**参数**

struct nand\_chip \*chip

NAND芯片

unsigned int eraseblock

要擦除的块

**说明**

此函数发送ERASE命令并在返回之前等待NAND就绪。此函数不选中/取消选中CS线。

成功返回0，否则返回负错误代码。

### int nand\_reset\_op(struct nand\_chip \*chip)

执行重置操作

**参数**

struct nand\_chip \*chip

NAND芯片

**说明**

此函数发送RESET命令并在返回之前等待NAND就绪。此函数不选中/取消选中CS线。

成功返回0，否则返回负错误代码。

### int nand\_read\_data\_op(struct nand\_chip \*chip, void \*buf, unsigned int len, bool force\_8bit)

从NAND中读取数据

**参数**

struct nand\_chip \*chip

NAND芯片

void \*buf

用于存储数据的缓冲区

unsigned int len

缓冲区的长度

bool force\_8bit

强制使用8位总线访问

**说明**

此函数在总线上执行原始数据读取。通常在启动另一个NAND操作（如nand\_read\_page\_op（））后使用此函数。此函数不选中/取消选中CS线。

成功返回0，否则返回负错误代码。

### int nand\_write\_data\_op(struct nand\_chip \*chip, const void \*buf, unsigned int len, bool force\_8bit)

从NAND中写入数据

**参数**

struct nand\_chip \*chip

NAND芯片

const void \*buf

包含要在总线上发送的数据的缓冲区

unsigned int len

缓冲区的长度

bool force\_8bit

强制使用8位总线访问

**说明**

此函数在总线上执行原始数据写入。通常在启动另一个NAND操作（如nand\_write\_page\_begin\_op（））后使用此函数。此函数不选中/取消选中CS线。

成功返回0，否则返回负错误代码。

### int nand\_op\_parser\_exec\_op(struct nand\_chip \*chip, const struct nand\_op\_parser \*parser, const struct nand\_operation \*op, bool check\_only)

执行操作解析器

**参数**

struct nand\_chip \*chip

NAND芯片

const struct nand\_op\_parser \*parser

由控制器驱动程序提供的模式说明

const struct nand\_operation \*op

要操作的NAND操作

bool check\_only

为true时，函数仅检查op是否可处理但不执行操作

**说明**

协助函数旨在简化仅支持有限指令序列的NAND控制器驱动程序的集成。支持的序列在解析器中说明，并且框架负责将op分解为多个子操作（如果需要），并将其返回到匹配模式的->：c：func：exec（）回调中，如果check\_only设置为false，则可以处理操作。

NAND控制器驱动程序应从其自己的->：c：func：exec\_op（）实现中调用此函数。

成功返回0，否则返回负错误代码。可以由不受支持的操作（没有支持的模式能够处理所请求的操作）或匹配模式->：c：func：exec（）挂钩返回的错误引起失败。

### unsigned int nand\_subop\_get\_addr\_start\_off(const struct nand\_subop \*subop, unsigned int instr\_idx)

获取地址数组中的起始偏移量

**参数**

const struct nand\_subop \*subop

整个子操作

unsigned int instr\_idx

子操作内指令的索引

**说明**

在驱动程序开发期间，可以试图直接使用地址指令的->addr.addrs字段。这是错误的，因为地址指令可能会被分割。

给定一个地址指令，返回第一个要发出的周期的偏移量。

### unsigned int nand\_subop\_get\_num\_addr\_cyc(const struct nand\_subop \*subop, unsigned int instr\_idx)

获取要断言的剩余地址周期

**参数**

const struct nand\_subop \*subop

整个子操作

unsigned int instr\_idx

子操作内指令的索引

**说明**

在驱动程序开发期间，人们可能会尝试直接使用数据指令的->addr->naddrs字段。这是错误的，因为指令可能会被分割。

给定一个地址指令，返回要发出的地址周期数。

### unsigned int nand\_subop\_get\_data\_start\_off(const struct nand\_subop \* subop, unsigned int instr\_idx)

获取数据数组中的起始偏移量

**参数**

const struct nand\_subop \* subop

整个子操作

unsigned int instr\_idx

子操作内的指令索引

**说明**

在驱动程序开发期间，人们可能会尝试直接使用数据指令的->data->buf.{in，out}字段。这是错误的，因为数据指令可能会被分割。

给定一个数据指令，返回要从哪里开始的偏移量。

### unsigned int nand\_subop\_get\_data\_len(const struct nand\_subop \* subop, unsigned int instr\_idx)

获取要检索的字节数

**参数**

const struct nand\_subop \* subop

整个子操作

unsigned int instr\_idx

子操作内的指令索引

**说明**

在驱动程序开发期间，人们可能会尝试直接使用数据指令的->data->len字段。这是错误的，因为数据指令可能会被分割。

返回要发送/接收的数据块的长度。

### int nand\_reset(struct nand\_chip \* chip，int chipnr)

重置和初始化NAND设备

**参数**

struct nand\_chip \* chip

NAND芯片

int chipnr

内部die id

**说明**

保存时间数据结构，然后应用SDR时序模式0（有关详细信息，请参见nand\_reset\_data\_interface），执行重置操作，然后恢复先前的时间。

成功返回0，否则返回负错误代码。

### int nand\_check\_erased\_ecc\_chunk(void \* data，int datalen，void \* ecc，int ecclen，void \* extraoob，int extraooblen，int bitflips\_threshold)

检查ECC块是否包含（几乎）只有0xff数据

**参数**

void \* data

要测试的数据缓冲区

int datalen

数据长度

void \* ecc

ECC缓冲区

int ecclen

ECC长度

void \* extraoob

额外OOB缓冲区

int extraooblen

额外OOB长度

int bitflips\_threshold

最大位翻转次数

**说明**

检查数据缓冲区及其关联的ECC和OOB数据是否仅包含0xff模式，这意味着底层区域已被擦除并准备好进行编程。bitflips\_threshold指定在认为区域未擦除之前最大位翻转次数。

注意

1 / ECC算法正在处理通常与NAND页面大小不同的预定义块大小。在修复位翻转时，ECC引擎将报告每个块的错误数量，并且NAND核心基础设施需要您返回整个页面的最大位翻转次数。这就是为什么您应该总是在单个块上使用此函数而不是在整个页面上使用的原因。检查每个块后，您应该相应地更新您的max\_bitflips值。

2 / 在检查已擦除的页面的位翻转时，您不仅应该检查有效数据，还应该检查它们关联的ECC数据，因为用户可能已将几乎所有位编程为1，但剩下的一些。在这种情况下，我们不应将块视为已删除，并且检查ECC字节可以防止这种情况。

3 / extraoob参数是可选的，并且应该在您的OOB数据受ECC引擎保护时使用。如果您支持子页面并希望将一些额外的OOB数据附加到ECC块上，则也可以使用它。

返回小于或等于bitflips\_threshold的位翻转的正数，或-ERROR\_CODE以超过阈值的错误码。成功时，传递的缓冲区填充为0xff。

### int nand\_read\_page\_raw\_notsupp(struct mtd\_info \* mtd，struct nand\_chip \* chip，u8 \* buf，int oob\_required，int page)

虚拟读取原始页面函数

**参数**

struct mtd\_info \* mtd

mtd信息结构

struct nand\_chip \* chip

nand芯片信息结构

u8 \* buf

用于存储读取数据的缓冲区

int oob\_required

调用者需要读取OOB数据到chip->oob\_poi

int page

要读取的页面号

**说明**

无条件返回-ENOTSUPP。

### int nand\_read\_page\_raw(struct mtd\_info \* mtd，struct nand\_chip \* chip，uint8\_t \* buf，int oob\_required，int page)

[INTERN]读取原始页面数据，没有ECC

**参数**

struct mtd\_info \* mtd

mtd信息结构

struct nand\_chip \* chip

nand芯片信息结构

uint8\_t \* buf

用于存储读取数据的缓冲区

int oob\_required

调用者需要读取OOB数据到chip->oob\_poi

int page

要读取的页面号

**说明**

不适用于综合计算ECC控制器，该控制器使用特殊的oob布局。

### int nand\_read\_oob\_std(struct mtd\_info \* mtd，struct nand\_chip \* chip，int page)

[REPLACEABLE]最常见的OOB数据读取函数

**参数**

struct mtd\_info \* mtd

mtd信息结构

struct nand\_chip \* chip

nand芯片信息结构

int page

要读取的页面号

### int nand\_read\_oob\_syndrome(struct mtd\_info \* mtd，struct nand\_chip \* chip，int page)

[REPLACEABLE] HW ECC使用综合读取OOB数据功能

**参数**

struct mtd\_info \* mtd

mtd信息结构

struct nand\_chip \* chip

NAND芯片信息struct

int page

要读取的页码

### int nand\_write\_oob\_std（struct mtd\_info \* mtd，struct nand\_chip \* chip，int page）

[可替换]最常见的OOB数据写入函数

**参数**

struct mtd\_info \* mtd

mtd信息struct

struct nand\_chip \* chip

nand芯片信息struct

int page

要写入的页码

### int nand\_write\_oob\_syndrome（struct mtd\_info \* mtd，struct nand\_chip \* chip，int page）

[可替换]HW ECC带有综合征的OOB数据写入函数 - 仅适用于大页Flash

**参数**

struct mtd\_info \* mtd

mtd信息struct

struct nand\_chip \* chip

nand芯片信息struct

int page

要写入的页码

### int nand\_write\_page\_raw\_notsupp（struct mtd\_info \* mtd，struct nand\_chip \* chip，const u8 \* buf，int oob\_required，int page）

虚拟的原始页写入函数

**参数**

struct mtd\_info \* mtd

mtd信息struct

struct nand\_chip \* chip

nand芯片信息struct

const u8 \* buf

数据缓冲区

int oob\_required

必须写入chip->oob\_poi到OOB

int page

要写入的页码

**说明**

无条件地返回 -ENOTSUPP。

### int nand\_write\_page\_raw（struct mtd\_info \* mtd，struct nand\_chip \* chip，const uint8\_t \* buf，int oob\_required，int page）

[INTERN]原始页写入函数

**参数**

struct mtd\_info \* mtd

mtd信息struct

struct nand\_chip \* chip

nand芯片信息struct

const uint8\_t \* buf

数据缓冲区

int oob\_required

必须写入chip->oob\_poi到OOB

int page

要写入的页码

**说明**

不适用于计算综合征的ECC控制器，这些控制器使用特殊的oob布局。

### int nand\_get\_set\_features\_notsupp（struct mtd\_info \* mtd，struct nand\_chip \* chip，int addr，u8 \* subfeature\_param）

设置/获取特征的存根，返回-ENOTSUPP

**参数**

struct mtd\_info \* mtd

MTD设备结构

struct nand\_chip \* chip

nand芯片信息struct

int addr

功能地址。

u8 \* subfeature\_param

子功能参数，四字节数组。

**说明**

应由不支持SET/GET FEATURES操作的NAND控制器驱动程序使用。

### int nand\_ecc\_choose\_conf（struct nand\_chip \* chip，const struct nand\_ecc\_caps \* caps，int oobavail）

设置ECC强度和ECC步长大小

**参数**

struct nand\_chip \* chip

nand芯片信息struct

const struct nand\_ecc\_caps \* caps

ECC引擎caps信息struct

int oobavail

ECC引擎可以使用的OOB大小

**说明**

根据以下逻辑选择ECC配置

如果已经设置了ECC步长大小和ECC强度（通常由DT设置），则检查是否受此控制器支持。

如果设置了NAND\_ECC\_MAXIMIZE，则选择最大的ECC强度。

否则，尝试将ECC步长大小和ECC强度与芯片的要求最接近的配对。如果可用的OOB大小无法适合芯片要求，则回退到最大的ECC步长大小和ECC强度。

成功后，设置所选的ECC设置。

### int nand\_scan\_with\_ids（struct mtd\_info \* mtd，int maxchips，struct nand\_flash\_dev \* ids）

[NAND接口]扫描NAND设备

**参数**

struct mtd\_info \* mtd

MTD设备结构

int maxchips

要扫描的芯片数量。如果此参数为零，则不运行nand\_scan\_ident()（对于必须自行处理此过程的特定驱动程序非常有用，例如docg4）。

### struct nand\_flash\_dev \* ids

可选的闪存ID表

**说明**

这将使用默认值填充所有未初始化的函数指针。读取闪存ID并使用适当的值填充mtd / chip结构。

### void nand\_cleanup（struct nand\_chip \* chip）

[NAND接口]释放NAND设备持有的资源

**参数**

struct nand\_chip \* chip

NAND芯片对象

### void nand\_release（struct mtd\_info \* mtd）

[NAND接口]取消注册MTD设备并释放NAND设备持有的资源

**参数**

struct mtd\_info \* mtd

MTD设备结构

### void \_\_nand\_calculate\_ecc（const unsigned char \* buf，unsigned int eccsize，unsigned char \* code）

[NAND接口]计算256/512字节块的3字节ECC

**参数**

const unsigned char \* buf

带有原始数据的输入缓冲区

unsigned int eccsize

每个ECC步骤的数据字节数（256或512）

unsigned char \* code

带有ECC的输出缓冲区

### int nand\_calculate\_ecc（struct mtd\_info \* mtd，const unsigned char \* buf，unsigned char \* code）

[NAND接口]计算256/512字节块的3字节ECC

**参数**

struct mtd\_info \* mtd

MTD块结构

const unsigned char \* buf

带有原始数据的输入缓冲区

unsigned char \* code

带有ECC的输出缓冲区

### int \_\_nand\_correct\_data(unsigned char \* buf, unsigned char \* read\_ecc, unsigned char \* calc\_ecc, unsigned int eccsize)

[NAND接口]检测和纠正位错误

**参数**

unsigned char \* buf

从芯片读取的原始数据

unsigned char \* read\_ecc

从芯片获取的ECC

unsigned char \* calc\_ecc

从原始数据计算的ECC

unsigned int eccsize

每个ECC步骤的数据字节数（256或512）

**说明**

检测并纠正eccsize字节块的1位错误

### int nand\_correct\_data（struct mtd\_info \* mtd，unsigned char \* buf，unsigned char \* read\_ecc，unsigned char \* calc\_ecc）

[NAND接口]检测和纠正位错误

**参数**

struct mtd\_info \* mtd

MTD块结构

unsigned char \* buf

从芯片读取的原始数据

unsigned char \* read\_ecc

从芯片获取的ECC

unsigned char \* calc\_ecc

从原始数据计算的ECC

**说明**

检测并更正256/512字节块中的1位误差

## 提供的内部函数

本章包含NAND驱动程序内部函数的自动生成文档。每个函数都有一个简短的说明，标有一个[XXX]标识符。参见章节“文档提示”以获取解释。标有[默认]标记的函数可能与板驱动程序开发人员相关。

### void nand\_release\_device(struct nand\_chip \*chip)

[通用]释放芯片

**参数**

struct nand\_chip \*chip

NAND芯片对象

**说明**

释放芯片锁并唤醒任何等待设备的人。

### uint8\_t nand\_read\_byte(struct mtd\_info \* mtd)

[默认]从芯片读取一个字节

**参数**

struct mtd\_info \* mtd

MTD设备结构

**说明**

默认的8位总线宽度读取函数

### uint8\_t nand\_read\_byte16(struct mtd\_info \* mtd)

[默认]从芯片读取一个字节的大小端感知

**参数**

struct mtd\_info \* mtd

MTD设备结构

**说明**

带有端点转换的16位总线宽度的默认读取函数。

### u16 nand\_read\_word(struct mtd\_info \* mtd)

[默认]从芯片读取一个字

**参数**

struct mtd\_info \* mtd

MTD设备结构

**说明**

16位总线宽度没有端点转换的默认读取函数。

### void nand\_select\_chip(struct mtd\_info \* mtd, int chipnr)

[默认]控制CE线

**参数**

struct mtd\_info \* mtd

MTD设备结构

int chipnr

要选择的芯片号，-1表示取消选择

**说明**

1芯片设备的默认选择函数。

void nand\_write\_byte(struct mtd\_info \* mtd, uint8\_t byte)

[默认]写入单个字节到芯片

**参数**

struct mtd\_info \* mtd

MTD设备结构

uint8\_t byte

要写入的值

**说明**

将字节写入I/O[7:0]的默认函数

### void nand\_write\_byte16(struct mtd\_info \* mtd, uint8\_t byte)

[默认]向16位宽芯片写入单个字节

**参数**

struct mtd\_info \* mtd

MTD设备结构

uint8\_t byte

要写入的值

**说明**

在16位宽芯片上将字节写入I/O[7:0]的默认函数。

### void nand\_write\_buf(struct mtd\_info \* mtd, const uint8\_t \* buf, int len)

[默认]将缓冲区写入芯片

**参数**

struct mtd\_info \* mtd

MTD设备结构

const uint8\_t \* buf

数据缓冲区

int len

要写入的字节数

**说明**

8位总线宽度的默认写入函数。

### void nand\_read\_buf(struct mtd\_info \* mtd, uint8\_t \* buf, int len)

[默认]将芯片数据读入缓冲区

**参数**

struct mtd\_info \* mtd

MTD设备结构

uint8\_t \* buf

要存储的数据缓冲区

int len

要读取的字节数

**说明**

8位总线宽度的默认读取函数。

### void nand\_write\_buf16(struct mtd\_info \* mtd, const uint8\_t \* buf, int len)

[默认]将缓冲区写入芯片

**参数**

struct mtd\_info \* mtd

MTD设备结构

const uint8\_t \* buf

数据缓冲区

int len

要写入的字节数

**说明**

16位总线宽度的默认写入函数。

### void nand\_read\_buf16(struct mtd\_info \* mtd, uint8\_t \* buf, int len)

[默认]将芯片数据读入缓冲区

**参数**

struct mtd\_info \* mtd

MTD设备结构

uint8\_t \* buf

要存储的数据缓冲区

int len

要读取的字节数

**说明**

16位总线宽度的默认读取函数。

### int nand\_block\_bad(struct mtd\_info \* mtd, loff\_t ofs)

[默认]从芯片读取坏块标记

**参数**

struct mtd\_info \* mtd

MTD设备结构

loff\_t ofs

设备开始的偏移量

**说明**

检查块是否损坏。

### int nand\_default\_block\_markbad(struct mtd\_info \* mtd, loff\_t ofs)

[默认]通过坏块标记将块标记为坏块

**参数**

struct mtd\_info \* mtd

MTD设备结构

loff\_t ofs

设备开始的偏移量

**说明**

这是默认实现，可以由硬件特定驱动程序覆盖。它提供了将坏块标记写入块的详细信息。

### int nand\_block\_markbad\_lowlevel(struct mtd\_info \* mtd, loff\_t ofs)

标记块为坏块

**参数**

struct mtd\_info \* mtd

MTD设备结构

loff\_t ofs

设备开始的偏移量

**说明**

此函数执行通用NAND坏块标记步骤（即，坏块表和/或标记）。我们仅允许硬件驱动程序指定如何将坏块标记写入OOB（chip->block\_markbad）。

我们按以下顺序尝试操作：

擦除受影响的块，以允许清洁地写入OOB标记

将坏块标记写入受影响块的OOB区域（除非存在标志NAND\_BBT\_NO\_OOB\_BBM）

更新BBT

请注意，我们保留（2）或（3）中遇到的第一个错误，完成过程并在最后转储错误。

### int nand\_check\_wp(struct mtd\_info \* mtd)

[通用]检查芯片是否为写保护

**参数**

struct mtd\_info \* mtd

MTD设备结构

**说明**

检查设备是否写保护。函数预期设备已被选择。

### int nand\_block\_isreserved(struct mtd\_info \* mtd, loff\_t ofs)

[通用]检查块是否标记为保留。

**参数**

struct mtd\_info \* mtd

MTD设备结构

loff\_t ofs

设备开始的偏移量

**说明**

检查该块是否标记为保留。

### int nand\_block\_checkbad(struct mtd\_info \* mtd, loff\_t ofs, int allowbbt)

[通用]检查块是否标记为坏块

**参数**

struct mtd\_info \* mtd

MTD 设备结构

loff\_t ofs

从设备开始的偏移量

int allowbbt

如果允许访问 bbt 区域，则为 1

**说明**

检查块是否损坏。通过读取坏块表或调用扫描函数来实现。

### void panic\_nand\_wait\_ready(struct mtd\_info \* mtd, unsigned long timeo)

[通用] 等待命令执行完后的就绪信号。

**参数**

struct mtd\_info \* mtd

MTD 设备结构

unsigned long timeo

超时时间

**说明**

当需要在中断上下文中等待时，使用 nand\_wait\_ready 的帮助函数。

### void nand\_wait\_status\_ready(struct mtd\_info \* mtd, unsigned long timeo)

[通用] 等待命令执行完后的状态就绪信号。

**参数**

struct mtd\_info \* mtd

MTD 设备结构

unsigned long timeo

超时时间（毫秒）

**说明**

等待就绪状态（即命令完成）或超时。

### void nand\_command(struct mtd\_info \* mtd, unsigned int command, int column, int page\_addr)

[默认] 发送命令到 NAND 设备

**参数**

struct mtd\_info \* mtd

MTD 设备结构

unsigned int command

要发送的命令

int column

此命令的列地址，如果没有则为 -1

int page\_addr

此命令的页地址，如果没有则为 -1

**说明**

将命令发送到 NAND 设备。该函数用于小页面设备（每页 512 个字节）。

### void nand\_command\_lp(struct mtd\_info \* mtd, unsigned int command, int column, int page\_addr)

[默认] 发送命令到 NAND 大页面设备。

**参数**

struct mtd\_info \* mtd

MTD 设备结构

unsigned int command

要发送的命令

int column

此命令的列地址，如果没有则为 -1

int page\_addr

此命令的页地址，如果没有则为 -1

**说明**

将命令发送到 NAND 设备。这是用于新大页面设备的版本。我们没有像小页面设备中那样的单独区域。我们必须模拟 NAND\_CMD\_READOOB 以保持代码兼容性。

### void panic\_nand\_get\_device(struct nand\_chip \* chip, struct mtd\_info \* mtd, int new\_state)

[通用] 获取所选访问的芯片

**参数**

struct nand\_chip \* chip

NAND 芯片说明符

struct mtd\_info \* mtd

MTD 设备结构

int new\_state

请求的状态

**说明**

当处于紧急状态时使用，不会取锁。

### int nand\_get\_device(struct mtd\_info \* mtd, int new\_state)

[通用] 获取所选访问的芯片

**参数**

struct mtd\_info \* mtd

MTD 设备结构

int new\_state

请求的状态

**说明**

获取设备并将其锁定以进行独占访问。

### void panic\_nand\_wait(struct mtd\_info \* mtd, struct nand\_chip \* chip, unsigned long timeo)

[通用] 等待命令完成

**参数**

struct mtd\_info \* mtd

MTD 设备结构

struct nand\_chip \* chip

NAND 芯片结构

unsigned long timeo

超时时间

**说明**

等待命令完成。这是 nand\_wait 的帮助函数，用于在中断上下文中等待时。可能会在紧急情况下发生，当试图通过 mtdoops 写入 oops 时。

### int nand\_wait(struct mtd\_info \* mtd, struct nand\_chip \* chip)

[默认] 等待命令完成

**参数**

struct mtd\_info \* mtd

MTD 设备结构

struct nand\_chip \* chip

NAND 芯片结构

**说明**

等待命令完成。这仅适用于擦除和编程。

### int nand\_reset\_data\_interface(struct nand\_chip \* chip, int chipnr)

重置数据接口和时序

**参数**

struct nand\_chip \* chip

NAND 芯片

int chipnr

内部芯片 ID

**说明**

将数据接口和时序重置为 ONFI 模式 0。

如果成功，则返回 0；否则返回负错误代码。

### int nand\_setup\_data\_interface(struct nand\_chip \* chip, int chipnr)

设置最佳数据接口和时序

**参数**

struct nand\_chip \* chip

NAND 芯片

int chipnr

内部芯片 ID

**说明**

查找并配置芯片和驱动程序支持的最佳数据接口和 NAND 时序。首先尝试从 ONFI 信息中检索支持的时序模式，如果 NAND 芯片不支持 ONFI，则依赖于 nand\_ids 表中指定的->onfi\_timing\_mode\_default。

如果成功，则返回 0；否则返回负错误代码。

### int nand\_init\_data\_interface(struct nand\_chip \* chip)

查找最佳数据接口和时序

**参数**

struct nand\_chip \* chip

NAND 芯片

**说明**

查找并配置芯片和驱动程序支持的最佳数据接口和 NAND 时序。首先尝试从 ONFI 信息中检索支持的时序模式，如果 NAND 芯片不支持 ONFI，则依赖于 nand\_ids 表中指定的->onfi\_timing\_mode\_default。在此函数后，nand\_chip->data\_interface 将初始化为可用的最佳时序模式。

如果成功，则返回 0；否则返回负错误代码。

### int nand\_fill\_column\_cycles(struct nand\_chip \* chip, u8 \* addrs, unsigned int offset\_in\_page)

填充地址的列周期

**参数**

struct nand\_chip \* chip

NAND 芯片

u8 \* addrs

要填充的地址周期数组

unsigned int offset\_in\_page

页面中的偏移量

**说明**

根据 NAND 总线宽度和页面大小填充 addrs 字段的前两个字节或第一个字节。

如果有任何一个参数无效，则返回编码列所需的周期数，或者返回负错误代码。

### int nand\_read\_param\_page\_op(struct nand\_chip \*chip, u8 page, void \*buf, unsigned int len)

执行READ PARAMETER PAGE操作

**参数**

struct nand\_chip \*chip

NAND芯片

u8 page

要读取的参数页

void \*buf

用于存储数据的缓冲区

unsigned int len

缓冲区的长度

说明

此函数发出READ PARAMETER PAGE操作。此函数不选择/取消选择CS线。

成功返回0，否则返回负错误代码。

### int nand\_set\_features\_op(struct nand\_chip \*chip, u8 feature, const void \*data)

执行SET FEATURES操作

**参数**

struct nand\_chip \*chip

NAND芯片

u8 feature

功能标识符

const void \*data

4个字节的数据

**说明**

此函数发送SET FEATURES命令，并等待NAND准备就绪后返回。此函数不选择/取消选择CS线。

成功返回0，否则返回负错误代码。

### int nand\_get\_features\_op(struct nand\_chip \*chip, u8 feature, void \*data)

执行GET FEATURES操作

**参数**

struct nand\_chip \*chip

NAND芯片

u8 feature

功能标识符

void \*data

4个字节的数据

**说明**

此函数发送GET FEATURES命令，并等待NAND准备就绪后返回。此函数不选择/取消选择CS线。

成功返回0，否则返回负错误代码。

### struct nand\_op\_parser\_ctx

解析器使用的上下文

**定义**

struct nand\_op\_parser\_ctx {

const struct nand\_op\_instr \*instrs;

unsigned int ninstrs;

struct nand\_subop subop;

};

**成员**

instrs

必须处理的所有指令数组

ninstrs

instrs数组的长度

subop

要传递给NAND控制器的子操作

**说明**

此结构由核心用于将NAND操作拆分为可以由NAND控制器处理的子操作。

### bool nand\_op\_parser\_must\_split\_instr(const struct nand\_op\_parser\_pattern\_elem \*pat, const struct nand\_op\_instr \*instr, unsigned int \*start\_offset)

检查是否必须拆分指令

**参数**

const struct nand\_op\_parser\_pattern\_elem \*pat

与instr匹配的解析器模式元素

const struct nand\_op\_instr \*instr

要检查的指令指针

unsigned int \*start\_offset

这是一个传入/传出参数。如果instr已经被拆分，则start\_offset是要从哪里开始的偏移量（可以是地址周期或数据缓冲区中的偏移量）。相反，如果函数返回true（即instr必须拆分），则更新此参数以指向尚未处理的第一个数据/地址周期。

**说明**

某些NAND控制器受限，无法使用唯一的操作发送X个地址周期，或无法同时读取/写入超过Y个字节。在这种情况下，将不能适应单个控制器操作的指令拆分为两个或更多块。

如果必须拆分指令，则返回true，否则返回false。还会将start\_offset参数更新为开始下一批指令的偏移量（如果是地址或数据指令）。

### bool nand\_op\_parser\_match\_pat(const struct nand\_op\_parser\_pattern \*pat, struct nand\_op\_parser\_ctx \*ctx)

检查模式是否与解析器上下文中剩余的指令匹配

**参数**

const struct nand\_op\_parser\_pattern \*pat

要测试的模式

struct nand\_op\_parser\_ctx \*ctx

要与模式pat匹配的解析器上下文结构

**说明**

检查模式pat是否与剩余的ctx中的一组或子组的指令匹配。如果是，则返回true，否则返回false。当返回true时，ctx->subop会更新为要传递给控制器驱动程序的指令集。

### int nand\_check\_erased\_buf(void \*buf, int len, int bitflips\_threshold)

检查缓冲区是否仅包含（几乎）所有0xff数据

**参数**

void \*buf

要测试的缓冲区

int len

缓冲区长度

int bitflips\_threshold

最大位翻转次数

**说明**

检查缓冲区是否仅包含0xff，这意味着底层区域已经擦除并准备好进行编程。bitflips\_threshold指定位翻转次数，超过此次数就认为该区域未擦除。

注意

此函数的逻辑已从memweight实现中提取，但nand\_check\_erased\_buf函数会在比特翻转次数超过bitflips\_threshold值时退出测试整个缓冲区。

返回小于或等于bitflips\_threshold的正数位翻转次数，或-ERROR\_CODE表示超过阈值的位翻转次数。

### int nand\_read\_page\_raw\_syndrome(struct mtd\_info \*mtd, struct nand\_chip \*chip, uint8\_t \*buf, int oob\_required, int page)

[INTERN]读取没有ECC的原始页面数据

**参数**

struct mtd\_info \*mtd

mtd信息结构

struct nand\_chip \*chip

nand芯片信息结构

uint8\_t \*buf

存储读取数据的缓冲区

int oob\_required

调用者需要OOB数据读取到chip->oob\_poi

int page

要读取的页码

**说明**

即使不使用OOB，我们仍需要特殊的OOB布局和处理。

### int nand\_read\_page\_swecc(struct mtd\_info \*mtd, struct nand\_chip \*chip, uint8\_t \*buf, int oob\_required, int page)

[可替换]基于软件ECC的页面读取函数

**参数**

struct mtd\_info \* mtd

mtd信息struct

struct nand\_chip \* chip

nand芯片信息struct

uint8\_t \* buf

用于存储读取数据的缓冲区

int oob\_required

调用者需要从chip->oob\_poi读取OOB数据

int page

要读取的页面编号

### int nand\_read\_subpage(struct mtd\_info \* mtd，struct nand\_chip \* chip，uint32\_t data\_offs，uint32\_t readlen，uint8\_t \* bufpoi，int page)

[可替换]基于ECC的子页面读取功能

**参数**

struct mtd\_info \* mtd

mtd信息struct

struct nand\_chip \* chip

nand芯片信息struct

uint32\_t data\_offs

所请求数据在页面内的偏移量

uint32\_t readlen

数据长度

uint8\_t \* bufpoi

用于存储读取数据的缓冲区

int page

要读取的页面编号

### int nand\_read\_page\_hwecc(struct mtd\_info \* mtd，struct nand\_chip \* chip，uint8\_t \* buf，int oob\_required，int page)

[可替换]硬件ECC基于页面读取功能

**参数**

struct mtd\_info \* mtd

mtd信息struct

struct nand\_chip \* chip

nand芯片信息struct

uint8\_t \* buf

用于存储读取数据的缓冲区

int oob\_required

调用者需要从chip->oob\_poi读取OOB数据

int page

要读取的页面编号

**说明**

不适用于需要特殊oob布局的综合计算ECC控制器。

### int nand\_read\_page\_hwecc\_oob\_first(struct mtd\_info \* mtd，struct nand\_chip \* chip，uint8\_t \* buf，int oob\_required，int page)

[可替换] hw ecc，先读取oob

**参数**

struct mtd\_info \* mtd

mtd信息struct

struct nand\_chip \* chip

nand芯片信息struct

uint8\_t \* buf

用于存储读取数据的缓冲区

int oob\_required

调用者需要从chip->oob\_poi读取OOB数据

int page

要读取的页面编号

**说明**

Hardware ECC用于大页面芯片，需要首先读取OOB。对于该ECC模式，write\_page方法从ECC\_HW中重新使用。这些方法从OOB区域读/写ECC，不像具有多个ECC步骤的ECC\_HW\_SYNDROME支持遵循“中缀ECC”方案，并从数据区读/写ECC，通过覆盖NAND制造商坏块标记。

### int nand\_read\_page\_syndrome(struct mtd\_info \* mtd，struct nand\_chip \* chip，uint8\_t \* buf，int oob\_required，int page)

[可替换]基于硬件ECC综合的页面读取

**参数**

struct mtd\_info \* mtd

mtd信息struct

struct nand\_chip \* chip

nand芯片信息struct

uint8\_t \* buf

用于存储读取数据的缓冲区

int oob\_required

调用者需要从chip->oob\_poi读取OOB数据

int page

要读取的页面编号

**说明**

hw生成器自动计算错误综合。因此，我们需要特殊的oob布局和处理。

### uint8\_t \* nand\_transfer\_oob(struct mtd\_info \* mtd，uint8\_t \* oob，struct mtd\_oob\_ops \* ops，size\_t len)

[INTERN]将oob传输到客户端缓冲区

**参数**

struct mtd\_info \* mtd

mtd信息struct

uint8\_t \* oob

oob目标地址

struct mtd\_oob\_ops \* ops

oob操作struct

size\_t len

要传输的oob的大小

### int nand\_setup\_read\_retry(struct mtd\_info \* mtd，int retry\_mode)

[INTERN]设置READ RETRY模式

**参数**

struct mtd\_info \* mtd

MTD设备struct

int retry\_mode

要使用的重试模式

**说明**

一些供应商提供了一个特殊的命令来移动Vt阈值，用于在页面中有太多的位翻转（即，ECC错误）时使用。设置新阈值后，主机应重试读取页面。

### int nand\_do\_read\_ops(struct mtd\_info \* mtd，loff\_t from，struct mtd\_oob\_ops \* ops)

[INTERN]使用ECC读取数据

**参数**

struct mtd\_info \* mtd

MTD设备结构

loff\_t from

要读取的偏移量

struct mtd\_oob\_ops \* ops

oob opsstruct

**说明**

内部函数。使用芯片保持调用。

### int nand\_do\_read\_oob(struct mtd\_info \* mtd，loff\_t from，struct mtd\_oob\_ops \* ops)

[INTERN]NAND读取外带

**参数**

struct mtd\_info \* mtd

MTD设备结构

loff\_t from

要读取的偏移量

struct mtd\_oob\_ops \* ops

oob操作说明结构

**说明**

从备用区域读取NAND外带数据。

### int nand\_read\_oob(struct mtd\_info \* mtd，loff\_t from，struct mtd\_oob\_ops \* ops)

[MTD接口]NAND读取数据和/或外带数据

**参数**

struct mtd\_info \* mtd

MTD设备结构

loff\_t from

要读取的偏移量

struct mtd\_oob\_ops \* ops

oob操作说明结构

**说明**

NAND读取数据和/或外带数据。

### int nand\_write\_page\_raw\_syndrome(struct mtd\_info \* mtd，struct nand\_chip \* chip，const uint8\_t \* buf，int oob\_required，int page)

[INTERN]原始页面写入功能

**参数**

struct mtd\_info \* mtd

mtd信息struct

struct nand\_chip \* chip

nand芯片信息struct

const uint8\_t \* buf

数据缓冲区

int oob\_required

必须将chip->oob\_poi写入OOB

int page

要写入的页面编号

**说明**

即使不检查ECC，我们仍需要特殊的oob布局和处理。

### int nand\_write\_page\_swecc(struct mtd\_info \* mtd，struct nand\_chip \* chip，const uint8\_t \* buf，int oob\_required，int page)

[可替换]基于软件ECC的页面写入功能

**参数**

struct mtd\_info \* mtd

mtd信息struct

struct nand\_chip \* chip

nand芯片信息struct

const uint8\_t \* buf

数据缓冲区

int oob\_required

必须将chip->oob\_poi写入OOB

int page

要写入的页面编号

### int nand\_write\_page\_hwecc(struct mtd\_info \* mtd，struct nand\_chip \* chip，const uint8\_t \* buf，int oob\_required，int page)

[可替换]基于硬件ECC的页面写入功能

**参数**

struct mtd\_info \* mtd

mtd信息struct

struct nand\_chip \* chip

NAND芯片信息结构

const uint8\_t \* buf

数据缓冲区

int oob\_required

必须将chip->oob\_poi写入OOB

int page

要写入的页面号

### int nand\_write\_subpage\_hwecc（struct mtd\_info \* mtd，struct nand\_chip \* chip，uint32\_t offset，uint32\_t data\_len，const uint8\_t \* buf，int oob\_required，int page）

[可替换]基于硬件ECC的子页面写入

**参数**

struct mtd\_info \* mtd

MTD信息结构

struct nand\_chip \* chip

NAND芯片信息结构

uint32\_t offset

页面内子页面的列地址

uint32\_t data\_len

数据长度

const uint8\_t \* buf

数据缓冲区

int oob\_required

必须将chip->oob\_poi写入OOB

int page

要写入的页面号

### int nand\_write\_page\_syndrome（struct mtd\_info \* mtd，struct nand\_chip \* chip，const uint8\_t \* buf，int oob\_required，int page）

[可替换]基于硬件ECC综合症的页面写入

**参数**

struct mtd\_info \* mtd

MTD信息结构

struct nand\_chip \* chip

NAND芯片信息结构

const uint8\_t \* buf

数据缓冲区

int oob\_required

必须将chip->oob\_poi写入OOB

int page

要写入的页面号

**说明**

硬件生成器会自动计算错误综合症。因此我们需要特殊的oob布局和处理。

### int nand\_write\_page（struct mtd\_info \* mtd，struct nand\_chip \* chip，uint32\_t offset，int data\_len，const uint8\_t \* buf，int oob\_required，int page，int raw）

写入一个页面

**参数**

struct mtd\_info \* mtd

MTD设备结构

struct nand\_chip \* chip

NAND芯片说明符

uint32\_t offset

页面内偏移量

int data\_len

实际要写入的数据长度

const uint8\_t \* buf

要写入的数据

int oob\_required

必须将chip->oob\_poi写入OOB

int page

要写入的页面号

int raw

使用写入页面的\_raw版本

### uint8\_t \* nand\_fill\_oob（struct mtd\_info \* mtd，uint8\_t \* oob，size\_t len，struct mtd\_oob\_ops \* ops）

[INTERN]将客户端缓冲区传输到oob中

**参数**

struct mtd\_info \* mtd

MTD信息结构

uint8\_t \* oob

oob数据缓冲区

size\_t len

oob数据写入长度

struct mtd\_oob\_ops \* ops

oob操作结构

### int nand\_do\_write\_ops（struct mtd\_info \* mtd，loff\_t to，struct mtd\_oob\_ops \* ops）

[INTERN] NAND带ECC的写入

**参数**

struct mtd\_info \* mtd

MTD信息结构

loff\_t to

要写入的偏移量

struct mtd\_oob\_ops \* ops

oob操作说明结构

**说明**

具有ECC的NAND写入。

### int panic\_nand\_write（struct mtd\_info \* mtd，loff\_t to，size\_t len，size\_t \* retlen，const uint8\_t \* buf）

[MTD接口] NAND带ECC的写入

**参数**

struct mtd\_info \* mtd

MTD信息结构

loff\_t to

要写入的偏移量

size\_t len

要写入的字节数

size\_t \* retlen

指向存储写入字节数的变量的指针

const uint8\_t \* buf

要写入的数据

**说明**

带ECC的NAND写入。当在中断上下文中执行写操作时使用，例如在紧急情况下写入oops可能会调用此函数。

### int nand\_do\_write\_oob（struct mtd\_info \* mtd，loff\_t to，struct mtd\_oob\_ops \* ops）

[MTD接口] NAND写OOB

**参数**

struct mtd\_info \* mtd

MTD信息结构

loff\_t to

要写入的偏移量

struct mtd\_oob\_ops \* ops

oob操作说明结构

**说明**

NAND写OOB。

### int nand\_write\_oob（struct mtd\_info \* mtd，loff\_t to，struct mtd\_oob\_ops \* ops）

[MTD接口] NAND写数据和/或OOB

**参数**

struct mtd\_info \* mtd

MTD信息结构

loff\_t to

要写入的偏移量

struct mtd\_oob\_ops \* ops

oob操作说明结构

### int single\_erase（struct mtd\_info \* mtd，int page）

[通用] NAND标准块擦除命令函数

**参数**

struct mtd\_info \* mtd

MTD信息结构

int page

要擦除的块的页面地址

**说明**

用于NAND芯片的标准擦除命令。返回NAND状态。

### int nand\_erase（struct mtd\_info \* mtd，struct erase\_info \* instr）

[MTD接口]擦除块

**参数**

struct mtd\_info \* mtd

MTD信息结构

struct erase\_info \* instr

擦除指令

**说明**

擦除一个或多个块。

### int nand\_erase\_nand（struct mtd\_info \* mtd，struct erase\_info \* instr，int allowbbt）

[INTERN]擦除块

**参数**

struct mtd\_info \* mtd

MTD信息结构

struct erase\_info \* instr

擦除指令

int allowbbt

允许擦除bbt区域

**说明**

擦除一个或多个块。

### void nand\_sync（struct mtd\_info \* mtd）

[MTD接口]同步

**参数**

struct mtd\_info \* mtd

MTD信息结构

**说明**

同步实际上是等待芯片准备好的函数。

### int nand\_block\_isbad（struct mtd\_info \* mtd，loff\_t offs）

[MTD接口]检查偏移处的块是否损坏

**参数**

struct mtd\_info \* mtd

MTD信息结构

loff\_t offs

相对于MTD开始的偏移量

### int nand\_block\_markbad（struct mtd\_info \* mtd，loff\_t ofs）

[MTD接口]将给定偏移处的块标记为坏块

**参数**

struct mtd\_info \* mtd

MTD信息结构

loff\_t ofs

相对于MTD开始的偏移量

### int nand\_max\_bad\_blocks（struct mtd\_info \* mtd，loff\_t ofs，size\_t len）

[MTD接口]一个MTD的最大坏块数

**参数**

struct mtd\_info \* mtd

MTD信息结构

loff\_t ofs

相对于MTD开始的偏移量

size\_t len

MTD的长度

### int nand\_default\_set\_features（struct mtd\_info \* mtd，struct nand\_chip \* chip，int addr，uint8\_t \* subfeature\_param）

[可替换]设置NAND芯片功能

**参数**

struct mtd\_info \* mtd

MTD信息结构

struct nand\_chip \* chip

NAND芯片信息结构

int addr

功能地址。

uint8\_t \* subfeature\_param

子功能参数，一个四字节数组。

### int nand\_default\_get\_features（struct mtd\_info \* mtd，struct nand\_chip \* chip，int addr，uint8\_t \* subfeature\_param）

[可替换]获取NAND芯片功能

**参数**

struct mtd\_info \* mtd

MTD设备结构

struct nand\_chip \* chip

nand芯片信息结构

int addr

功能地址。

uint8\_t \* subfeature\_param

子功能**参数**，一个四字节数组。

### int nand\_suspend(struct mtd\_info \* mtd)

[MTD接口]暂停NAND闪存

**参数**

struct mtd\_info \* mtd

MTD设备结构

### void nand\_resume（struct mtd\_info \* mtd）

[MTD接口]恢复NAND闪存

**参数**

struct mtd\_info \* mtd

MTD设备结构

void nand\_shutdown(struct mtd\_info \* mtd)

[MTD接口]完成当前NAND操作并防止进一步操作

**参数**

struct mtd\_info \* mtd

MTD设备结构

### int nand\_scan\_ident(struct mtd\_info \* mtd，int maxchips，struct nand\_flash\_dev \* table)

扫描NAND设备

**参数**

struct mtd\_info \* mtd

MTD设备结构

int maxchips

要扫描的芯片数

struct nand\_flash\_dev \*表

替代NAND ID表

**说明**

这是正常的nand\_scan（）函数的第一阶段。它读取闪存ID并相应地设置MTD字段。

这个帮助程序曾经直接从控制器驱动程序中调用，这些驱动程序需要调整一些与ECC相关的**参数**，然后才能调用nand\_scan\_tail（）。这种分离防止了在此阶段进行动态分配，这是不方便的，因此已被禁止，以发挥->：c：func：init\_ecc（）/ cleanup\_ecc（）的好处。

### int nand\_check\_ecc\_caps（struct nand\_chip \* chip，const struct nand\_ecc\_caps \* caps，int oobavail）

检查预设ECC设置的健全性

**参数**

struct nand\_chip \* chip

nand芯片信息结构

const struct nand\_ecc\_caps \* caps

ECC caps信息结构

int oobavail

ECC引擎可以使用的OOB大小

**说明**

当ECC步长和强度已经设置时，检查它们是否得到控制器的支持，并且计算的ECC字节是否适合芯片的OOB。成功时，设置计算的ECC字节。

### int nand\_match\_ecc\_req（struct nand\_chip \* chip，const struct nand\_ecc\_caps \* caps，int oobavail）

尽可能少使用ECC字节来满足芯片的要求

**参数**

struct nand\_chip \* chip

nand芯片信息结构

const struct nand\_ecc\_caps \* caps

ECC引擎caps信息结构

int oobavail

ECC引擎可以使用的OOB大小

**说明**

如果提供了芯片的ECC要求，请尝试使用最少量的ECC字节（即使用最多的无OOB字节）来满足它。成功时，设置所选的ECC设置。

### int nand\_maximize\_ecc（struct nand\_chip \* chip，const struct nand\_ecc\_caps \* caps，int oobavail）

选择可用的最大ECC强度

**参数**

struct nand\_chip \* chip

nand芯片信息结构

const struct nand\_ecc\_caps \* caps

ECC引擎caps信息结构

int oobavail

ECC引擎可以使用的OOB大小

**说明**

选择控制器支持的最大ECC强度，并且可以适合芯片的OOB。成功时，设置所选的ECC设置。

### int nand\_scan\_tail(struct mtd\_info \* mtd)

扫描NAND设备

**参数**

struct mtd\_info \* mtd

MTD设备结构

**说明**

这是正常的nand\_scan（）函数的第二阶段。它用默认值填充所有未初始化的函数指针，并在适当的情况下扫描坏块表。

### int check\_pattern(uint8\_t \* buf，int len，int paglen，struct nand\_bbt\_descr \* td)

[通用]检查缓冲区中是否有模式

**参数**

uint8\_t \* buf

要搜索的缓冲区

int len

要搜索的缓冲区的长度

int paglen

页面长度

struct nand\_bbt\_descr \* td

搜索模式**说明**符

**说明**

在给定位置检查模式。用于搜索坏块表和好/坏块标识符。

### int check\_short\_pattern(uint8\_t \* buf，struct nand\_bbt\_descr \* td)

[通用]检查缓冲区中是否有模式

**参数**

uint8\_t \* buf

要搜索的缓冲区

struct nand\_bbt\_descr \* td

搜索模式**说明**符

**说明**

在给定位置检查模式。用于搜索坏块表和好/坏块标识符。与check\_pattern相同，但没有可选的空检查。

### u32 add\_marker\_len（struct nand\_bbt\_descr \* td）

计算数据区标记的长度

**参数**

struct nand\_bbt\_descr \* td

用于计算的BBT说明符

**说明**

如果标记位于OOB区域，则长度为0。

### int read\_bbt(struct mtd\_info \* mtd，uint8\_t \* buf，int page，int num，struct nand\_bbt\_descr \* td，int offs)

[通用]从页开始读取坏块表

**参数**

struct mtd\_info \* mtd

MTD设备结构

uint8\_t \* buf

临时缓冲区

int page

起始页码

int num

要读取的bbt说明符数

struct nand\_bbt\_descr \* td

bbt说明符表

int offs

表中的块号偏移量

**说明**

从页面开始读取坏块表。

### int read\_abs\_bbt(struct mtd\_info \* mtd，uint8\_t \* buf，struct nand\_bbt\_descr \* td，int chip)

[通用]从给定的页面开始读取坏块表

**参数**

struct mtd\_info \* mtd

MTD设备结构

uint8\_t \* buf

临时缓冲区

struct nand\_bbt\_descr\* td

坏块表的说明符

int chip

读取特定芯片的表格，-1读取所有芯片;仅适用于设置了NAND\_BBT\_PERCHIP选项的情况

**说明**

从给定的页面开始读取所有芯片的坏块表。我们假设bbt位按顺序连续排列。

### int scan\_read\_oob(struct mtd\_info\* mtd，uint8\_t\* buf，loff\_t offs，size\_t len)

[通用]扫描数据+OOB区域到缓冲器

**参数**

struct mtd\_info\* mtd

MTD设备结构

uint8\_t\* buf

临时缓冲区

loff\_t offs

扫描的偏移量

size\_t len

要读取的数据区域长度

**说明**

从数据+OOB中读取扫描读取的数据。可能遍历多个页面，在缓冲器中交错页面，OOB，页面，OOB，... 完成传输并返回“最强”纠错码条件（错误或位翻转）。可能在第一个（非ECC）错误时退出。

### void read\_abs\_bbts(struct mtd\_info\* mtd，uint8\_t\* buf，struct nand\_bbt\_descr\* td，struct nand\_bbt\_descr\* md)

[通用]从给定页面开始读取所有芯片的坏块表格（s）

**参数**

struct mtd\_info\* mtd

MTD设备结构

uint8\_t\* buf

临时缓冲区

struct nand\_bbt\_descr\* td

坏块表的说明符

struct nand\_bbt\_descr\* md

坏块表的说明符镜像

**说明**

从给定页面开始读取所有芯片的坏块表（s）。我们假设bbt位按顺序连续排列。

### int create\_bbt(struct mtd\_info\* mtd，uint8\_t\* buf，struct nand\_bbt\_descr\* bd，int chip)

[通用]通过扫描设备创建一个坏块表

**参数**

struct mtd\_info\* mtd

MTD设备结构

uint8\_t\* buf

临时缓冲区

struct nand\_bbt\_descr\* bd

用于好/坏块搜索模式的说明符

int chip

创建特定芯片的表，-1读取所有芯片;仅适用于设置了NAND\_BBT\_PERCHIP选项的情况

**说明**

通过扫描设备来创建一个坏块表，以获取给定的好/坏块标识模式。

### int search\_bbt(struct mtd\_info\* mtd，uint8\_t\* buf，struct nand\_bbt\_descr\* td)

[通用]扫描设备以获取特定的坏块表

**参数**

struct mtd\_info\* mtd

MTD设备结构

uint8\_t\* buf

临时缓冲区

struct nand\_bbt\_descr\* td

坏块表的说明符

**说明**

通过搜索给定的标识模式来读取坏块表。搜索从设备的开头向上或从末尾向下进行。搜索始终从块的开头开始。如果给出选项NAND\_BBT\_PERCHIP，则会为每个芯片搜索一个bbt，其中包含该芯片的坏块信息。这是为某些DOC设备提供支持所必需的。

bbt标识模式驻留在块中第一页的oob区域中。

### void search\_read\_bbts(struct mtd\_info\* mtd，uint8\_t\* buf，struct nand\_bbt\_descr\* td，struct nand\_bbt\_descr\* md)

[通用]扫描设备以获取坏块表格

**参数**

struct mtd\_info\* mtd

MTD设备结构

uint8\_t\* buf

临时缓冲区

struct nand\_bbt\_descr\* td

坏块表的说明符

struct nand\_bbt\_descr\* md

坏块表的说明符镜像

**说明**

搜索和读取坏块表（s）。

### int get\_bbt\_block(struct nand\_chip\* this，struct nand\_bbt\_descr\* td，struct nand\_bbt\_descr\* md，int chip)

获取第一个有效的擦除块，适合存储BBT

**参数**

struct nand\_chip\* this

NAND设备

struct nand\_bbt\_descr\* td

BBT说明

struct nand\_bbt\_descr\* md

mirror BBT说明符

int chip

芯片选择器

**说明**

此函数返回一个正的块号，指向有效的擦除块，适合存储BBT（即在保留给BBT的范围内），或者如果所有块已经被使用或标记为坏，则为-ENOSPC。如果td->pages [chip]已经指向一个有效的块，我们将重新使用它，否则我们将搜索下一个有效的块。

### void mark\_bbt\_block\_bad(struct nand\_chip\* this，struct nand\_bbt\_descr\* td，int chip，int block)

将保留给BBT的块之一标记为坏

**参数**

struct nand\_chip\* this

NAND设备

struct nand\_bbt\_descr\* td

BBT说明

int chip

芯片选择器

int block

要标记的BBT块

**说明**

为BBT保留的块可能会变坏。这个函数是标记这样的块为坏的助手。它会更新内存中的BBT，使用坏块标记标记块为坏，并使关联的td->pages []条目无效。

### int write\_bbt(struct mtd\_info\* mtd，uint8\_t\* buf，struct nand\_bbt\_descr\* td，struct nand\_bbt\_descr\* md，int chipsel)

[通用]（重新）写入坏块表

**参数**

struct mtd\_info\* mtd

MTD设备结构

uint8\_t\* buf

临时缓冲区

struct nand\_bbt\_descr\* td

坏块表的说明符

struct nand\_bbt\_descr\* md

坏块表的说明符镜像

int chipsel

选择特定芯片，-1的所有

**说明**

（重新）写入坏块表。

### int nand\_memory\_bbt(struct mtd\_info\* mtd，struct nand\_bbt\_descr\* bd)

[通用]创建基于内存的坏块表格

**参数**

struct mtd\_info \* mtd

MTD 设备结构

struct nand\_bbt\_descr \* bd

**说明**好/坏块搜索模式

**说明**

该函数通过扫描设备的制造商/软件标记的好/坏块来创建基于内存的 bbt。

### int check\_create(struct mtd\_info \* mtd, uint8\_t \* buf, struct nand\_bbt\_descr \* bd)

[GENERIC] 创建和写入 bbt(s) 如有必要。

**参数**

struct mtd\_info \* mtd

MTD 设备结构

uint8\_t \* buf

临时缓冲区

struct nand\_bbt\_descr \* bd

说明好/坏块搜索模式

**说明**

该函数检查之前读取 bbt 的结果，并在必要时创建/更新 bbt。如果芯片/设备没有找到 bbt，则需要创建 bbt。如果缺少其中一个表或者其中一个表的版本号小于另一个，则需要更新。

### void mark\_bbt\_region(struct mtd\_info \* mtd, struct nand\_bbt\_descr \* td)

[GENERIC] 标记坏块表区域

**参数**

struct mtd\_info \* mtd

MTD 设备结构

struct nand\_bbt\_descr \* td

坏块表说明符

**说明**

将坏块表区域标记为“坏”，以防止操作的意外擦除/写入。这些区域由标记 0x02 标识。

### void verify\_bbt\_descr(struct mtd\_info \* mtd, struct nand\_bbt\_descr \* bd)

验证坏**块说明符**

**参数**

struct mtd\_info \* mtd

MTD 设备结构

struct nand\_bbt\_descr \* bd

要验证的表

**说明**

此函数对坏块说明表执行几个合理性检查。

### int nand\_scan\_bbt(struct mtd\_info \* mtd, struct nand\_bbt\_descr \* bd)

[NAND 接口] 扫描、查找、读取并可能创建坏块表。

**参数**

struct mtd\_info \* mtd

MTD 设备结构

struct nand\_bbt\_descr \* bd

说明好/坏块搜索模式

**说明**

该函数检查坏块表是否已经可用。如果没有，则扫描设备以查找制造商标记的好/坏块，并将坏块表写入选定的位置。

坏块表内存在此处被分配。必须调用 nand\_free\_bbt 函数来释放它。

### int nand\_update\_bbt(struct mtd\_info \* mtd, loff\_t offs)

更新坏块表

**参数**

struct mtd\_info \* mtd

MTD 设备结构

loff\_t offs

新标记块的偏移量

**说明**

此函数更新坏块表。

### int nand\_create\_badblock\_pattern(struct nand\_chip \* this)

[INTERN] 创建 bbt 说明符结构

**参数**

struct nand\_chip \* this

用于创建**说明**符的 NAND 芯片

**说明**

此函数根据 this 的属性分配和初始化用于 BBM 检测的 nand\_bbt\_descr。新**说明**符存储在 this->badblock\_pattern 中。因此，在传递给此函数时，this->badblock\_pattern 应为 NULL。

### int nand\_isreserved\_bbt(struct mtd\_info \* mtd, loff\_t offs)

[NAND 接口] 检查块是否已预留

**参数**

struct mtd\_info \* mtd

MTD 设备结构

loff\_t offs

设备中的偏移量

### int nand\_isbad\_bbt(struct mtd\_info \* mtd, loff\_t offs, int allowbbt)

[NAND 接口] 检查块是否为坏块

**参数**

struct mtd\_info \* mtd

MTD 设备结构

loff\_t offs

设备中的偏移量

int allowbbt

允许访问坏块表区域

### int nand\_markbad\_bbt(struct mtd\_info \* mtd, loff\_t offs)

[NAND 接口] 在 BBT 中标记坏块

**参数**

struct mtd\_info \* mtd

MTD 设备结构

loff\_t offsS

标记的坏块的偏移量

## 贡献者

以下人员为 NAND 驱动程序做出了贡献：

Steven J. Hillsjhill@realitydiluted.com

David Woodhousedwmw2@infradead.org

Thomas Gleixnertglx@linutronix.de

许多用户为修复漏洞、改进和测试提供了帮助。非常感谢。

以下人员为本文档做出了贡献：

Thomas Gleixner[tglx@linutronix.de](mailto:tglx@linutronix.de)

# 并行端口设备

## int parport\_yield(struct pardevice \*dev)

暂时放弃并口

**参数**

struct pardevice \*dev

并口上的设备

**说明**

如果放弃端口对其他驱动程序有帮助，该函数将放弃端口。之后，它尝试使用 parport\_claim() 恢复端口，并且返回与 parport\_claim() 相同。如果失败，则会保留未声明的端口，驱动程序有责任恢复端口。

parport\_yield() 和 parport\_yield\_blocking() 函数用于标记驱动程序中的点，其他驱动程序可以在该点声称端口并使用其设备。放弃端口类似于释放和恢复端口，但效率更高，因为如果没有其他需要端口的设备，不会采取任何措施。实际上，即使有其他设备等待但当前设备仍在其“时间片”内，也不会执行任何操作。默认的时间片为半秒钟，但可以通过 /proc 接口进行调整。

## int parport\_yield\_blocking(struct pardevice \*dev)

暂时放弃并口

**参数**

struct pardevice \*dev

并口上的设备

**说明**

如果放弃端口对其他驱动程序有帮助，该函数将放弃端口。之后，它尝试使用 parport\_claim\_or\_block() 恢复端口，并且返回与 parport\_claim\_or\_block() 相同。

## int parport\_wait\_event（struct parport \*port，signed long timeout）

等待并行端口上的事件

**参数**

struct parport \*port

要等待的端口

signed long timeout

等待的时间（以jiffies为单位）

此函数最多等待timeout个jiffies，以便在并行端口上发生中断。如果端口超时设置为零，则立即返回。

如果在超时期满之前发生中断，此函数立即返回零。如果超时，则返回1。小于零的错误代码表示错误（很可能是未处理的信号），并且调用代码应尽快完成其正在执行的操作。

## int parport\_wait\_peripheral（struct parport \*port，unsigned char mask，unsigned char result）

等待35ms以使状态线发生变化

**参数**

struct parport \*port

要监视的端口

unsigned char mask

要监视的状态线

unsigned char result

所选状态行的期望值

**说明**

此函数等待，直到掩码状态线具有期望的值，或者经过35ms（请参见IEEE 1284-1994第24至25页了解为什么硬编码了此特定值）。掩码和结果参数是位掩码，其位由parport.h中的常量PARPORT\_STATUS\_BUSY等定义。

端口将快速轮询以开始，以期望外设的快速响应。此快速轮询时间是可配置的（使用/proc），默认为500微秒。如果此端口的超时时间（参见parport\_set\_timeout（））为零，则快速轮询时间为35ms，并且此函数不调用schedule（）。

如果此端口的超时时间非零，则在快速轮询失败后使用parport\_wait\_event（）等待最多10ms，如果发生中断，则唤醒。

## int parport\_negotiate（struct parport \*port，int mode）

协商IEEE 1284模式

**参数**

struct parport \*port

要使用的端口

int mode

要协商的模式

**说明**

使用此功能以协商特定的IEEE 1284传输模式。模式参数应是以IEEE1284\_MODE\_xxx开头的parport.h中的常量之一。

如果外设接受到了指定模式的协商，则返回为0，如果外设不符合IEEE 1284标准（或不存在），则为-1，如果外设拒绝了协商，则为1。

## ssize\_t parport\_write（struct parport \*port，const void \*buffer，size\_t len）

向并行端口写入数据块

**参数**

struct parport \*port

要写入的端口

const void \*buffer

数据缓冲区（在内核空间）

size\_t len

要传输的数据字节数

**说明**

这将使用最近协商的IEEE 1284传输模式（使用parport\_negotiate（）），将最多len个字节的缓冲器写入指定的端口，只要该模式支持向前传输（从主机到外设）。

调用者有责任确保缓冲区的前len个字节有效。

此函数返回传输的字节数（如果为零或正数），否则为错误代码。

## ssize\_t parport\_read（struct parport \*port，void \*buffer，size\_t len）

从并行端口读取数据块

**参数**

struct parport \*port

要读取的端口

void \*buffer

数据缓冲区（在内核空间）

size\_t len

要传输的数据字节

**说明**

这将使用最近协商的IEEE 1284传输模式（使用parport\_negotiate（）），将端口指定的最多len个字节的缓冲区读入缓冲区，只要该模式支持反向传输（从外设到主机））。

调用者有责任确保缓冲区的前len个字节可以写入。

此函数返回传输的字节数（如果为零或正数），否则为错误代码。

## long parport\_set\_timeout（struct pardevice \*dev，long inactivity）

设置设备的不活动超时时间

**参数**

struct pardevice \*dev

端口上的设备

long inactivity

不活动超时（以jiffies为单位）

**说明**

这会为端口上的特定设备设置不活动超时时间。这会影响parport\_wait\_peripheral（）等函数。特殊值0表示在处理此设备时不要调用schedule（）。

返回是先前的不活动超时。

对此设备进行任何parport\_wait\_event（）的调用程序都会被唤醒。

## int \_\_parport\_register\_driver（struct parport\_driver \*drv，struct module \*owner，const char \*mod\_name）

注册并行端口设备驱动程序

**参数**

struct parport\_driver \*drv

说明驱动程序的结构

struct module \*owner

drv的所有者模块

const char \*mod\_name

模块名称字符串

**说明**

并行端口设备驱动程序可以调用此函数以便接收有关在系统中发现的端口以及不再可用的端口的通知。

如果devmodel为true，则使用新设备模型进行注册。

drv结构是由调用者分配的，并且在调用parport\_unregister\_driver（）之后不得解除分配。

如果使用非设备模型：驱动程序的attach（）函数可能会阻塞。attach（）给定的端口在回调的持续时间内将有效，但如果驱动程序想要复制指针，则必须调用parport\_get\_port（）来执行此操作。在该端口上调用parport\_register\_device（）将为您执行此操作。

驱动程序的detach（）函数可能会阻塞。detach（）给定的端口在回调的持续时间内将有效，但如果驱动程序想要复制指针，则必须调用parport\_get\_port（）来执行此操作。

成功返回0。非设备模型将始终成功。但是，新设备模型可能会失败并返回错误代码。

## void parport\_unregister\_driver（struct parport\_driver \* drv）

注销并口设备驱动程序

**参数**

struct parport\_driver \* drv

说明驱动程序的结构是由parport\_register\_driver（）给出的。

**说明**

当要卸载使用parport\_register\_driver（）注册的并口设备驱动程序时，应调用此函数。

当它返回时，将不再调用驱动程序的attach（）例程，并调用了each端口的attach（），并调用了detach（）例程。

当此函数返回时，保证所有驱动程序的attach（）和detach（）调用已完成。

## struct parport \* parport\_get\_port（struct parport \* port）

增加端口的引用计数

**参数**

struct parport \* port

端口

**说明**

这将确保struct parport指针有效，直到匹配的parport\_put\_port（）调用。

## void parport\_put\_port（struct parport \* port）

减少端口的引用计数

**参数**

struct parport \* port

端口

**说明**

一旦不再需要端口，就应对每个对parport\_get\_port（）的调用调用一次此调用。当引用计数达到零（端口不再使用）时，将调用free\_port。

## struct parport \* parport\_register\_port（unsigned long base，int irq，int dma，struct parport\_operations \* ops）

注册并行端口

**参数**

unsigned long base

基本I / O地址

int irq

IRQ线

int dma

DMA信道

struct parport\_operations \* ops

指向端口驱动程序的端口操作结构的指针

**说明**

当并口（低级）驱动程序发现应向并行端口设备驱动程序提供的端口时，应调用parport\_register\_port（）。对于没有意义的端口，基础，irq和dma参数供端口驱动程序方便使用，不需要设置任何特殊值。可以通过调整表示端口的返回的parport结构的相关成员来修改它们。但是，在调用parport\_announce\_port之后不应该对它们进行篡改。

如果系统中有使用parport\_register\_driver（）注册自己的并行端口设备驱动程序，则此时不会告知端口; parport\_announce\_port（）完成此操作。

ops结构由调用者分配，并且在调用parport\_remove\_port（）之前不能被释放。

如果没有内存来分配新的parport结构，则此函数将返回NULL。

## void parport\_announce\_port（struct parport \* port）

告诉设备驱动程序有关并行端口

**参数**

struct parport \* port

并行端口通告

**说明**

在端口驱动程序使用parport\_register\_port注册并执行任何必要的初始化或调整之后，应调用parport\_announce\_port（）以通知所有调用parport\_register\_driver（）的设备驱动程序。它们的attach（）函数将被调用，以port作为参数。

## void parport\_remove\_port（struct parport \* port）

注销并行端口

**参数**

struct parport \* port

要注销的并行端口

**说明**

当并行端口驱动程序被强行卸载或并行端口变为不可访问时，端口驱动程序必须调用此函数以处理仍然想要使用它的设备驱动程序。

与该端口相关联的parport结构具有其操作结构替换为包含**返回**错误的“ null”操作的结构，或者仅不执行任何操作。

使用parport\_register\_driver（）注册自己的任何驱动程序都将通过将其detach（）例程调用为参数来通知不再可以访问端口。

## struct pardevice \* parport\_register\_dev\_model（struct parport \* port，const char \* name，const struct pardev\_cb \* par\_dev\_cb，int id）

在并行端口上注册设备

**参数**

struct parport \* port

连接设备的端口

const char \* name

用于引用设备的名称

const struct pardev\_cb \* par\_dev\_cb

包含回调的结构

int id

要赋予设备的设备号

**说明**

由并行端口设备驱动程序调用此函数，表示连接了设备到端口，并告诉系统所有它需要知道的信息。

struct pardev\_cb包含指向回调函数的指针。抢占回调函数，preempt，当此设备驱动程序声称已访问端口但其他设备驱动程序想要使用它时，会被调用。它被赋予为其参数的私有，并且如果它愿意让系统代表其向另一个驱动程序释放端口，则应返回零。如果它想保持控制端口，则应返回非零，并且不会采取任何行动。在其抢占回调拒绝抢占尝试后，驱动程序在尽早机会试图释放端口是良好的礼仪。请注意，如果抢占回调允许进行抢占，就无需释放端口;它是自动完成的。此函数可能不会阻塞，因为它可能会从中断上下文调用。如果设备驱动程序不支持抢占，则可以将preempt设置为NULL。

唤醒（“kick”）回调函数wakeup在并口可用于独占访问时调用；也就是说，当从唤醒回调函数内部调用parport\_claim()时，可以保证获得成功。如果驱动程序想要声明端口，则应该这样做；否则，它不需要采取任何行动。此函数可能不会阻塞，因为它可能会从中断上下文中调用。如果设备驱动程序不想以这种方式被明确邀请声明端口，则wakeup可以为空。

中断处理程序irq\_func在并口接收到中断时调用。请注意，如果设备驱动程序想要使用中断，应该使用parport\_enable\_irq()，并且还可以检查表示端口的parportstruct 中的irq成员。

并口（低级）驱动程序是第一个调用request\_irq()并且中断处理程序首先被调用的驱动程序。该处理程序会执行必要的操作来确认中断（对于类似PC的端口，没有什么特殊的要做）。然后，它会通知IEEE 1284代码有关中断的信息，这可能涉及根据当前IEEE 1284阶段对IEEE 1284事件做出反应。之后，它会调用irq\_func。不用说，irq\_func将从中断上下文中调用，并且可能不会阻塞。

PARPORT\_DEV\_EXCL标志用于防止共享端口，因此只有在无法共享端口且会导致不正确行为时才应该使用它。请谨慎使用！通常，标志将为零。

此函数返回一个表示端口上设备的struct 指针，如果没有足够的内存来分配该struct 的空间，则返回NULL。

## void parport\_unregister\_device（struct pardevice \* dev）

撤销并口注册设备

**参数**：

struct pardevice \* dev

表示设备的struct 指针

这撤销了parport\_register\_device()的效果。

## struct parport \* parport\_find\_number（int number）

按编号查找并口

**参数**：

int number

并口编号

这会返回指定编号的并口，如果没有则返回NULL。

已经有一个隐式的parport\_get\_port()；要丢弃parport\_find\_number()给出的端口引用，请使用parport\_put\_port()。

## struct parport \* parport\_find\_base（unsigned long base）

按基地址查找并口

**参数**：

unsigned long base

基本I/O地址

这会返回具有指定基地址的并口，如果没有则返回NULL。

已经有一个隐式的parport\_get\_port()；要丢弃parport\_find\_base()给出的端口引用，请使用parport\_put\_port()。

## int parport\_claim（struct pardevice \* dev）

声明并口设备访问

**参数**：

struct pardevice \* dev

表示端口上设备的struct 指针

此函数不会阻塞，因此可以从中断上下文中使用。如果parport\_claim()成功声明对端口的访问权限，则返回零，并且该端口可用于使用。如果端口正在由另一个驱动程序使用并且该驱动程序不愿放弃对端口的控制，则可能会失败（返回非零值）。

## int parport\_claim\_or\_block（struct pardevice \* dev）

声明并口设备访问

**参数**：

struct pardevice \* dev

表示端口上设备的struct 指针

这与parport\_claim()行为相同，但如果必要等待端口空闲，它将会阻塞。返回为1表示它睡眠；0表示它成功而无需睡眠。负错误代码表示失败。

## void parport\_release（struct pardevice \* dev）

放弃对并口设备的访问

**参数**：

struct pardevice \* dev

表示并口设备的struct 指针

此函数不能失败，但是不应该在未声明端口的情况下进行调用。同样，如果端口已被声明，则不应再次尝试声明它。

## struct pardevice \* parport\_open（int devnum，const char \* name）

按规范设备编号查找设备

**参数**：

int devnum

规范设备编号

const char \* name

与设备相关联的名称

此函数类似于parport\_register\_device()，但它通过其编号而不是其附加到的端口来定位设备。

除devnum之外，所有参数都与parport\_register\_device()相同。返回与parport\_register\_device()相同。

## void parport\_close（struct pardevice \* dev）

关闭通过parport\_open()打开的设备

**参数**：

struct pardevice \* dev

要关闭的设备

这对应于parport\_open()，就像parport\_unregister\_device()对应于parport\_register\_device()。

**说明**

设置端口FIFO超时值。cflag值应反映实际的硬件设置。

# 16x50 串口驱动

## void uart\_update\_timeout(struct uart\_port \* port, unsigned int cflag, unsigned int baud)

更新每端口FIFO超时.

**参数**

struct uart\_port \* port

描述端口的 uart\_port 结构

unsigned int cflag

termios cflag 值

unsigned int baud

端口速度

**说明**

设置端口 FIFO 超时值。cflag值应该反映实际的硬件设置

## unsigned int uart\_get\_baud\_rate(struct uart\_port \* port, struct ktermios \* termios, struct ktermios \* old, unsigned int min, unsigned int max)

返回特定端口的波特率

**参数**

struct uart\_port \* port

说明所讨论端口的uart\_port结构。

struct ktermios \* termios

所需的termios设置。

struct ktermios \* old

旧的termios（或为NULL）

unsigned int min

最小接受的波特率

unsigned int max

最大接受的波特率

**说明**

将termios结构解码为数字波特率，考虑到魔法的38400波特率（具有spd\_\*标志），并将B0比特率映射到9600波特率。

如果新的波特率无效，请尝试旧的termios设置。如果仍然无效，我们尝试9600波特率。

更新termios结构以反映实际使用的波特率。不要为请求B0（“挂起”）的情况执行此操作。

## unsigned int uart\_get\_divisor(struct uart\_port \* port, unsigned int baud)

返回串行端口时钟除数

**参数**

struct uart\_port \* port

说明端口的uart\_port结构。

unsigned int baud

所需的波特率

**说明**

计算端口的串行端口时钟除数。

## void uart\_console\_write（struct uart\_port \* port，const char \* s，unsigned int count，void（\* putchar）（struct uart\_port \*，int）

向串行端口写入控制台消息

**参数**

struct uart\_port \* port

要写入消息的端口

const char \* s

字符数组

unsigned int count

要写入的字符串中的字符数

void（\*）（struct uart\_port \*，int）putchar

写入字符到端口的函数

## int uart\_parse\_earlycon（char \* p，unsigned char \* iotype，resource\_size\_t \* addr，char \*\* options）

解析早期连接选项

**参数**

char \* p

指向第二个字段的指针（即，仅超出'<name>，'）

unsigned char \* iotype

解码的iotype的指针（输出）

resource\_size\_t \* addr

解码的mapbase/ iobase的指针（输出）

char \*\* options

<选项>字段的指针；如果不存在，则为NULL（输出）

**说明**

解码形式为

earlycon = <name>，io | mmio | mmio16 | mmio32 | mmio32be | mmio32native，<addr>，<options> console = <name>，io | mmio | mmio16 | mmio32 | mmio32be | mmio32native，<addr>，<options>的早期连接内核命令行参数。

还接受<name>，0x<addr>，<options>的可选形式；返回的iotype将为UPIO\_MEM。

成功返回0，失败返回-EINVAL

## void uart\_parse\_options（const char \* options，int \* baud，int \* parity，int \* bits，int \* flow）

解析串行端口波特率/奇偶校验位/数据位/流控制。

**参数**

const char \* options

指向选项字符串的指针

int \* baud

指向波特率的“int”变量的指针。

int \* parity

指向奇偶校验位的“int”变量的指针。

int \* bits

指向数据位的“int”变量的指针。

int \* flow

指向流控制字符的“int”变量的指针。

**说明**

uart\_parse\_options解码包含串行控制台选项的字符串。字符串的格式为<baud><parity><bits><flow>，例如：115200n8r

## int uart\_set\_options（struct uart\_port \* port，struct console \* co，int baud，int parity，int bits，int flow）

设置串行控制台参数

**参数**

struct uart\_port \* port

指向串行端口uart\_port结构的指针

struct console \* co

控制台指针

int baud

波特率

int parity

奇偶校验字符 - 'n'（无），'o'（奇数），'e'（偶数）

int bits

数据位数

int flow

流控制字符 - 'r'（rts）

## int uart\_register\_driver（struct uart\_driver \* drv）

使用串行控制器内核层注册驱动程序

**参数**

struct uart\_driver \* drv

低级驱动程序结构

**说明**

将串行控制器驱动程序注册到核心驱动程序。我们反过来向tty层注册，并初始化每个端口状态的核心驱动程序。

我们在/proc/tty/driver中有一个名为正常驱动程序的proc文件。

drv->port应为NULL，并且在此调用成功后，应使用uart\_add\_one\_port注册每个端口结构。

## void uart\_unregister\_driver（struct uart\_driver \* drv）

从串行控制器核心层中移除驱动程序

**参数**

struct uart\_driver \* drv

低级驱动程序结构

**说明**

从核心驱动程序中删除对驱动程序的所有引用。必须通过uart\_remove\_one\_port()从低级驱动程序中删除所有端口，如果使用uart\_add\_one\_port()注册了端口。 (即，drv->port == NULL)

## int uart\_add\_one\_port（struct uart\_driver \* drv，struct uart\_port \* uport）

附加驱动程序**定义**的端口结构

**参数**

struct uart\_driver \* drv

用于此端口的串行低级驱动程序结构的指针

struct uart\_port \* uport

用于此端口的串行端口结构。

**说明**

这允许驱动程序使用核心驱动程序注册自己的uart\_port结构。主要目的是允许低级串口驱动程序扩展uart\_port，而不是有更多的结构层次。

## int uart\_remove\_one\_port（struct uart\_driver \* drv，struct uart\_port \* uport）

分离驱动程序定义的端口结构

**参数**

struct uart\_driver \* drv

用于此端口的串行低级驱动程序结构的指针

struct uart\_port \* uport

该端口的uart端口struct

**说明**

此函数将指定的端口struct 从核心驱动程序中解绑（并挂断）。将不再对此端口调用低级别代码。

## void uart\_handle\_dcd\_change（struct uart\_port \* uport，unsigned int status）

处理承载检测状态变化

**参数**

struct uart\_port \* uport

打开端口的uart\_port结构

unsigned int status

新的承载检测状态，非零则为活动状态

**说明**

调用者必须持有uport->lock

## void uart\_handle\_cts\_change（struct uart\_port \* uport，unsigned int status）

处理清除发送状态变化

**参数**

struct uart\_port \* uport

打开端口的uart\_port结构

unsigned int status

新的清除发送状态，非零则为活动状态

**说明**

调用者必须持有uport->lock

## void uart\_insert\_char（struct uart\_port \* port，unsigned int status，unsigned int overrun，unsigned int ch，unsigned int flag）

将字符推入uart层

**参数**

struct uart\_port \* port

相应端口

unsigned int status

串行端口RX缓冲区的状态（8250的LSR）

unsigned int overrun

status中溢出位的掩码

unsigned int ch

要推送的字符

unsigned int flag

字符的标志（见TTY\_NORMAL和friends）

**说明**

用户负责在完成插入后调用tty\_flip\_buffer\_push。

## void uart\_get\_rs485\_mode（struct device \* dev，struct serial\_rs485 \* rs485conf）

检索给定串口的rs485属性

**参数**

struct device \* dev

uart设备

struct serial\_rs485 \* rs485conf

输出参数

**说明**

此函数实现了Documentat中说明的设备树绑定/serial/rs485.txt。

## struct uart\_8250\_port \* serial8250\_get\_port（int line）

检索struct uart\_8250\_port

**参数**

int line

串行线路编号

**说明**

此函数检索特定线路的struct uart\_8250\_port。此struct 必须不用于执行除其他方式不可访问的8250或串行核心操作以外的任何操作。它的唯一目的是使struct可访问运行时-pm回调以进行上下文挂起/恢复。这里所做的锁定假设是没有的，因为如果端口上执行任何操作，就不应调用运行时-pm挂起/恢复回调。

## void serial8250\_suspend\_port（int line）

挂起一个串行端口

**参数**

int line

串行线路编号

**说明**

挂起一个串行端口。

## void serial8250\_resume\_port（int line）

恢复一个串行端口

**参数**

int line

串行线路编号

**说明**

恢复一个串行端口。

## int serial8250\_register\_8250\_port（struct uart\_8250\_port \* up）

注册一个串口

**参数**

struct uart\_8250\_port \* up

串口模板

**说明**

为指定的串行端口配置串行端口。如果端口存在并正在使用，则会挂断并注销它。

然后会探测该端口，如果需要，IRQ会自动检测。如果失败，会返回一个错误。

成功后，端口准备好使用，将返回线路编号。

## void serial8250\_unregister\_port（int line）

在运行时删除一个16x50串行端口

**参数**

int line

串行线路编号

**说明**

删除一个串行端口。这可能不会从中断上下文中调用。我们将该端口交回我们的控制。

# 脉冲宽度调制 (PWM)

脉宽调制是一种主要用于控制供电到电气设备的调制技术。

PWM框架为PWM信号的提供者和消费者提供了一个抽象。提供一个或多个PWM信号的控制器被注册为struct pwm\_chip。提供者应将此结构嵌入驱动程序特定的结构中。该结构包含说明特定芯片的字段。

芯片公开了一个或多个PWM信号源，每个信号源都以struct pwm\_device的形式公开。可以对PWM设备执行操作，以控制信号的周期、占空比、极性和活动状态。

请注意，PWM设备是独占资源：它们始终只能由一个消费者同时使用。

## enum pwm\_polarity

PWM信号的极性

**常量**

PWM\_POLARITY\_NORMAL

高电平信号持续时间为占空比，其余脉冲周期为低电平信号

PWM\_POLARITY\_INVERSED

低电平信号持续时间为占空比，其余脉冲周期为高电平信号

## struct pwm\_args

与电路板相关的PWM参数

**定义**

struct pwm\_args {

u64 period;

enum pwm\_polarity polarity;

};

**成员**

period

参考周期

polarity

参考极性

**说明**

此结构说明了附加到PWM设备的与电路板相关的参数。这些参数通常从PWM查找表或设备树中检索。

请勿将此与PWM状态混淆：PWM参数表示用户希望在此PWM设备上使用的初始配置，而不是当前的PWM硬件状态。

## struct pwm\_device

PWM通道对象

**定义**

struct pwm\_device {

const char \*label;

unsigned long flags;

unsigned int hwpwm;

unsigned int pwm;

struct pwm\_chip \*chip;

void \*chip\_data;

struct pwm\_args args;

struct pwm\_state state;

};

**成员**

label

PWM 设备的名称

flags

与 PWM 设备关联的标志

hwpwm

PWM设备的每芯片相对索引

pwm

PWM 设备的全局索引

chip

提供此PWM设备的PWM芯片

chip\_data

与 PWM 设备关联的芯片专用数据

args

脉宽调制参数

state

当前 PWM 通道状态

## void pwm\_get\_state(const struct [pwm\_device](https://www.kernel.org/doc/html/latest/driver-api/miscellaneous.html" \l "c.pwm_device" \o "pwm_device) \*pwm, struct pwm\_state \*state)

检索当前PWM状态

**参数**

const struct pwm\_device \*pwm

PWM设备

struct pwm\_state \*state

用当前PWM状态填充的状态

## void pwm\_init\_state(const struct pwm\_device \*pwm, struct pwm\_state \*state)

准备新状态，以便应用于pwm\_apply\_state()

**参数**

const struct pwm\_device \*pwm

PWM设备

struct pwm\_state \*state

用准备好的PWM状态填充的状态

**说明**

这个函数准备一个状态，稍后可以调整并应用到PWM设备上，调用pwm\_apply\_state()。这是一个方便的函数，它首先检索当前PWM状态，然后用pwm->args中定义的参考值替换周期和极性字段。一旦函数返回，您可以根据需要调整->enabled和->duty\_cycle字段，然后调用pwm\_apply\_state()。

pwm\_apply\_state()->duty\_cycle最初设置为零，避免当前->duty\_cycle值超过pwm\_args->period值的情况，否则如果用户在调整->duty\_cycle之前调用pwm\_apply\_state()，会触发一个错误。

## unsigned int pwm\_get\_relative\_duty\_cycle(const struct pwm\_state \*state, unsigned int scale)

获取相对占空比值

**参数**

const struct pwm\_state \*state

要从中提取占空比的PWM状态

unsigned int scale

相对占空比的目标比例尺

**说明**

这个函数将存储在状态中的绝对占空比（以纳秒表示）转换为相对于周期的值。

例如，如果您想以百分比表示占空比，请调用：

pwm\_get\_state(pwm, state); duty = pwm\_get\_relative\_duty\_cycle(state, 100);

## int pwm\_set\_relative\_duty\_cycle(struct pwm\_state \*state, unsigned int duty\_cycle, unsigned int scale)

设置相对占空比值

**参数**

struct pwm\_state \*state

要填充的PWM状态

unsigned int duty\_cycle

相对占空比值

unsigned int scale

占空比表达的比例尺

**说明**

这个函数将相对值转换为绝对占空比（以纳秒为单位），并将结果放在state->duty\_cycle中。

例如，如果您想配置50%的占空比，请调用：

pwm\_init\_state(pwm, state); pwm\_set\_relative\_duty\_cycle(state, 50, 100); pwm\_apply\_state(pwm, state);

如果duty\_cycle和/或scale不一致（scale == 0或duty\_cycle> scale），则此函数返回-EINVAL。

## struct pwm\_ops

**定义**

struct pwm\_ops {

int (\*request)(struct pwm\_chip \*chip, struct pwm\_device \*pwm);

void (\*free)(struct pwm\_chip \*chip, struct pwm\_device \*pwm);

int (\*config)(struct pwm\_chip \*chip, struct pwm\_device \*pwm, int duty\_ns, int period\_ns);

int (\*set\_polarity)(struct pwm\_chip \*chip, struct pwm\_device \*pwm, enum pwm\_polarity polarity);

int (\*capture)(struct pwm\_chip \*chip, struct pwm\_device \*pwm, struct pwm\_capture \*result, unsigned long timeout);

int (\*enable)(struct pwm\_chip \*chip, struct pwm\_device \*pwm);

void (\*disable)(struct pwm\_chip \*chip, struct pwm\_device \*pwm);

int (\*apply)(struct pwm\_chip \*chip, struct pwm\_device \*pwm, struct pwm\_state \*state);

void (\*get\_state)(struct pwm\_chip \*chip, struct pwm\_device \*pwm, struct pwm\_state \*state);

#ifdef CONFIG\_DEBUG\_FS;

void (\*dbg\_show)(struct pwm\_chip \*chip, struct seq\_file \*s);

#endif;

struct module \*owner;

};

**成员**

request

用于请求 PWM 的可选挂钩

free

用于释放 PWM 的可选挂钩

config

为此 PWM 配置占空比和周期长度

set\_polarity

配置此 PWM 的极性

capture

捕获并报告 PWM 信号

enable

启用 PWM 输出切换

disable

禁用 PWM 输出切换

apply

自动应用新的 PWM 配置。state 参数应根据实际硬件配置进行调整（如果近似于周期或 duty\_cycle 值，state 应反映出来）

get\_state

获取当前 PWM 状态。当 PWM 芯片注册时，每个 PWM 设备只调用一次此函数。

dbg\_show

在 debugfs 中显示内容的可选例程

owner

有助于防止移除输出有源 PWM 的模块

## struct pwm\_chip

抽象一个PWM控制器

**定义**

struct pwm\_chip {

struct device \*dev;

struct list\_head list;

const struct pwm\_ops \*ops;

int base;

unsigned int npwm;

struct pwm\_device \*pwms;

struct pwm\_device \* (\*of\_xlate)(struct pwm\_chip \*pc, const struct of\_phandle\_args \*args);

unsigned int of\_pwm\_n\_cells;

};

**成员**

dev

提供 PWM 的设备

list

列出供内部使用的节点

ops

此 PWM 控制器的回调

base

本芯片控制的第一个PWM数

npwm

该芯片控制的PWM数量

pwms

由框架分配的 PWM 设备数组

of\_xlate

在给定设备树 PWM 说明符的情况下请求 PWM 设备

of\_pwm\_n\_cells

设备树 PWM 说明符中预期的单元数

## struct pwm\_capture

PWM捕获数据

**定义**

struct pwm\_capture {

unsigned int period;

unsigned int duty\_cycle;

};

**成员**

period

PWM 信号的周期（以纳秒为单位）

duty\_cycle

PWM 信号的占空比（以纳秒为单位）

## int pwm\_config( struct [pwm\_device](https://www.kernel.org/doc/html/v4.19/driver-api/miscellaneous.html" \l "c.pwm_device" \o "脉宽调制设备) \*  pwm , int  duty\_ns , int  period\_ns )

更改 PWM 设备配置

**参数**

struct pwm\_device \* pwm

PWM设备

int duty\_ns

“开启”时间（以纳秒为单位）

int period\_ns

一个周期的持续时间（以纳秒为单位）

**返回**

成功时为 0，失败时为负错误代码。

## int pwm\_set\_polarity( struct [pwm\_device](https://www.kernel.org/doc/html/v4.19/driver-api/miscellaneous.html" \l "c.pwm_device" \o "脉宽调制设备) \*  pwm , enum [pwm\_polarity](https://www.kernel.org/doc/html/v4.19/driver-api/miscellaneous.html" \l "c.pwm_polarity" \o "脉宽调制极性) 极性)

配置 PWM 信号的极性

**参数**

struct pwm\_device \* pwm

PWM设备

enum pwm\_polarity polarity

PWM 信号的新极性

**说明**

注意，当PWM设备启用时无法配置极性。

**返回**

成功返回0，失败返回负数错误代码。

## int pwm\_enable(struct pwm\_device \* pwm)

启动PWM输出切换

**参数**

struct pwm\_device \* pwm

PWM设备

**返回**

成功返回0，失败返回负数错误代码。

## void pwm\_disable(struct pwm\_device \* pwm)

停止PWM输出切换

**参数**

struct pwm\_device \* pwm

PWM设备

## int pwm\_set\_chip\_data(struct pwm\_device \* pwm，void \* data)

设置PWM的私有芯片数据

**参数**

struct pwm\_device \* pwm

PWM设备

void \* data

指向芯片特定数据的指针

**返回**

成功返回0，失败返回负数错误代码。

## void \* pwm\_get\_chip\_data(struct pwm\_device \* pwm)

获取PWM的私有芯片数据

**参数**

struct pwm\_device \* pwm

PWM设备

返回

返回PWM设备芯片专用数据的指针。

## int pwmchip\_add\_with\_polarity(struct pwm\_chip \* chip，enum pwm\_polarity polarity)

注册新的PWM芯片

**参数**

struct pwm\_chip \* chip

要添加的PWM芯片

枚举pwm\_polarity polarity

PWM频道的初始极性

**说明**

注册新的PWM芯片。如果chip->base < 0，则使用动态分配的基本值。所有通道的初始极性由极性参数指定。

**返回**

成功返回0，失败返回负数错误代码。

## int pwmchip\_add(struct pwm\_chip \* chip)

注册新的PWM芯片

**参数**

struct pwm\_chip \* chip

要添加的PWM芯片

**说明**

注册新的PWM芯片。

**返回**

成功返回0，失败返回负数错误代码。

## int pwmchip\_remove(struct pwm\_chip \* chip)

删除PWM芯片

**参数**

struct pwm\_chip \* chip

要删除的PWM芯片

**说明**

删除PWM芯片。如果PWM芯片提供的PWM设备仍在请求，则此函数可能会返回忙。

**返回**

成功返回0，失败返回负数错误代码。

## struct pwm\_device \* pwm\_request(int pwm，const char \* label)

请求PWM设备

**参数**

int pwm

全局PWM设备索引

const char \* label

PWM设备标签

**说明**

此函数已弃用，请改用pwm\_get()。

**返回**

指向PWM设备的指针或ERR\_PTR（）编码的错误代码（如果失败）。

## struct pwm\_device \* pwm\_request\_from\_chip(struct pwm\_chip \* chip，unsigned int index，const char \* label)

相对于PWM芯片请求PWM设备

**参数**

struct pwm\_chip \* chip

PWM芯片

unsigned int index

要请求的PWM的每芯片索引

const char \* label

此PWM的文字说明字符串

**返回**

返回给定PWM芯片给定索引处的PWM设备的指针。如果索引不适用于指定的PWM芯片或无法请求PWM设备，则返回负错误代码。

## void pwm\_free(struct pwm\_device \* pwm)

释放PWM设备

**参数**

struct pwm\_device \* pwm

PWM设备

**说明**

此函数已弃用，请改用pwm\_put()。

## int pwm\_apply\_state(struct pwm\_device \* pwm，const struct pwm\_state \* state)

原子地将新状态应用于PWM设备

**参数**

struct pwm\_device \* pwm

PWM设备

const struct pwm\_state \* state

要应用的新状态

## int pwm\_capture(struct pwm\_device \* pwm，struct pwm\_capture \* result，unsigned long timeout)

捕获并报告PWM信号

**参数**

struct pwm\_device \* pwm

PWM设备

struct pwm\_capture \* result

填充捕获结果的结构

unsigned long timeout

等待的时间（以毫秒为单位），在放弃捕获之前

**返回**

成功返回0，失败返回负数错误代码。

## int pwm\_adjust\_config(struct pwm\_device \* pwm)

将当前PWM配置调整为PWM参数

**参数**

struct pwm\_device \* pwm

PWM设备

**说明**

此函数将PWM配置调整为由DT或PWM查找表提供的PWM参数。这对于将引导加载程序配置适应Linux配置特别有用。

## struct pwm\_device \* of\_pwm\_get（struct device\_node \* np，const char \* con\_id）

通过PWM框架请求PWM

**参数**

struct device\_node \* np

从中获取PWM的设备节点

const char \* con\_id

消费者名称

**说明**

返回从设备树节点的“pwms”属性中指定的phandle和索引解析的PWM设备，或失败的负错误代码。从设备树解析的值存储在返回的PWM设备对象中。

如果con\_id为NULL，则请求列在“pwms”属性中的第一个PWM设备。否则，“pwm-names”属性用于反向查找PWM索引。这也意味着“pwm-names”属性对于通过con\_id参数查找PWM设备的设备变得强制。

**返回**

请求的PWM设备指针或在失败时编码错误代码的ERR\_PTR()。

## struct pwm\_device \*pwm\_get(struct device \*dev, const char \*con\_id)

查找并请求PWM设备

**参数**

struct device \*dev

PWM消费者设备

const char \*con\_id

消费者名称

**说明**

首先使用DT进行查找。如果设备没有从设备树实例化，则通过板设置代码提供的表查找PWM芯片和相对索引（请参见pwm\_add\_table()）。

一旦找到PWM芯片，指定的PWM设备将被请求并准备好使用。

**返回**

请求的PWM设备指针或在失败时编码错误代码的ERR\_PTR()。

## void pwm\_put(struct pwm\_device \*pwm)

释放PWM设备

**参数**

struct pwm\_device \*pwm

PWM设备

## struct pwm\_device \*devm\_pwm\_get(struct device \*dev, const char \*con\_id)

资源管理的pwm\_get()

**参数**

struct device \*dev

PWM消费者设备

const char \*con\_id

消费者名称

**说明**

此函数类似于pwm\_get()，但获取的PWM设备将在驱动程序分离时自动释放。

**返回**

请求的PWM设备指针或在失败时编码错误代码的ERR\_PTR()。

## struct pwm\_device \*devm\_of\_pwm\_get(struct device \*dev, struct device\_node \*np, const char \*con\_id)

资源管理的of\_pwm\_get()

**参数**

struct device \*dev

PWM消费者设备

struct device\_node \*np

从中获取PWM的设备节点

const char \*con\_id

消费者名称

**说明**

此函数类似于of\_pwm\_get()，但获取的PWM设备将在驱动程序分离时自动释放。

**返回**

请求的PWM设备指针或在失败时编码错误代码的ERR\_PTR()。

## void devm\_pwm\_put(struct device \*dev, struct pwm\_device \*pwm)

资源管理的pwm\_put()

**参数**

struct device \*dev

PWM消费者设备

struct pwm\_device \*pwm

PWM设备

**说明**

释放以前使用devm\_pwm\_get()分配的PWM。通常不需要调用此函数，因为devm分配的资源会在驱动程序分离时自动释放。

# W1: Dallas的 1-wire总线

## 内核内部的W1接口

### include/linux/w1.h

W1内核接口函数

#### struct w1\_reg\_num

断开的从属设备ID

**定义**

struct w1\_reg\_num {

#if defined(\_\_LITTLE\_ENDIAN\_BITFIELD);

\_\_u64 family:8,id:48, crc:8;

#elif defined(\_\_BIG\_ENDIAN\_BITFIELD);

\_\_u64 crc:8,id:48, family:8;

#else;

#error "Please fix <asm/byteorder.h>";

#endif;

};

**成员**

family

标识设备类型

ID

与家族一起是唯一的设备ID

crc

其他字节的校验和

crc

其他字节的校验和

id

与家族一起是唯一的设备ID

family

标识设备类型

#### struct w1\_slave

在总线上持有单个从设备

**定义**

struct w1\_slave {

struct module \*owner;

unsigned char name[W1\_MAXNAMELEN];

struct list\_head w1\_slave\_entry;

struct w1\_reg\_num reg\_num;

atomic\_t refcnt;

int ttl;

unsigned long flags;

struct w1\_master \*master;

struct w1\_family \*family;

void \*family\_data;

struct device dev;

struct device \*hwmon;

};

**成员**

owner

指向单线“wire”内核模块。

name

设备 ID 是 ascii。

w1\_slave\_entry

链表的数据

reg\_num

二进制的slave id

refcnt

引用计数，0时删除

ttl

每次搜索递减此奴隶未找到，在 0 deatch

flags

W1\_SLAVE\_ACTIVE W1\_SLAVE\_DETACH 的位标志

master

这个奴隶所在的总线

family

设备系列类型模块

family\_data

家庭模块使用的指针

dev

内核设备标识符

hwmon

指向 hwmon 设备的指针

#### struct w1\_bus\_master

总线主机上可用的操作

**定义**

struct w1\_bus\_master {

void \*data;

u8 (\*read\_bit)(void \*);

void (\*write\_bit)(void \*, u8);

u8 (\*touch\_bit)(void \*, u8);

u8 (\*read\_byte)(void \*);

void (\*write\_byte)(void \*, u8);

u8 (\*read\_block)(void \*, u8 \*, int);

void (\*write\_block)(void \*, const u8 \*, int);

u8 (\*triplet)(void \*, u8);

u8 (\*reset\_bus)(void \*);

u8 (\*set\_pullup)(void \*, int);

void (\*search)(void \*, struct w1\_master \*, u8, w1\_slave\_found\_callback);

};

**成员**

data

下面所有函数的第一个参数

read\_bit

采样线路电平返回读取的电平（0 或 1）

write\_bit

设置线路电平

touch\_bit

真正支持 1-wire 协议的设备的最低级功能。touch\_bit(0) = write-0 cycle touch\_bit(1) = write-1 / read cycle 返回读取的位（0 或 1）

read\_byte

读取一个字节。与 8 个 touch\_bit(1) 调用相同。 返回读取的字节

write\_byte

写入一个字节。与 8 个 touch\_bit(x) 调用相同。

read\_block

同一系列read\_byte()调用 返回读取的字节数

write\_block

与一系列write\_byte()调用相同

triplet

结合两次读取和一次智能写入以进行 ROM 搜索 返回bit0=Id bit1=comp\_id bit2=dir\_taken

reset\_bus

long write-0 with a read for the presence pulse detection return -1=错误，0=设备存在，1=不存在设备

set\_pullup

发出指定持续时间的强上拉脉冲。 返回-1=错误，0=完成

search

非常好的硬件可以处理不同类型的 ROM 搜索 w1\_master\* 传递给从发现回调。u8 是 search\_type、W1\_SEARCH 或 W1\_ALARM\_SEARCH

**注**

read\_bit和write\_bit是非常低级的函数，只应该在不真正支持单线操作的硬件，如并行/串行端口中使用。只定义read\_bit和write\_bit或至少定义touch\_bit和reset\_bus.

#### enum w1\_master\_flags

位字段在w1\_master.flags中使用

**常量**

W1\_ABORT\_SEARCH

在关闭时提前中止搜索

W1\_WARN\_MAX\_COUNT

达到最大计数时限制警告

#### struct w1\_master

每个总线主设备一个

**定义**

struct w1\_master {

struct list\_head w1\_master\_entry;

struct module \*owner;

unsigned char name[W1\_MAXNAMELEN];

struct mutex list\_mutex;

struct list\_head slist;

struct list\_head async\_list;

int max\_slave\_count, slave\_count;

unsigned long attempts;

int slave\_ttl;

int initialized;

u32 id;

int search\_count;

u64 search\_id;

atomic\_t refcnt;

void \*priv;

int enable\_pullup;

int pullup\_duration;

long flags;

struct task\_struct \*thread;

struct mutex mutex;

struct mutex bus\_mutex;

struct device\_driver \*driver;

struct device dev;

struct w1\_bus\_master \*bus\_master;

u32 seq;

};

**成员**

w1\_master\_entry

主设备链接列表

owner

模块所有者

name

动态分配总线名称

list\_mutex

保护slist和async\_list

slist

从设备链表

async\_list

Netlink命令链表执行

max\_slave\_count

一次最多搜索的从设备数

slave\_count

已知从设备数

attempts

搜索次数

slave\_ttl

从设备超时前的搜索次数

initialized

防止init/removal比赛条件

id

w1总线编号

search\_count

自动搜索次数，-1为无限

search\_id

允许继续搜索

refcnt

引用计数

priv

私有数据存储

enable\_pullup

允许强拉电

pullup\_duration

下一次强拉电的时间

flags

w1\_master\_flags之一

thread

总线搜索和Netlink命令的线程

mutex

保护大部分的w1\_master

bus\_mutex

保护并发总线访问

driver

sysfs驱动程序

dev

sysfs设备

bus\_master

可用的IO操作

seq

用于Netlink广播的序列号

#### struct w1\_family\_ops

一种家族类型的操作

**定义**

struct w1\_family\_ops {

int (\*add\_slave)(struct w1\_slave \*sl);

void (\*remove\_slave)(struct w1\_slave \*sl);

const struct attribute\_group \*\*groups;

const struct hwmon\_chip\_info \*chip\_info;

};

**成员**

add\_slave

添加从设备

remove\_slave

删除从设备

groups

系统组

chip\_info

指向struct hwmon\_chip\_info的指针

#### struct w1\_family

参考计数家族结构

**定义**

struct w1\_family {

struct list\_head family\_entry;

u8 fid;

const struct w1\_family\_ops \*fops;

const struct of\_device\_id \*of\_match\_table;

atomic\_t refcnt;

};

**成员**

family\_entry

家族链接列表

fid

8位家族标识符

fops

此家族的操作

of\_match\_table

打开固件匹配表

refcnt

参考计数器

#### module\_w1\_family (\_\_w1\_family)

用于注册1-wire家族驱动程序的辅助宏

**参数**

\_\_w1\_family

w1\_family struct

**说明**

用于1-Wire家族，它们在模块初始化/ 退出中不进行任何特殊操作。 这样可以消除很多样板文件。 每个模块只能使用此宏一次，并调用它将替换module\_init（）和module\_exit（）

### drivers/w1/w1.c

W1核心功能。

#### void w1\_search(**struct** w1\_master \*dev, u8 search\_type, w1\_slave\_found\_callback cb)

执行ROM搜索并注册找到的任何设备。

**参数**

struct w1\_master \*dev

要搜索的主设备

u8 search\_type

W1\_SEARCH搜索所有设备，或W1\_ALARM\_SEARCH仅返回处于警报状态的设备

w1\_slave\_found\_callback cb

找到设备时要调用的函数

**说明**

1-wire搜索是一种简单的二叉树搜索。 对于地址的每个位，我们读取两个位并写入一个位。 写入的位将使所有不匹配该位的devies进入睡眠状态。 当两个读取不同时，方向选择是显而易见的。 当两个位都为0时，我们必须选择一条路径。 当我们可以扫描所有64位而不必选择路径时，我们完成了。

请参阅 www.maxim-ic.com上的“应用说明187 1-wire 搜索算法”

#### int w1\_process\_callbacks(struct [w1\_master](https://www.kernel.org/doc/html/latest/driver-api/w1.html" \l "c.w1_master" \o "w1_master) \*dev)

执行每个dev->async\_list回调条目

**参数**

struct w1\_master \*dev

w1\_master设备

**说明**

必须保持w1主列表\_type\_mutex。

**返回**

如果有要执行的命令，则为1，否则为0

### drivers/w1/w1\_family.c

允许注册设备族操作。

#### int w1\_register\_family(struct w1\_family \*newf)

注册设备族驱动程序

**参数**

struct w1\_family \*newf

要注册的家族

#### void w1\_unregister\_family(struct w1\_family \*fent)

注销设备族驱动程序

**参数**

struct w1\_family \*fent

要取消注册的家族

### drivers/w1/w1\_internal.h

主设备的W1内部初始化

#### struct w1\_async\_cmd

从w1\_process kthread执行回调

**定义**

struct w1\_async\_cmd {

struct list\_head async\_entry;

void (\*cb)(struct w1\_master \*dev, struct w1\_async\_cmd \*async\_cmd);

};

**成员**

async\_entry

链接条目

cb

回调函数，必须在返回之前列出并销毁此列表

**说明**

当插入到w1\_master async\_list中时，将执行w1\_process回调。 将其嵌入具有命令详细信息的结构中

### drivers/w1/w1\_int.c

主设备的W1内部初始化。

#### int w1\_add\_master\_device(struct w1\_bus\_master \*master)

注册新主设备

**参数**

struct w1\_bus\_master \*master

注册的主设备总线设备

#### void w1\_remove\_master\_device(struct w1\_bus\_master \*bm)

注销主设备

**参数**

struct w1\_bus\_master \*bm

要删除的主设备总线设备

### drivers/w1/w1\_netlink.h

W1外部netlink API结构和命令

#### enum w1\_cn\_msg\_flags

cn\_msg.flags结构体的位域标志

**常量**

W1\_CN\_BUNDLE

将请求捆绑到较少的消息中回复。准备在一个数据包中处理多个struct cn\_msg、struct\_w1\_netlink\_msg和struct\_w1\_netlink\_cmd

#### enum w1\_netlink\_message\_types

消息类型

**常量**

W1\_SLAVE\_ADD

通知添加了一个从设备

W1\_SLAVE\_REMOVE

通知删除了一个从设备

W1\_MASTER\_ADD

通知添加了一个新总线主机

W1\_MASTER\_REMOVE

通知删除了一个总线主机

W1\_MASTER\_CMD

在特定主机上启动操作

W1\_SLAVE\_CMD

发送复位、选择从站，然后进行读/写/触摸操作

W1\_LIST\_MASTERS

用于确定总线主机标识符

#### struct w1\_netlink\_msg

保存w1消息类型、id和结果

**定义**

struct w1\_netlink\_msg {

\_\_u8 type;

\_\_u8 status;

\_\_u16 len;

union {

\_\_u8 id[8];

struct w1\_mst {

\_\_u32 id;

\_\_u32 res;

} mst;

} id;

\_\_u8 data[];

};

**成员**

type

枚举 w1\_netlink\_message\_types 之一

status

内核反馈成功 0 或 errno 失败值

len

w1\_netlink\_msg 之后的数据长度

id

union 持有总线主机 ID (msg.id) 和从设备 ID (id[8])。

id.id

从站 ID（8 字节）

id.mst

总线主机识别

id.mst.id

总线主控 ID

id.mst.res

保留总线主控

data

任何后续数据的起始地址

**说明**

w1消息在netlink上的基本消息结构。netlink连接器数据序列为：structnlmsghdr、struct cn\_msg，然后是一个或多个struct\_w1\_netlink\_msg（每个带可选数据）

#### enum w1\_commands

可用于主机或从机操作的命令

**常量**

W1\_CMD\_READ

读取len字节

W1\_CMD\_WRITE

写入len字节

W1\_CMD\_SEARCH

启动标准搜索，仅返回在该搜索期间找到的从设备

W1\_CMD\_ALARM\_SEARCH

搜索当前正在警报的设备

W1\_CMD\_TOUCH

触摸一系列字节。

W1\_CMD\_RESET

在给定主机上发送总线复位

W1\_CMD\_SLAVE\_ADD

将从设备添加到给定的主机，8字节的从ID在data [0]处

W1\_CMD\_SLAVE\_REMOVE

从给定主机中删除从设备，在data [0]处的8字节从ID

W1\_CMD\_LIST\_SLAVES

列出在此主机上注册的从设备

W1\_CMD\_MAX

可用命令数量

#### struct w1\_netlink\_cmd

保存命令和数据

**定义**

struct w1\_netlink\_cmd {

\_\_u8 cmd;

\_\_u8 res;

\_\_u16 len;

\_\_u8 data[];

};

**成员**

cmd

其中之一enum\_w1\_commands

res

保留

len

跟随w1\_netlink\_cmd的数据长度

数据

后续任何数据的起始地址

**说明**

一个或多个struct\_w1\_netlink\_cmd放置在w1\_netlink\_msg.data开始处，每个可选数据。

### drivers/w1/w1\_io.c

W1输入/输出。

#### u8 w1\_touch\_bit(struct w1\_master \*dev, int bit)

生成写0或写1周期并采样电平。

**参数**

struct w1\_master \*dev

主设备

int bit

0-写入0、1-写入0读取级别

#### void w1\_write\_8(struct w1\_master \*dev, u8 byte)

写入8位。

**参数**

struct w1\_master \*dev

主设备

u8 byte

要写入的字节

#### u8 w1\_triplet(struct w1\_master \*dev, int bdir)

执行三元组-用于搜索ROM地址。

**参数**

struct w1\_master \*dev

主设备

int bdir

如果id\_bit和comp\_bit均为0，则写入的位

**说明**

返回位：

位0=id\_bit位1=comp\_bit位2=dir\_taken

如果设置了位0和1，则应重新启动搜索。

**返回**

位域-见上文

#### u8 w1\_read\_8(struct w1\_master \*dev)

读取8位。

**参数**

struct w1\_master \*dev

主设备

**返回**

读取的字节

#### void w1\_write\_block(struct w1\_master \*dev, const u8 \*buf, int len)

写入一系列字节。

**参数**

struct w1\_master \*dev

主设备

const u8 \*buf

要写入的数据的指针

int len

要写入的字节数

#### void w1\_touch\_block(struct w1\_master \*dev, u8 \*buf, int len)

触摸一系列字节。

**参数**

struct w1\_master \*dev

主设备

u8 \*buf

要写入的数据的指针

int len

要写入的字节数

#### u8 w1\_read\_block(struct w1\_master \*dev, u8 \*buf, int len)

读取一系列字节。

**参数**

struct w1\_master \*dev

主设备

u8 \*buf

要填充的缓冲区的指针

int len

要读取的字节数

**返回**

读取的字节数

#### int w1\_reset\_bus(struct w1\_master \*dev)

发出复位总线序列。

**参数**

struct w1\_master \*dev

主设备

**返回**

0=设备存在，1=没有设备存在或错误

#### int w1\_reset\_select\_slave(struct w1\_slave \*sl)

复位并选择从站

**参数**

struct w1\_slave \*sl

要选择的从站

**说明**

通过发送跳过ROM或ROM匹配来重置总线，然后选择从站。如果总线上只有一个设备注册，则发出跳过rom。必须持有w1主锁。

**返回**

0=成功，其他任何=错误

#### int w1\_reset\_resume\_command(struct w1\_master \*dev)

恢复而不是匹配ROM

**参数**

struct w1\_master \*dev

主设备

**说明**

当在多个从设备中存在一个需要进行多个连续命令并在每个命令之间重置的工作流时，此函数类似于先进行重置，然后对上一个匹配的ROM进行ROM匹配。优势在于跳过了匹配的ROM步骤，而选择了恢复命令。当然，从设备必须支持该命令。

如果总线只有一个从设备，则传统的ROM匹配会被跳过，并且为了提高效率会执行“跳过ROM”。在多从设备总线上，这当然行不通，但是恢复命令是下一个最好的选择。

必须保持w1主锁定状态。

#### void w1\_next\_pullup(struct w1\_master \*dev, int delay)

为强拉电设置注册

**参数**

struct w1\_master \*dev

主设备

int delay

毫秒为单位的时间

**说明**

在下一次写入操作之后放出指定持续时间的强拉电。并非所有硬件都支持强拉电。不支持强拉电的硬件将在写入操作后按给定的时间睡眠而没有强拉电。这是下一次写入的一次性请求，指定零将清除先前的请求。必须保持w1主锁定状态。

**返回**

0=成功，其他任何错误

#### void w1\_write\_bit(struct w1\_master \*dev, int bit)

生成写-0或写-1周期。

**参数**

struct w1\_master \*dev

主设备

int bit

写入的位

**说明**

仅在dev->bus\_master->touch\_bit为空时调用

#### void w1\_pre\_write(struct w1\_master \*dev)

预写操作

**参数**

struct w1\_master \*dev

主设备

**说明**

预写操作，目前仅支持强拉电。根据需要进行强拉电编程，如果请求了强拉电并且硬件支持它，则进行强拉电编程。

#### void w1\_post\_write(struct w1\_master \*dev)

后写选项

**参数**

struct w1\_master \*dev

主设备

**说明**

后写操作，目前仅支持强拉电。如果请求了强拉电，则在硬件支持它的情况下清除它，否则执行延迟，在任何情况下清除请求。

#### u8 w1\_read\_bit(struct w1\_master \*dev)

生成写-1周期并采样电平。

**参数**

struct w1\_master \*dev

主设备

**说明**

仅在dev->bus\_master->touch\_bit为空时调用

# RapidIO子系统指南

## 介绍

RapidIO是面向嵌入式市场的高速交换式布线互连技术，提供内存映射I/O和基于消息的交易，可通过交换式布线网络进行。RapidIO具有类似于PCI总线标准的标准化发现机制，可以简单地检测网络中的设备。

本文档适用于计划在新架构上支持RapidIO、编写新驱动程序或了解子系统内部的开发人员。

## 已知错误和限制

### 错误

空.

### 限制

不支持访问/管理RapidIO内存区域

不支持多个主机枚举

## RapidIO驱动程序接口

驱动程序被提供了一组调用，以便与子系统接口，以收集有关设备的信息、请求/映射内存区域资源以及管理邮箱/门铃。

函数

#### int rio\_local\_read\_config\_32(struct rio\_mport \* port, u32 offset, u32 \* data)

从本地配置空间读取32位

**参数**

struct rio\_mport \* port

主端口

u32 offset

进入本地配置空间

u32 \* data

指针读取数据

**说明**

从本地设备的配置空间中的指定偏移量读取32位数据。

#### int rio\_local\_write\_config\_32(struct rio\_mport \* port, u32 offset, u32 data)

向本地配置空间写入32位

**参数**

struct rio\_mport \* port

主端口

u32 offset

进入本地配置空间

u32 data

要写入的数据

**说明**

将32位数据写入本地设备的配置空间中的指定偏移量。

#### int rio\_local\_read\_config\_16(struct rio\_mport \* port, u32 offset, u16 \* data)

从本地配置空间读取16位

**参数**

struct rio\_mport \* port

主端口

u32 offset

进入本地配置空间

u16 \* data

指针读取数据

**说明**

从本地设备的配置空间中的指定偏移量读取16位数据。

#### int rio\_local\_write\_config\_16(struct rio\_mport \* port, u32 offset, u16 data)

向本地配置空间写入16位

**参数**

struct rio\_mport \* port

主端口

u32 offset

进入本地配置空间

u16 data

要写入的数据

**说明**

将16位数据写入本地设备的配置空间中的指定偏移量。

#### int rio\_local\_read\_config\_8(struct rio\_mport \* port, u32 offset, u8 \* data)

从本地配置空间读取8位

**参数**

struct rio\_mport \* port

主端口

u32 offset

进入本地配置空间

u8 \* data

指针读取数据

**说明**

从本地设备的配置空间中的指定偏移量读取8位数据。

#### int rio\_local\_write\_config\_8(struct rio\_mport \* port, u32 offset, u8 data)

向本地配置空间写入8位

**参数**

struct rio\_mport \* port

主端口

u32 offset

偏移量进入本地配置空间

u8 data

要写入的数据

**说明**

将8位数据写入本地设备配置空间中指定的偏移量。

#### int rio\_read\_config\_32(struct rio\_dev \* rdev，u32 offset，u32 \* data)

从配置空间读取32位

**参数**

struct rio\_dev \* rdev

RIO设备

u32 offset

进入设备配置空间的偏移量

u32 \* data

读取数据的指针

**说明**

从RIO设备的配置空间中指定偏移量读取32位数据。

#### int rio\_write\_config\_32(struct rio\_dev \* rdev，u32 offset，u32 data)

写入32位到配置空间

**参数**

struct rio\_dev \* rdev

RIO设备

u32 offset

进入设备配置空间的偏移量

u32 data

要写入的数据

**说明**

将32位数据写入RIO设备配置空间中指定的偏移量。

#### int rio\_read\_config\_16(struct rio\_dev \* rdev，u32 offset，u16 \* data)

从配置空间读取16位

**参数**

struct rio\_dev \* rdev

RIO设备

u32 offset

进入设备配置空间的偏移量

u16 \* data

读取数据的指针

**说明**

从RIO设备的配置空间中指定偏移量读取16位数据。

#### int rio\_write\_config\_16(struct rio\_dev \* rdev，u32 offset，u16 data)

写入16位到配置空间

**参数**

struct rio\_dev \* rdev

RIO设备

u32 offset

进入设备配置空间的偏移量

u16 data

要写入的数据

**说明**

将16位数据写入RIO设备配置空间中指定的偏移量。

#### int rio\_read\_config\_8(struct rio\_dev \* rdev，u32 offset，u8 \* data)

从配置空间读取8位

**参数**

struct rio\_dev \* rdev

RIO设备

u32 offset

进入设备配置空间的偏移量

u8 \* data

读取数据的指针

**说明**

从RIO设备的配置空间中指定偏移量读取8位数据。

#### int rio\_write\_config\_8(struct rio\_dev \* rdev，u32 offset，u8 data)

将8位写入配置空间

**参数**

struct rio\_dev \* rdev

RIO设备

u32 offset

进入设备配置空间的偏移量

u8 data

要写入的数据

**说明**

将8位数据写入RIO设备配置空间中指定的偏移量。

#### int rio\_send\_doorbell(struct rio\_dev \* rdev，u16 data)

向设备发送门铃消息

**参数**

struct rio\_dev \* rdev

RIO设备

u16 data

门铃消息数据

**说明**

向RIO设备发送门铃消息。门铃消息由data参数提供16位信息字段。

#### void rio\_init\_mbox\_res(struct resource \* res，int start，int end)

初始化RIO邮箱资源

**参数**

struct resource \* res

资源结构

int start

邮箱范围的起始位置

int end

邮箱范围的结束位置

**说明**

该函数用于初始化邮件箱资源的字段以供使用。它使用起始和结束参数初始化一系列邮箱。

#### void rio\_init\_dbell\_res(struct resource \* res，u16 start，u16 end)

初始化RIO门铃资源

**参数**

struct resource \* res

资源结构

u16 start

门铃范围的起始位置

u16 end

门铃范围的结束位置

**说明**

该函数用于初始化门铃资源的字段以供使用。它使用起始和结束参数初始化一系列门铃消息。

#### RIO\_DEVICE（dev，ven）

用于说明特定RIO设备的宏

**参数**

dev

16位RIO设备ID

ven

16位RIO供应商ID

**说明**

此宏用于创建匹配特定设备的struct rio\_device\_id。组装供应商和组装设备字段将设置为RIO\_ANY\_ID。

#### int rio\_add\_outb\_message(struct rio\_mport \* mport，struct rio\_dev \* rdev，int mbox，void \* buffer，size\_t len)

向出站邮箱队列添加RIO消息

**参数**

struct rio\_mport \* mport

包含出站队列的RIO主端口

struct rio\_dev \* rdev

将发送消息的RIO设备

int mbox

出站邮箱队列

void \* buffer

指向消息缓冲区的指针

size\_t len

消息缓冲区的长度

**说明**

将RIO消息缓冲区添加到传输的出站邮箱队列中。成功返回0。

#### int rio\_add\_inb\_buffer(struct rio\_mport \* mport，int mbox，void \* buffer)

将缓冲区添加到入站邮箱队列

**参数**

struct rio\_mport \* mport

包含入站邮箱的主端口

int mbox

入站邮箱编号

void \* buffer

指向消息缓冲区的指针

**说明**

将缓冲区添加到接收的入站邮箱队列中。成功返回0。

#### void \* rio\_get\_inb\_message(struct rio\_mport \* mport，int mbox)

从入站邮箱队列获取RIO消息

**参数**

struct rio\_mport \* mport

包含入站邮箱的主端口

int mbox

入站邮箱编号

**说明**

从入站邮箱队列获取RIO消息。成功返回0。

#### const char \* rio\_name(struct rio\_dev \* rdev)

获取唯一的RIO设备标识符

**参数**

struct rio\_dev \* rdev

RIO设备

**说明**

获取唯一的RIO设备标识符。返回设备标识符字符串。

#### void \* rio\_get\_drvdata(struct rio\_dev \* rdev)

获取RIO驱动程序特定的数据

**参数**

struct rio\_dev \* rdev

RIO设备

**说明**

获取RIO驱动程序特定的数据。返回指向驱动程序特定数据的指针。

#### void rio\_set\_drvdata(struct rio\_dev \* rdev, void \* data)

设置RIO驱动程序特定的数据

**参数**

struct rio\_dev \* rdev

RIO设备

void \* data

指向驱动程序特定数据的指针

**说明**

设置RIO驱动程序特定的数据。设备结构驱动程序数据指针设置为数据参数。

#### struct rio\_dev \* rio\_dev\_get(struct rio\_dev \* rdev)

增加RIO设备结构的引用计数

**参数**

struct rio\_dev \* rdev

正在引用的RIO设备

**说明**

每个对设备的活动引用应该被引用计数。

RIO设备的驱动程序通常应在它们绑定到设备时在它们的probe() 方法中记录这样的引用，并通过调用rio\_dev\_put() 在它们的断开(connect())方法中释放它们。

#### void rio\_dev\_put(struct rio\_dev \* rdev)

释放对RIO设备结构的使用

**参数**

struct rio\_dev \* rdev

断开连接的RIO设备

**说明**

当设备的用户完成它的使用时必须被调用。当设备的最后一个用户调用此函数时，设备的内存被释放。

#### int rio\_register\_driver(struct rio\_driver \* rdrv)

注册新的RIO驱动程序

**参数**

struct rio\_driver \* rdrv

要注册的RIO驱动程序结构

**说明**

将struct rio\_driver添加到已注册驱动程序的列表中。返回负值表示错误，否则返回0。如果没有发生错误，则驱动程序仍然注册，即使在注册过程中没有设备被声明。

#### void rio\_unregister\_driver(struct rio\_driver \* rdrv)

注销RIO驱动程序

**参数**

struct rio\_driver \* rdrv

要取消注册的RIO驱动程序结构

**说明**

从已注册的RIO驱动程序列表中删除struct rio\_driver,让其有机会通过调用其remove() 方法清理它负责的每个设备，并将这些设备标记为没有驱动程序。

#### u16 rio\_local\_get\_device\_id(struct rio\_mport \* port)

获取端口的基本/扩展设备ID

**参数**

struct rio\_mport \* port

从中获取设备ID的RIO主端口

**说明**

从实现主端口的本地设备中读取基本/扩展设备ID。返回8/16位设备ID。

#### int rio\_query\_mport(struct rio\_mport \* port, struct rio\_mport\_attr \* mport\_attr)

查询mport设备属性

**参数**

struct rio\_mport \* port

要查询的mport设备

struct rio\_mport\_attr \* mport\_attr

mport属性数据结构

**说明**

通过指向属性数据结构的指针返回指定mport的属性。

#### struct rio\_net \* rio\_alloc\_net(struct rio\_mport \* mport)

分配并初始化新的RIO网络数据结构

**参数**

struct rio\_mport \* mport

与RIO网络相关联的主端口

**说明**

分配RIO网络结构，初始化每个网络列表头，并将相关的主端口添加到关联主端口的网络列表中。成功返回RIO网络指针，失败返回NULL。

#### void rio\_local\_set\_device\_id(struct rio\_mport \* port, u16 did)

设置端口的基本/扩展设备ID

**参数**

struct rio\_mport \* port

RIO主端口

u16 did

要写入的设备ID值

**说明**

从设备中写入基本/扩展设备ID。

#### int rio\_add\_device(struct rio\_dev \* rdev)

将RIO设备添加到设备模型中

**参数**

struct rio\_dev \* rdev

RIO设备

**说明**

将RIO设备添加到全局设备列表中，并将RIO设备添加到RIO设备列表中。为RIO设备创建通用sysfs节点。

#### int rio\_request\_inb\_mbox(struct rio\_mport \* mport, void \* dev\_id, int mbox, int entries, void (\*minb) (struct rio\_mport \* mport, void \*dev\_id, int mbox, int slot)

请求入站邮箱服务

**参数**

struct rio\_mport \* mport

从中分配邮箱资源的RIO主端口

void \* dev\_id

特定于设备的指针以传递事件

int mbox

要声明的邮箱编号

int entries

入站邮箱队列中的条目数

void (\*) (struct rio\_mport \* mport, void \*dev\_id, int mbox, int slot) minb

当收到入站消息时执行的回调

**说明**

请求拥有入站邮箱资源并将回调函数绑定到资源。成功返回0。

#### int rio\_release\_inb\_mbox(struct rio\_mport \* mport, int mbox)

释放入站邮箱消息服务

**参数**

struct rio\_mport \* mport

从中释放邮箱资源的RIO主端口

int mbox

要释放的邮箱编号

**说明**

释放入站邮箱资源的所有权。如果请求已满足，则返回0。

#### int rio\_request\_outb\_mbox(struct rio\_mport \* mport, void \* dev\_id, int mbox, int entries, void (\*moutb) (struct rio\_mport \* mport, void \*dev\_id, int mbox, int slot)

请求出站邮箱服务

**参数**

struct rio\_mport \* mport

从中分配邮箱资源的RIO主端口

void \* dev\_id

特定于设备的指针以传递事件

int mbox

要申报的邮箱号码

int entries

出站邮箱队列中的条目数

void (\*) (struct rio\_mport \* mport, void \*dev\_id, int mbox, int slot) moutb

执行传出消息发送时的回调函数

**说明**：

请求传出邮箱资源的所有权并将回调函数绑定到该资源上。成功返回0。

#### int rio\_release\_outb\_mbox(struct rio\_mport \*mport，int mbox)

释放传出邮箱消息服务

**参数**：

struct rio\_mport \*mport

释放邮箱资源的RIO主机端口

int mbox

要释放的邮箱号

**说明**：

释放传入邮箱资源的所有权。如果请求被满足，则返回0。

#### int rio\_request\_inb\_dbell(struct rio\_mport \*mport，void \*dev\_id，u16 start，u16 end，void（\\*dinb）（struct rio\_mport \*mport，void \*dev\_id，u16 src，u16 dst，u16 info）

请求传入门铃消息服务

**参数**：

struct rio\_mport \*mport

分配与门铃资源关联的RIO主机端口

void \*dev\_id

传递事件的设备特定指针

u16 start

门铃信息范围开始

u16 end

门铃信息范围结尾

void（\\*）（struct rio\_mport \*mport，void \*dev\_id，u16 src，u16 dst，u16 info）dinb

接收到门铃时要执行的回调函数

**说明**：

请求传入门铃资源的所有权并将回调函数绑定到该资源上。如果请求被满足，则返回0。

#### int rio\_release\_inb\_dbell（struct rio\_mport \*mport，u16 start，u16 end）

释放传入门铃消息服务

**参数**：

struct rio\_mport \*mport

释放门铃资源的RIO主机端口

u16 start

门铃信息范围开始

u16 end

门铃信息范围结尾

**说明**：

释放传入门铃资源的所有权并从门铃事件列表中删除回调。如果请求被满足，则返回0。

#### struct resource \*rio\_request\_outb\_dbell(struct rio\_dev \*rdev，u16 start，u16 end)

请求传出门铃消息范围

**参数**：

struct rio\_dev \*rdev

分配与门铃资源关联的RIO设备

u16 start

门铃消息范围开始

u16 end

门铃消息范围结尾

**说明**：

请求拥有一个门铃消息范围。如果请求被满足，则返回资源；如果失败，则返回NULL。

#### int rio\_release\_outb\_dbell(struct rio\_dev \*rdev，struct resource \*res)

释放传出门铃消息范围

**参数**：

struct rio\_dev \*rdev

释放门铃资源的RIO设备

struct resource \*res

要释放的门铃资源

**说明**：

释放门铃消息范围的所有权。如果请求被满足则返回0。

#### int rio\_add\_mport\_pw\_handler(struct rio\_mport \*mport，void \*context，int（\\*pwcback）（struct rio\_mport mport，void \*context，union rio\_pw\_msg \*msg，int step）

将端口写入消息处理程序添加到mport特定的pw处理程序列表中

**参数**：

struct rio\_mport \*mport

绑定端口写入回调的RIO主机端口

void \*context

要传递事件的处理程序特定上下文

int（\\*）（struct rio\_mport mport，void \*context，union rio\_pw\_msg \*msg，int step）pwcback

接收到端口写入消息时要执行的回调函数

**说明**：

如果请求被满足，则返回0。

#### int rio\_del\_mport\_pw\_handler(struct rio\_mport \*mport，void \*context，int（\\*pwcback）（struct rio\_mport mport，void \*context，union rio\_pw\_msg \*msg，int step）

从mport特定的pw处理程序列表中删除端口写入消息处理程序

**参数**：

struct rio\_mport \*mport

从中解除绑定端口写入回调的RIO主机端口

void \*context

已注册处理程序特定的上下文以传递事件

int（\\*）（struct rio\_mport mport，void \*context，union rio\_pw\_msg \*msg，int step）pwcback

已注册的回调函数

**说明**：

如果请求被满足，则返回0。

#### int rio\_request\_inb\_pwrite(struct rio\_dev \*rdev，int（\\*pwcback）（struct rio\_dev \*rdev，union rio\_pw\_msg \*msg，int step）

为特定的RapidIO设备请求传入端口写入消息服务

**参数**：

struct rio\_dev \*rdev

要向其中注册端口写入回调例程的RIO设备

int（\\*）（struct rio\_dev \*rdev，union rio\_pw\_msg \*msg，int step）pwcback

接收到端口写入消息时要执行的回调例程

**说明**：

将端口写入回调函数绑定到RapidIO设备上。如果请求被满足，则返回0。

#### int rio\_release\_inb\_pwrite(struct rio\_dev \*rdev)

释放与特定RapidIO设备关联的传入端口写入消息服务

**参数**：

struct rio\_dev \*rdev

注册传入端口写入回调的RIO设备

**说明**：

从rio\_dev结构中删除回调。如果请求被满足，则返回0。

#### void rio\_pw\_enable(struct rio\_mport \*mport，int enable)

启用/禁用主端口的端口写入处理

**参数**：

struct rio\_mport \*mport

与端口写入处理相关的主端口

int enable

1=启用，0=禁用

#### int rio\_map\_inb\_region(struct rio\_mport \*mport，dma\_addr\_t local，u64 rbase，u32 size，u32 rflags)

映射传入内存区域。

**参数**：

struct rio\_mport \*mport

主端口。

dma\_addr\_t local

要映射的内存区域的物理地址

u64 rbase

为该窗口分配的RIO基地址

u32 size

内存区域的大小

u32 rflags

用于映射的标志。

**返回**

0-成功

此函数将从RIO空间创建映射到本地内存。

#### void rio\_unmap\_inb\_region(struct rio\_mport \*mport，dma\_addr\_t lstart)

取消映射传入内存区域

**参数**：

struct rio\_mport \*mport

主端口

dma\_addr\_t lstart

要取消映射的内存区域的物理地址

#### int rio\_map\_outb\_region(struct rio\_mport \*mport, u16 destid, u64 rbase, u32 size, u32 rflags, dma\_addr\_t \*local)

映射出站内存区域。

**参数**

struct rio\_mport \* mport

主端口。

u16 destid

目标ID窗口指向

u64 rbase

RIO基地址窗口翻译为

u32 size

内存区域的大小

u32 rflags

映射的标志。

dma\_addr\_t \* local

映射的物理内存区域地址

**返回**

0-成功。

此函数将创建从RIO空间到本地内存的映射。

#### void rio\_unmap\_outb\_region(struct rio\_mport \* mport，u16 destid，u64 rstart)

取消映射入站内存区域

**参数**

struct rio\_mport \* mport

主端口

u16 destid

映射指向的目标ID

u64 rstart

RIO基地址窗口翻译为

#### u32 rio\_mport\_get\_physefb(struct rio\_mport \* port，int local，u16 destid，u8 hopcount，u32 \* rmap)

帮助函数，返回物理层扩展特性块的寄存器偏移量。

**参数**

struct rio\_mport \* port

发出事务的主端口

int local

指示本地主端口或远程设备访问

u16 destid

设备的目标ID

u8 hopcount

到设备的交换机跳数

u32 \* rmap

指向存储寄存器映射类型信息的位置的指针

#### 结构rio\_dev \* rio\_get\_comptag(u32 comp\_tag，struct rio\_dev \* from)

通过组件标记开始或继续搜索RIO设备

**参数**

u32 comp\_tag

匹配的RIO组件标记

struct rio\_dev \* from

在搜索中找到的上一个RIO设备，或NULL进行新搜索

**说明**

迭代已知RIO设备的列表。如果找到具有匹配comp\_tag的RIO设备，则返回指向其设备结构的指针。否则，将返回NULL。通过将NULL传递给from参数来启动新搜索。否则，如果from不为NULL，则从全局列表上的下一个设备进行搜索。

#### int rio\_set\_port\_lockout(struct rio\_dev \* rdev，u32 pnum，int lock)

为交换机端口设置/清除LOCKOUT位（RIO EM 1.3）。

**参数**

struct rio\_dev \* rdev

指向RIO设备控制结构的指针

u32 pnum

要设置LOCKOUT位的交换机端口号

int lock

操作：设置（= 1）或清除（= 0）

#### int rio\_enable\_rx\_tx\_port(struct rio\_mport \* port，int local，u16 destid，u8 hopcount，u8 port\_num)

启用给定端口的输入接收器和输出发射器

**参数**

struct rio\_mport \* port

与RIO网络相关联的主端口

int local

local = 1选择本地端口否则会到达远端设备

u16 destid

要检查主机位的设备的目标ID

u8 hopcount

到达目标的跳数

u8 port\_num

在远程设备上启用的端口（交换机上的-编号）

**说明**

从General Control Command and Status Register（EXT\_PTR + 0x3C）返回0或1

#### int rio\_mport\_chk\_dev\_access(struct rio\_mport \* mport，u16 destid，u8 hopcount)

验证对指定设备的访问。

**参数**

struct rio\_mport \* mport

发送事务的主端口

u16 destid

网络中的设备目标ID

u8 hop count

进入网络的跳数

#### int rio\_inb\_pwrite\_handler(struct rio\_mport \* mport，union rio\_pw\_msg \* pw\_msg)

入站端口写入消息处理程序

**参数**

struct rio\_mport \* mport

与端口写入相关联的mport设备

union rio\_pw\_msg \* pw\_msg

指向入站端口写入消息的指针

**说明**

处理入站端口写入消息。如果请求已满足，则返回0。

#### u32 rio\_mport\_get\_efb(struct rio\_mport \* port，int local，u16 destid，u8 hopcount，u32 from)

获取指向下一个扩展特性块的指针

**参数**

struct rio\_mport \* port

发出事务的主端口

int local

指示本地主端口或远程设备访问

u16 destid

设备的目标ID

u8 hopcount

到设备的交换机跳数

u32 from

当前扩展特性块标题的偏移量（如果从0开始则从ExtFeaturePtr开始）

#### u32 rio\_mport\_get\_feature(struct rio\_mport \* port，int local，u16 destid，u8 hopcount，int ftr)

查询设备的扩展功能

**参数**

struct rio\_mport \* port

发出事务的主端口

int local

指示本地主端口或远程设备访问

u16 destid

设备的目标ID

u8 hopcount

到设备的交换机跳数

int ftr

扩展功能代码

**说明**

告诉设备是否支持给定的RapidIO功能。返回设备的RIO配置空间中所请求的扩展功能块的偏移量，或在设备不支持它的情况下返回0。

#### struct rio\_dev \* rio\_get\_asm(u16 vid，u16 did，u16 asm\_vid，u16 asm\_did，struct rio\_dev \* from)

通过vid / did / asm\_vid / asm\_did开始或继续搜索RIO设备

**参数**

u16 vid

要匹配的RIO vid或RIO\_ANY\_ID以匹配所有vid

u16 did

要匹配的RIO did或RIO\_ANY\_ID以匹配所有did

u16 asm\_vid

要匹配的RIO asm\_vid或RIO\_ANY\_ID以匹配所有asm\_vids

u16 asm did

要匹配的RIO asm\_did或RIO\_ANY\_ID以匹配所有asm\_dids

struct rio\_dev \* from

在搜索中找到的上一个RIO设备，或null进行新搜索

**说明**

迭代已知的RIO设备列表。如果找到具有匹配vid，did，asm\_vid，asm\_did的RIO设备，则增加对该设备的引用计数，并返回指向其设备结构的指针。否则，将返回NULL。通过将NULL传递给from参数来启动新搜索。否则，如果from不为NULL，则搜索将从全局列表上的下一个设备继续进行。始终对from的引用计数进行递减，如果它不为NULL。

#### struct rio\_dev\* rio\_get\_device(u16 vid, u16 did, struct rio\_dev\* from)

通过vid/did开始或继续搜索RIO设备

**参数**

u16 vid

匹配的RIO VID或RIO\_ANY\_ID以匹配所有VID

u16 did

匹配的RIO DID或RIO\_ANY\_ID以匹配所有DID

struct rio\_dev\* from

之前在搜索中找到的RIO设备，或者为新搜索的NULL

**说明**

迭代已知RIO设备列表。如果找到一个具有匹配的vid和did的RIO设备，则将对该设备的引用计数增加，并返回指向其设备结构的指针。否则返回NULL。通过将NULL传递给from参数启动新搜索。否则，如果from不为NULL，则从全局列表中的下一个设备继续搜索。如果from不是NULL，则始终将from的引用计数递减。

#### int rio\_lock\_device(struct rio\_mport\* port, u16 destid, u8 hopcount, int wait\_ms)

获得指定设备的主机设备锁定

**参数**

struct rio\_mport\* port

发送交易的主端口

u16 destid

设备/交换机的目标ID

u8 hopcount

到达交换机的跳数

int wait\_ms

最长等待时间（以毫秒为单位）（0=无超时）

**说明**

尝试获得指定设备的主机设备锁定，如果成功返回0或者如果超时返回EINVAL。

#### int rio\_unlock\_device(struct rio\_mport\* port, u16 destid, u8 hopcount)

释放指定设备的主机设备锁定

**参数**

struct rio\_mport\* port

发送交易的主端口

u16 destid

设备/交换机的目标ID

u8 hopcount

到达交换机的跳数

**说明**

如果成功释放设备锁定，则返回0或如果失败返回EINVAL。

#### int rio\_route\_add\_entry(struct rio\_dev\* rdev, u16 table, u16 route\_destid, u8 route\_port, int lock)

向交换机路由表中添加一条路由条目

**参数**

struct rio\_dev\* rdev

RIO设备

u16 table

路由表ID

u16 route\_destid

要路由的目标ID

u8 route\_port

要路由的端口号

int lock

在开关设备标志上应用硬件锁定（1=锁定，0=不使用锁定）

**说明**

如果可用，则调用交换机特定的add\_entry()方法将路由条目添加到交换机路由表中。否则，使用由RapidIO规范定义的标准RT更新方法。如果交换机具有每个端口路由表，则可以使用table参数选择特定的路由表，否则可以通过在table中传递RIO\_GLOBAL\_TABLE来使用标准（或全局）表。

返回成功则为0，失败则为-EINVAL。

#### int rio\_route\_get\_entry(struct rio\_dev\* rdev, u16 table, u16 route\_destid, u8\* route\_port, int lock)

从交换机路由表中读取条目

**参数**

struct rio\_dev\* rdev

RIO设备

u16 table

路由表ID

u16 route\_destid

要路由的目标ID

u8\* route\_port

指向读取端口号的指针

int lock

在开关设备标志上应用硬件锁定（1=锁定，0=不使用锁定）

**说明**

如果可用，则调用交换机特定的get\_entry()方法从交换机路由表中提取路由条目。否则，使用由RapidIO规范定义的标准RT读取方法。如果交换机具有每个端口的路由表，则可以使用table参数选择特定的路由表，否则可以通过在table中传递RIO\_GLOBAL\_TABLE来使用标准（或全局）表。

返回成功则为0，失败则为-EINVAL。

#### int rio\_route\_clr\_table(struct rio\_dev\* rdev, u16 table, int lock)

清除交换机路由表

**参数**

struct rio\_dev\* rdev

RIO设备

u16 table

路由表ID

int lock

在开关设备标志上应用硬件锁定（1=锁定，0=不使用锁定）

**说明**

如果可用，则调用交换机特定的clr\_table()方法来清除交换机路由表。否则，使用由RapidIO规范定义的标准RT写入方法。如果交换机具有每个端口的路由表，则可以使用table参数选择特定的路由表，否则可以通过在table中传递RIO\_GLOBAL\_TABLE来使用标准（或全局）表。

返回成功则为0，失败则为-EINVAL。

#### struct dma\_chan\* rio\_request\_mport\_dma(struct rio\_mport\* mport)

请求与指定本地RapidIO mport设备相关的支持RapidIO的DMA通道。

**参数**

struct rio\_mport\* mport

执行DMA数据传输的RIO mport

**说明**

返回指向已分配DMA通道的指针，如果失败则返回NULL。

#### struct dma\_chan\* rio\_request\_dma(struct rio\_dev\* rdev)

请求支持指定目标RapidIO设备的RapidIO支持的DMA通道。

**参数**

struct rio\_dev\* rdev

与DMA传输关联的RIO设备

**说明**

返回指向已分配DMA通道的指针，如果失败则返回NULL。

#### void rio\_release\_dma(struct dma\_chan\* dchan)

释放指定的DMA通道

**参数**

struct dma\_chan\* dchan

要释放的DMA通道

#### struct dma\_async\_tx\_descriptor\* rio\_dma\_prep\_xfer(struct dma\_chan\* dchan, u16 destid, struct rio\_dma\_data\* data, enum dma\_transfer\_direction direction, unsigned long flags)

用DMAENGINE定义的device\_prep\_slave\_sg回调的RapidIO特定包装器。

**参数**

struct dma\_chan\* dchan

要配置的DMA通道

u16 destid

目标RapidIO设备的目标ID

struct rio\_dma\_data \* data

RIO特定的数据说明符

enum dma\_transfer\_direction direction

DMA数据传输方向（从或向设备）

unsigned long flags

dmaengine定义的标志

**说明**

为指定的数据传输初始化RapidIO可用的DMA通道。使用DMA通道私有扩展来传递与远程目标RIO设备相关的信息。

**返回**

如果成功，则返回DMA事务说明符指针，

如果失败，则返回带错误值的指针或空指针。

#### struct dma\_async\_tx\_descriptor \* rio\_dma\_prep\_slave\_sg(struct rio\_dev \* rdev, struct dma\_chan \* dchan, struct rio\_dma\_data \* data, enum dma\_transfer\_direction direction, unsigned long flags)

设备\_prep\_slave\_sg回调的RapidIO特定包装器，由DMAENGINE定义。

**参数**

struct rio\_dev \* rdev

RIO设备控制结构

struct dma\_chan \* dchan

要配置的DMA通道

struct rio\_dma\_data \* data

RIO特定的数据说明符

enum dma\_transfer\_direction direction

DMA数据传输方向（从或向设备）

unsigned long flags

dmaengine定义的标志

**说明**

为指定的数据传输初始化RapidIO可用的DMA通道。使用DMA通道私有扩展来传递与远程目标RIO设备相关的信息。

**返回**

如果成功，则返回DMA事务说明符指针，

如果失败，则返回带错误值的指针或空指针。

#### int rio\_register\_scan(int mport\_id, struct rio\_scan \* scan\_ops)

注册枚举/发现方法的接口

**参数**

int mport\_id

要设置布局扫描程序的mport设备ID（RIO\_MPORT\_ANY =为所有可用的mport设置）

struct rio\_scan \* scan\_ops

枚举/发现操作结构体

**说明**

向RapidIO子系统注册枚举/发现操作，并将其附加到指定的mport设备（如果指定了RIO\_MPORT\_ANY，则附加到所有可用的mport）。

如果mport已经附加了枚举器，则返回错误。在RIO\_MPORT\_ANY的情况下，跳过具有有效扫描程序的mport（没有错误）。

#### int rio\_unregister\_scan(int mport\_id, struct rio\_scan \* scan\_ops)

从mport中删除枚举/发现方法

**参数**

int mport\_id

要取消注册布局扫描程序的mport设备ID（RIO\_MPORT\_ANY =适用于使用指定scan\_ops的所有mport）

struct rio\_scan \* scan\_ops

枚举/发现操作结构体

**说明**

删除分配给指定mport设备的枚举或发现方法。如果指定了RIO\_MPORT\_ANY，则从附有这些方法的所有mport中删除指定的操作。

## 内部结构

这一章包含RapidIO子系统的自动生成文档。

### 结构

#### struct rio\_switch

RIO开关信息

**定义**

struct rio\_switch {

struct list\_head node;

u8 \*route\_table;

u32 port\_ok;

struct rio\_switch\_ops \*ops;

spinlock\_t lock;

struct rio\_dev \*nextdev[0];

};

**成员**

node

全局开关列表中的节点

route\_table

开关路由表的副本

port\_ok

每个端口的状态（每个端口一个比特）- OK = 1或UNINIT = 0

ops

指向特定于开关的操作的指针

lock

用于串行化操作更新的锁

nextdev

指向下一个连接设备的每个端口指针的数组

#### struct rio\_switch\_ops

每个开关操作

**定义**

struct rio\_switch\_ops {

struct module \*owner;

int (\*add\_entry) (struct rio\_mport \*mport, u16 destid, u8 hopcount, u16 table, u16 route\_destid, u8 route\_port);

int (\*get\_entry) (struct rio\_mport \*mport, u16 destid, u8 hopcount, u16 table, u16 route\_destid, u8 \*route\_port);

int (\*clr\_table) (struct rio\_mport \*mport, u16 destid, u8 hopcount, u16 table);

int (\*set\_domain) (struct rio\_mport \*mport, u16 destid, u8 hopcount, u8 sw\_domain);

int (\*get\_domain) (struct rio\_mport \*mport, u16 destid, u8 hopcount, u8 \*sw\_domain);

int (\*em\_init) (struct rio\_dev \*dev);

int (\*em\_handle) (struct rio\_dev \*dev, u8 swport);

};

**成员**

owner

此结构体的模块所有者

add\_entry

用于特定于开关的路由添加函数的回调

get\_entry

用于特定于开关的路由获取函数的回调

clr\_table

用于特定于开关的清除路由表功能的回调

set\_domain

用于特定于开关的域设置函数的回调

get\_domain

用于特定于开关的域获取函数的回调

em\_init

用于特定于开关的错误管理初始函数的回调

em\_handle

用于特定于开关的错误管理处理程序函数的回调

**说明**

定义了初始化/控制特定RIO开关设备所需的操作。

#### struct rio\_dev

RIO设备信息

**定义**

struct rio\_dev {

struct list\_head global\_list;

struct list\_head net\_list;

struct rio\_net \*net;

bool do\_enum;

u16 did;

u16 vid;

u32 device\_rev;

u16 asm\_did;

u16 asm\_vid;

u16 asm\_rev;

u16 efptr;

u32 pef;

u32 swpinfo;

u32 src\_ops;

u32 dst\_ops;

u32 comp\_tag;

u32 phys\_efptr;

u32 phys\_rmap;

u32 em\_efptr;

u64 dma\_mask;

struct rio\_driver \*driver;

struct device dev;

struct resource riores[RIO\_MAX\_DEV\_RESOURCES];

int (\*pwcback) (struct rio\_dev \*rdev, union rio\_pw\_msg \*msg, int step);

u16 destid;

u8 hopcount;

struct rio\_dev \*prev;

atomic\_t state;

struct rio\_switch rswitch[0];

};

**成员**

global\_list

所有RIO设备列表中的节点

net\_list

作为网络一部分的RIO设备列表中的节点

net

这个设备是其中一部分的网络

do\_enum

枚举标志

did

设备ID

vid

供应商ID

device\_rev

设备版本

asm\_did

组件设备ID

asm\_vid

组件供应商ID

asm\_rev

组件修订版

efptr

扩展特性指针

pef

处理元素特征

swpinfo

交换机端口信息

src\_ops

源操作能力

dst\_ops

目标操作能力

comp\_tag

RIO组件标记

phys\_efptr

RIO设备扩展功能指针

phys\_rmap

LP-串行寄存器映射类型（1或2）

em\_efptr

RIO错误管理功能指针

dma\_mask

此设备实现的RIO地址位掩码

driver

声称拥有此设备的驱动程序

riores

该设备所拥有的RIO资源

pwcback

端口写回调函数，用于这个设备

destid

网络目的地ID（或与交换机相关的目的地ID）

hopcount

到该设备的跳数

prev

先前RIO设备连接到当前设备

state

设备状态

rswitch

struct rio\_switch（如果该设备有效）

#### struct rio\_msg

RIO message event

**定义**

struct rio\_msg {

struct resource \*res;

void (\*mcback) (struct rio\_mport \* mport, void \*dev\_id, int mbox, int slot);

};

**成员**

res

邮箱资源

mcback

消息事件回调

#### struct rio\_dbell

RIO门铃事件

**定义**

struct rio\_dbell {

struct list\_head node;

struct resource \*res;

void (\*dinb) (struct rio\_mport \*mport, void \*dev\_id, u16 src, u16 dst, u16 info);

void \*dev\_id;

};

**成员**

node

列表中的门铃事件节点

res

门铃资源

dinb

门铃事件回调

dev\_id

特定于设备的指针以传递事件

#### struct rio\_mport

RIO主端口信息

**定义**

struct rio\_mport {

struct list\_head dbells;

struct list\_head pwrites;

struct list\_head node;

struct list\_head nnode;

struct rio\_net \*net;

struct mutex lock;

struct resource iores;

struct resource riores[RIO\_MAX\_MPORT\_RESOURCES];

struct rio\_msg inb\_msg[RIO\_MAX\_MBOX];

struct rio\_msg outb\_msg[RIO\_MAX\_MBOX];

int host\_deviceid;

struct rio\_ops \*ops;

unsigned char id;

unsigned char index;

unsigned int sys\_size;

u32 phys\_efptr;

u32 phys\_rmap;

unsigned char name[RIO\_MAX\_MPORT\_NAME];

struct device dev;

void \*priv;

#ifdef CONFIG\_RAPIDIO\_DMA\_ENGINE;

struct dma\_device dma;

#endif;

struct rio\_scan \*nscan;

atomic\_t state;

unsigned int pwe\_refcnt;

};

**成员**

dbells

门铃事件列表

pwrites

端口写事件列表

node

主端口全局列表中的节点

nnode

网络主端口列表中的节点

net

连接到此mport的RIO网络

锁

同步列表操作的锁

iore

此主端口接口所拥有的I/O mem资源

riores

此主端口接口所拥有的RIO资源

inb\_msg

RIO入站消息事件**说明**符

outb\_msg

RIO出站消息事件**说明**符

host\_deviceid

与此主端口相关联的主机设备ID

ops

配置空间函数

id

端口ID，在所有端口中唯一

index

端口索引，在相同类型的所有端口接口中唯一

sys\_size

RapidIO公共传输系统的大小

phys\_efptr

RIO端口扩展特性指针

phys\_rmap

LP-Serial EFB寄存器映射类型（1或2）。

name

端口名称字符串

dev

与mport相关联的设备结构

priv

主端口私有数据

dma

与mport相关联的DMA设备

nscan

RapidIO网络枚举/发现操作

state

mport设备状态

pwe\_refcnt

端口写使能引用计数器，用于跟踪启用/禁用请求

#### struct rio\_net

RIO网络信息

**定义**

struct rio\_net {

struct list\_head node;

struct list\_head devices;

struct list\_head switches;

struct list\_head mports;

struct rio\_mport \*hport;

unsigned char id;

struct device dev;

void \*enum\_data;

void (\*release)(struct rio\_net \*net);

};

**成员**

node

RIO网络全局列表中的节点

devices

此网络中的设备列表

switches

此网络中的交换机列表

mports

访问此网络的主端口列表

hport

访问此网络的默认端口

id

RIO网络ID

dev

设备对象

enum\_data

特定于网络枚举器的私有数据

release

特定于枚举器的释放回调

#### struct rio\_mport\_attr

RIO 导入设备属性

**定义**

struct rio\_mport\_attr {

int flags;

int link\_speed;

int link\_width;

int dma\_max\_sge;

int dma\_max\_size;

int dma\_align;

};

**成员**

flags

mport设备功能标志

link\_speed

SRIO链接速度值（如RapidIO规范所定义）

link\_width

SRIO链接宽度值（如RapidIO规范所定义）

dma\_max\_sge

DMA通道可以处理的SG列表条目数

dma\_max\_size

单个DMA传输中的最大字节数（SG条目）

dma\_align

DMA操作的对齐移位（如其他DMA操作一样）

#### struct rio\_ops

低级RIO配置空间操作

**定义**

struct rio\_ops {

int (\*lcread) (struct rio\_mport \*mport, int index, u32 offset, int len, u32 \*data);

int (\*lcwrite) (struct rio\_mport \*mport, int index, u32 offset, int len, u32 data);

int (\*cread) (struct rio\_mport \*mport, int index, u16 destid, u8 hopcount, u32 offset, int len, u32 \*data);

int (\*cwrite) (struct rio\_mport \*mport, int index, u16 destid, u8 hopcount, u32 offset, int len, u32 data);

int (\*dsend) (struct rio\_mport \*mport, int index, u16 destid, u16 data);

int (\*pwenable) (struct rio\_mport \*mport, int enable);

int (\*open\_outb\_mbox)(struct rio\_mport \*mport, void \*dev\_id, int mbox, int entries);

void (\*close\_outb\_mbox)(struct rio\_mport \*mport, int mbox);

int (\*open\_inb\_mbox)(struct rio\_mport \*mport, void \*dev\_id, int mbox, int entries);

void (\*close\_inb\_mbox)(struct rio\_mport \*mport, int mbox);

int (\*add\_outb\_message)(struct rio\_mport \*mport, struct rio\_dev \*rdev, int mbox, void \*buffer, size\_t len);

int (\*add\_inb\_buffer)(struct rio\_mport \*mport, int mbox, void \*buf);

void \*(\*get\_inb\_message)(struct rio\_mport \*mport, int mbox);

int (\*map\_inb)(struct rio\_mport \*mport, dma\_addr\_t lstart, u64 rstart, u64 size, u32 flags);

void (\*unmap\_inb)(struct rio\_mport \*mport, dma\_addr\_t lstart);

int (\*query\_mport)(struct rio\_mport \*mport, struct rio\_mport\_attr \*attr);

int (\*map\_outb)(struct rio\_mport \*mport, u16 destid, u64 rstart, u32 size, u32 flags, dma\_addr\_t \*laddr);

void (\*unmap\_outb)(struct rio\_mport \*mport, u16 destid, u64 rstart);

};

**成员**

lcread

执行本地（主端口）配置空间读取的回调。

lcwrite

执行本地（主端口）配置空间写入的回调。

cread

执行网络配置空间读取的回调。

cwrite

执行网络配置空间写入的回调。

dsend

发送门铃消息的回调。

pwenable

启用/禁用端口写入消息处理的回调。

open\_outb\_mbox

初始化出站邮箱的回调。

close\_outb\_mbox

关闭出站邮箱的回调。

open\_inb\_mbox

初始化入站邮箱的回调。

close\_inb\_mbox

关闭入站邮箱的回调。

add\_outb\_message

向出站邮箱队列添加消息的回调。

add\_inb\_buffer

向入站邮箱队列添加缓冲区的回调。

get\_inb\_message

从入站邮箱队列获取消息的回调。

map\_inb

将RapidIO地址区域映射到本地内存空间的回调。

unmap\_inb

取消使用map\_inb()映射的RapidIO地址区域的回调。

query\_mport

查询mport设备属性的回调。

map\_outb

将出站地址区域映射到本地内存空间的回调。

unmap\_outb

取消使用map\_outb()映射的出站RapidIO地址区域的回调

#### struct rio\_driver

RIO 驱动信息

**定义**

struct rio\_driver {

struct list\_head node;

char \*name;

const struct rio\_device\_id \*id\_table;

int (\*probe) (struct rio\_dev \* dev, const struct rio\_device\_id \* id);

void (\*remove) (struct rio\_dev \* dev);

void (\*shutdown)(struct rio\_dev \*dev);

int (\*suspend) (struct rio\_dev \* dev, u32 state);

int (\*resume) (struct rio\_dev \* dev);

int (\*enable\_wake) (struct rio\_dev \* dev, u32 state, int enable);

struct device\_driver driver;

};

**成员**

node

驱动列表中的节点

name

RIO驱动程序名称

id\_table

与该驱动程序相关的RIO设备ID

probe

插入RIO设备

remove

删除RIO设备

shutdown

关闭通知回调

suspend

RIO设备挂起

resume

RIO设备唤醒

enable\_wake

RIO设备启用唤醒事件

driver

LDM驱动程序结构体

**说明**

为插入/删除和电源管理目的提供有关RIO设备驱动程序的信息

#### struct rio\_scan

RIO枚举和发现操作

**定义**

struct rio\_scan {

struct module \*owner;

int (\*enumerate)(struct rio\_mport \*mport, u32 flags);

int (\*discover)(struct rio\_mport \*mport, u32 flags);

};

**成员**

owner

此结构的模块所有者

enumerate

执行RapidIO架构枚举的回调。

discover

执行RapidIO 架构发现的回调。

#### struct rio\_scan\_node

注册快速IO枚举和发现方法与RapidIO核心的列表节点.

**定义**

struct rio\_scan\_node {

int mport\_id;

struct list\_head node;

struct rio\_scan \*ops;

};

**成员**

mport\_id

此枚举器服务的mport（net）的ID

node

在已注册的枚举器的全局列表中的节点

ops

RIO枚举和发现操作

### 枚举和发现

#### u16 rio\_destid\_alloc(struct rio\_net \* net)

为给定网络分配下一个可用的destID

**参数**

struct rio\_net \* net

RIO网络

**说明**

返回指定RIO网络的下一个可用设备目的ID。将分配的ID标记为正在使用。如果新的destID不可用，则返回RIO\_INVALID\_DESTID。

#### int rio\_destid\_reserve(struct rio\_net \* net, u16 destid)

保留指定的destID

**参数**

struct rio\_net \* net

RIO网络

u16 destid

保留的destID

**说明**

尝试保留指定的destID。如果成功则返回0。

#### void rio\_destid\_free(struct rio\_net \* net, u16 destid)

释放之前分配的destID

**参数**

struct rio\_net \* net

RIO网络

u16 destid

要释放的destID

**说明**

使指定的destID可用于使用。

#### u16 rio\_destid\_first(struct rio\_net \* net)

返回第一个正在使用的destID

**参数**

struct rio\_net \* net

RIO网络

#### u16 rio\_destid\_next(struct rio\_net \* net, u16 from)

返回下一个正在使用的destID

**参数**

struct rio\_net \* net

RIO网络

u16 from

要继续搜索的目标ID

#### u16 rio\_get\_device\_id(struct rio\_mport \* port, u16 destid, u8 hopcount)

获取设备的基本/扩展设备ID

**参数**

struct rio\_mport \* port

RIO主端口

u16 destid

设备的目标ID

u8 hopcount

到设备的跳数

**说明**

从设备读取基本/扩展设备ID。返回8/16位设备ID。

#### void rio\_set\_device\_id(struct rio\_mport \* port, u16 destid, u8 hopcount, u16 did)

设置设备的基本/扩展设备ID

**参数**

struct rio\_mport \* port

RIO主端口

u16 destid

设备的目标ID

u8 hopcount

到设备的跳数

u16 did

要写入的设备ID值

**说明**

从设备写入基本/扩展设备ID。

#### int rio\_clear\_locks(struct rio\_net \* net)

释放所有主机锁并发出枚举完成信号

**参数**

struct rio\_net \* net

要运行的RIO网络

**说明**

将枚举完成标志设置为每个设备上的组件标记CSR。完成后，释放每个设备上的主机锁。成功返回0，失败返回-EINVAL。

#### int rio\_enum\_host(struct rio\_mport \* port)

设置主机锁并初始化主机目标ID

**参数**

struct rio\_mport \* port

要发出事务的主端口

**说明**

使用主机设备ID值设置本地主机主端口锁和目标ID寄存器。主机设备ID值由平台提供。成功返回0，失败返回-1。

#### int rio\_device\_has\_destid(struct rio\_mport \* port, int src\_ops, int dst\_ops)

测试设备是否包含目的地ID寄存器

**参数**

struct rio\_mport \* port

发出事务的主端口

int src\_ops

RIO设备源操作

int dst\_ops

RIO设备目的地操作

**说明**

检查所提供的src\_ops和dst\_ops是否具有必要的事务功能，这表明设备是否将实现一个目的ID寄存器。如果为真返回1，如果为假返回0。

#### void rio\_release\_dev(struct device \* dev)

释放RIO设备结构

**参数**

struct device \* dev

与RIO设备结构相关联的LDM设备

**说明**

获取与RIO设备结构关联的RIO设备结构。释放RIO设备结构。

#### int rio\_is\_switch(struct rio\_dev \* rdev)

测试RIO设备是否具有交换功能

**参数**

struct rio\_dev \* rdev

RIO设备

**说明**

获取RIO设备处理元素特征寄存器内容，并测试交换功能。如果设备是交换机，则返回1，否则返回0。释放RIO设备结构。

#### struct rio\_dev \* rio\_setup\_device(struct rio\_net \* net, struct rio\_mport \* port, u16 destid, u8 hopcount, int do\_enum)

分配和设置RIO设备

**参数**

struct rio\_net \* net

RIO网络

struct rio\_mport \* port

发送事务的主端口

u16 destid

当前目标ID

u8 hopcount

当前跳数

int do\_enum

枚举/发现模式标志

**说明**

根据配置空间内容分配RIO设备并配置字段。如果设备有目的地ID寄存器，则在枚举模式下分配目的地ID，或在发现模式下从配置空间读取。如果设备具有交换功能，则分配交换机并相应地进行配置。成功返回指向RIO设备的指针，否则返回NULL。

#### int rio\_sport\_is\_active(struct rio\_dev \* rdev, int sp)

测试交换端口是否具有活动连接。

**参数**

struct rio\_dev \* rdev

RapidIO设备对象

int sp

交换端口号

**说明**

读取特定交换端口的端口错误状态CSR，以确定端口是否具有活动链接。如果端口处于活动状态则返回RIO\_PORT\_N\_ERR\_STS\_PORT\_OK，否则返回0。

#### u16 rio\_get\_host\_deviceid\_lock(struct rio\_mport \* port, u8 hopcount)

读取设备上的主机设备ID锁定CSR

**参数**

struct rio\_mport \* port

发送事务的主端口

u8 hopcount

到设备的跳数

**说明**

在枚举期间使用以读取RIO设备上的主机设备ID锁定CSR。返回锁定寄存器的值。

#### int rio\_enum\_peer(struct rio\_net \* net, struct rio\_mport \* port, u8 hopcount, struct rio\_dev \* prev, int prev\_port)

通过主端口递归枚举RIO网络

**参数**

struct rio\_net \* net

正在被枚举的RIO网络

struct rio\_mport \* port

发送事务的主端口

u8 hopcount

进入网络的跳数

struct rio\_dev \* prev

已连接到正在枚举的设备的上一个RIO设备

int prev\_port

之前的RIO设备上的端口

**说明**

递归枚举RIO网络。事务通过传递给端口进行发送。

#### int rio\_enum\_complete(struct rio\_mport \* port)

测试网络枚举是否完成

**参数**

struct rio\_mport \* port

发送事务的主端口

**说明**

测试非零值的PGCCSR发现位（枚举完成标志）。

如果枚举完成，则返回1；否则返回0。

#### int rio\_disc\_peer(struct rio\_net \* net, struct rio\_mport \* port, u16 destid, u8 hopcount, struct rio\_dev \* prev, int prev\_port)

通过主端口递归发现RIO网络

**参数**

struct rio\_net \* net

正在被发现的RIO网络

struct rio\_mport \* port

发送事务的主端口

u16 destid

网络中当前的目标ID

u8 hopcount

进入网络的跳数

struct rio\_dev \* prev

先前的rio\_dev

int prev\_port

之前的端口号

**说明**

递归发现RIO网络。事务通过传递给端口进行发送。

#### int rio\_mport\_is\_active(struct rio\_mport \* port)

测试主端口链接是否活动

**参数**

struct rio\_mport \* port

要测试的主端口

**说明**

读取主端口的端口错误状态CSR，以确定端口是否具有活动链接。如果主端口处于活动状态则返回RIO\_PORT\_N\_ERR\_STS\_PORT\_OK，否则返回0。

#### void rio\_update\_route\_tables(struct rio\_net \* net)

更新交换机的路由表

**参数**

struct rio\_net \* net

要进行更新的RIO网络

**说明**

对于每个枚举设备，请确保系统中的每个交换机都具有正确的路由条目。为在第一次枚举通过交换机时未知的设备添加路由。

#### void rio\_init\_em(struct rio\_dev \* rdev)

初始化RIO错误管理（用于交换机）

**参数**

struct rio\_dev \* rdev

RIO设备

**说明**

针对每个枚举的交换机，调用特定于设备的错误管理初始化例程（如果由交换机驱动程序提供）。

#### int rio\_enum\_mport(struct rio\_mport \* mport, u32 flags)

通过主端口开始枚举

**参数**

struct rio\_mport \* mport

发送事务的主端口

u32 flags

枚举控制标志

**说明**

开始枚举过程。如果有人列举了我们的主端口设备，那么放弃。如果没有并且我们有一个活动链接，那么开始递归对等枚举。如果枚举成功则返回0，如果枚举失败则返回-EBUSY。

#### void rio\_build\_route\_tables（struct rio\_net \* net）

从交换机路由条目生成路由表

**参数**

struct rio\_net \* net

要在该RIO网络上运行路由表扫描

**说明**

为每个交换机设备生成一个路由表，通过复制交换机中现有的路由条目来实现。

#### int rio\_disc\_mport（struct rio\_mport \* mport，u32 flags）

通过主端口启动发现过程

**参数**

struct rio\_mport \* mport

用于发送事务的主端口

u32 flags

发现控制标志

**说明**

开始发现过程。如果我们有一个活动链接，则等待枚举完成的信号（如果允许等待）。当枚举完成被发出信号时，开始递归对等发现。如果发现成功则返回0，如果失败则返回-EBUSY。

#### int rio\_basic\_attach（void）

**参数**

无参

**说明**

当此枚举/发现方法作为模块加载时，此函数为所有可用的RapidIO主端口设备注册其特定的枚举和发现例程。 “scan”命令行参数控制该模块自动启动RapidIO枚举/发现的能力。

成功则返回0，如果无法注册则返回-EIO。

此枚举/发现方法无法卸载，因此不提供匹配的cleanup\_module例程。

### 驱动功能

#### int rio\_setup\_inb\_dbell（struct rio\_mport \* mport，void \* dev\_id，struct resource \* res，void（\* dinb）（struct rio\_mport \* mport，void \* dev\_id，u16 src，u16 dst，u16 info）

绑定入站门铃回调

**参数**

struct rio\_mport \* mport

绑定门铃回调的RIO主端口

void \* dev\_id

设备特定指针传递事件

struct resource \* res

门铃消息资源

void（\*）（struct rio\_mport \* mport，void \* dev\_id，u16 src，u16 dst，u16 info）dinb

当接收到门铃时执行回调

**说明**

将门铃资源/回调对添加到端口的门铃事件列表中。如果请求已满足，则返回0。

#### int rio\_chk\_dev\_route（struct rio\_dev \* rdev，struct rio\_dev \*\* nrdev，int \* npnum）

验证到指定设备的路由。

**参数**

struct rio\_dev \* rdev

未响应RIO设备

struct rio\_dev \*\* nrdev

路径上的最后一个活动设备（及对应的RIO端口）

int \* npnum

nrdev在路径上到rdev的端口号

**说明**

沿着到指定RIO设备的路线进行，以确定路径上最后一个可用的设备（及相应的RIO端口）。

#### int rio\_chk\_dev\_access（struct rio\_dev \* rdev）

验证对指定设备的访问权限。

**参数**

struct rio\_dev \* rdev

指向RIO设备控制结构的指针

#### int rio\_get\_input\_status（struct rio\_dev \* rdev，int pnum，u32 \* lnkresp）

发送链接请求/输入状态控制符，并返回链路响应（如果请求）。

**参数**

struct rio\_dev \* rdev

要发出输入状态命令的RIO设备

int pnum

要发出命令的设备端口号

u32 \* lnkresp

链接伙伴的响应

#### int rio\_clr\_err\_stopped（struct rio\_dev \* rdev，u32 pnum，u32 err\_status）

清除端口故障停止状态。

**参数**

struct rio\_dev \* rdev

指向RIO设备控制结构的指针

u32 pnum

清除错误的交换机端口号

u32 err\_status

端口错误状态（如果0从设备读取寄存器）

**说明**

TODO：目前，此例程与idt\_gen3 RapidIO交换机设备指定的恢复过程不兼容。必须进行审查以实现通用恢复过程，该程序兼容可用设备的全部范围。IDT gen3交换机驱动程序现在实现了特定于硬件的错误处理程序，该程序向端口发出软件复位以重置ERR\_STOP位和ackID。

#### int rio\_std\_route\_add\_entry（struct rio\_mport \* mport，u16 destid，u8 hopcount，u16 table，u16 route\_destid，u8 route\_port）

使用RIO规范rev.1.3中定义的标准寄存器添加交换机路由表条目

**参数**

struct rio\_mport \* mport

要发出事务的主端口

u16 destid

设备的目标ID

u8 hopcount

到设备的交换机跳数

u16 table

路由表ID（全局或特定于端口）

u16 route\_destid

RT中的destID条目

u8 route\_port

指定destID的目标端口

#### int rio\_std\_route\_get\_entry（struct rio\_mport \* mport，u16 destid，u8 hopcount，u16 table，u16 route\_destid，u8 \* route\_port）

使用RIO规范rev.1.3中定义的标准寄存器读取与指定的destID相关联的交换机路由表条目（端口号）

**参数**

struct rio\_mport \* mport

要发出事务的主端口

u16 destid

设备的目标ID

u8 hopcount

到设备的交换机跳数

u16 table

路由表ID（全局或特定于端口）

u16 route\_destid

RT中的destID条目

u8 \* route\_port

指定的destID的目标端口返回

#### int rio\_std\_route\_clr\_table（struct rio\_mport \* mport，u16 destid，u8 hopcount，u16 table）

使用RIO规范rev.1.3中定义的标准寄存器清除swotch路由表。

**参数**

struct rio\_mport \* mport

要发出事务的主端口

u16 destid

设备的目标ID

u8 hopcount

设备的交换机跳数

u16 table

路由表的 ID（全局或端口特定）

#### struct rio\_mport \* rio\_find\_mport(int mport\_id)

通过其 ID 查找 RIO mport

**参数**

int mport\_id

mport 设备的编号（ID）

**说明**

给定 RIO mport 编号，目标 mport 位于全局 mport 列表中。如果找到该 mport，则返回指向其数据结构的指针。如果找不到 mport，则返回 NULL。

#### int rio\_mport\_scan(int mport\_id)

在指定的 mport 上执行枚举/发现

**参数**

int mport\_id

mport 设备的编号（ID）

#### RIO\_LOP\_READ(size, type, len)

生成 rio\_local\_read\_config\_\* 函数

**参数**

size

读取的配置空间大小（8、16、32 位）

type

值参数的 C 类型

len

读取到的配置空间长度（1、2、4 字节）

**说明**

生成 rio\_local\_read\_config\_\* 函数，用于访问本地设备上的配置空间寄存器。

#### RIO\_LOP\_WRITE(size, type, len)

生成 rio\_local\_write\_config\_\* 函数

**参数**

size

写入的配置空间大小（8、16、32 位）

type

值参数的 C 类型

len

写入到的配置空间长度（1、2、4 字节）

**说明**

生成 rio\_local\_write\_config\_\* 函数，用于访问本地设备上的配置空间寄存器。

#### RIO\_OP\_READ(size, type, len)

生成 rio\_mport\_read\_config\_\* 函数

**参数**

size

读取的配置空间大小（8、16、32 位）

type

值参数的 C 类型

len

读取到的配置空间长度（1、2、4 字节）

**说明**

生成 rio\_mport\_read\_config\_\* 函数，用于访问本地设备上的配置空间寄存器。

#### RIO\_OP\_WRITE(size, type, len)

生成 rio\_mport\_write\_config\_\* 函数

**参数**

size

写入的配置空间大小（8、16、32 位）

type

值参数的 C 类型

len

写入到的配置空间长度（1、2、4 字节）

**说明**

生成 rio\_mport\_write\_config\_\* 函数，用于访问本地设备上的配置空间寄存器。

### 设备型号支持

#### const struct rio\_device\_id \* rio\_match\_device(const struct rio\_device\_id \* id, const struct rio\_dev \* rdev)

告知 RIO 设备是否具有匹配的 RIO 设备 ID 结构

**参数**

const struct rio\_device\_id \* id

要与之进行匹配的 RIO 设备 ID 结构

const struct rio\_dev \* rdev

要与之进行匹配的 RIO 设备结构

**说明**

从驱动程序探测和总线匹配中使用，以检查 RIO 设备是否与 RIO 驱动程序提供的设备 ID 结构匹配。如果没有匹配，则返回匹配的 struct rio\_device\_id，否则返回 NULL。

#### int rio\_device\_probe(struct device \* dev)

告知 RIO 设备结构是否具有匹配的 RIO 设备 ID 结构

**参数**

struct device \* dev

要与之进行匹配的 RIO 设备结构

**说明**

若 drv 声明了 rio\_dev，则返回 0 并设置 rio\_dev->driver，否则返回错误

#### int rio\_device\_remove(struct device \* dev)

从系统中删除 RIO 设备

**参数**

struct device \* dev

要与之进行匹配的 RIO 设备结构

**说明**

从系统中删除 RIO 设备。如果它有相关的驱动程序，则运行驱动程序的 remove() 方法。然后更新引用计数。

#### int rio\_match\_bus(struct device \* dev, struct device\_driver \* drv)

告知 RIO 设备结构是否具有匹配的 RIO 驱动程序设备 ID 结构

**参数**

struct device \* dev

要与之进行匹配的标准设备结构

struct device\_driver \* drv

包含要与之匹配的 ID 的标准驱动程序结构

**说明**

由驱动程序使用，以检查系统中是否存在其受支持设备列表中的 RIO 设备。如果有匹配的 struct rio\_device\_id，则返回 1，否则返回 0。

#### int rio\_bus\_init(void)

向设备模型注册 RapidIO 总线

**参数**

void

无参数

**说明**

向 Linux 设备模型注册 RIO mport 设备类和 RIO 总线类型。

### PPC32 支持

#### int fsl\_local\_config\_read(struct rio\_mport \* mport, int index, u32 offset, int len, u32 \* data)

生成 MPC85xx 本地配置空间读取

**参数**

struct rio\_mport \* mport

RapidIO 主控端口信息

int index

RapdiIO 接口的 ID

u32 offset

配置空间内的偏移量

int len

维护事务的长度（以字节为单位）

u32 \* data

要读取的值

**说明**

生成 MPC85xx 本地配置空间读取。成功返回 0，否则返回 -EINVAL。

#### int fsl\_local\_config\_write(struct rio\_mport \* mport, int index, u32 offset, int len, u32 data)

生成 MPC85xx 本地配置空间写入

**参数**

struct rio\_mport \* mport

RapidIO 主控端口信息

int index

RapdiIO 接口的 ID

u32 offset

配置空间内的偏移量

int len

维护事务的长度（以字节为单位）

u32 data

要写入的值

**说明**

生成 MPC85xx 本地配置空间写入。成功返回 0，否则返回 -EINVAL。

#### int fsl\_rio\_config\_read(struct rio\_mport \* mport, int index, u16 destid, u8 hopcount, u32 offset, int len, u32 \* val)

生成 MPC85xx 读取维护事务

**参数**

struct rio\_mport \* mport

RapidIO 主控端口信息

int index

RapdiIO 接口的 ID

u16 destid

事务的目标 ID

u8 hopcount

目标设备的跳数

u32 偏移量

配置空间中的偏移量

int 长度

维护事务长度（以字节为单位）

u32 \* val

要读取的位置

**说明**

生成MPC85xx读取维护事务。成功返回0，失败返回-EINVAL。

#### int fsl\_rio\_config\_write(struct rio\_mport \* mport，int index，u16 destid，u8 hopcount，u32 offset，int len，u32 val)

生成MPC85xx写维护事务

**参数**

struct rio\_mport \* mport

RapidIO主端口信息

int index

RapdiIO接口的ID

u16 destid

事务目标ID

u8 hopcount

目标设备的跳数

u32 offset

配置空间中的偏移量

int len

维护事务长度（以字节为单位）

u32 val

要写入的值

**说明**

生成MPC85xx写入维护事务。成功返回0，失败返回-EINVAL。

#### int fsl\_rio\_setup(struct platform\_device \* dev)

设置Freescale PowerPC RapidIO接口

**参数**

struct platform\_device \* dev

平台设备指针

**说明**

初始化MPC85xx RapidIO硬件接口，使用系统特定信息配置主端口，并向RapidIO子系统注册主端口。

## 赞扬

以下人员直接或间接地为 RapidIO 子系统做出了贡献：

1. Matt Porter[mporter@kernel.crashing.org](mailto:mporter@kernel.crashing.org)
2. Randy Vinson[rvinson@mvista.com](mailto:rvinson@mvista.com)
3. Dan Malek[dan@embeddedalley.com](mailto:dan@embeddedalley.com)

以下人员为本文档做出了贡献：

Matt Porter[mporter@kernel.crashing.org](mailto:mporter@kernel.crashing.org)

# 编写s390通道设备驱动程序

## 介绍

本文档描述了可用于驱动基于 s390 的通道附加 I/O 设备的设备驱动程序的接口。这包括与硬件交互的接口和与通用驱动程序核心交互的接口。这些接口由 s390 公共 I/O 层提供。

本文档假定您熟悉与 s390 通道 I/O 架构相关的技术术语。有关此架构的描述，请参阅“z/Architecture：操作原理”，IBM 出版物 no. SA22-7832。

虽然 s390 系统上的大多数 I/O 设备通常是通过此处描述的通道 I/O 机制驱动的，但还有各种其他方法（如 diag 接口）。这些超出了本文档的范围。

s390 公共 I/O 层还提供对一些严格来说不属于 I/O 设备的设备的访问。此处也考虑了它们，尽管它们不是本文档的重点。

一些附加信息也可以在 Documentation/s390/driver-model.txt 下的内核源代码中找到。

## CSS总线

CSS总线包含系统上可用的子通道。它们分为几类：

用于系统的标准I / O子通道。它们在ccw总线上有一个子设备，并在下面说明。

绑定到vfio-ccw驱动程序的I / O子通道。请参见vfio-ccw：基本基础设施。

消息子通道。目前没有Linux驱动程序。

CHSC子通道（最多一个）。可以使用chsc子通道驱动程序发送异步chsc命令。

eADM子通道。用于与存储类内存通信。

## CCW总线

CCW总线通常包含s390系统中可用的大多数设备。以通道命令字（ccw）命名，用于寻址其设备的基本命令结构，ccw总线包含所谓的通道连接设备。它们通过I / O子通道寻址，可在css总线上看到。但是，通道连接设备的设备驱动程序永远不会直接与子通道交互，而只通过ccw总线上的I / O设备，即ccw设备进行交互。

### 用于通道连接设备的I / O功能

已将某些硬件结构转换为用于通用I / O层和设备驱动程序的C结构。有关表示此处的硬件结构的更多信息，请参阅操作原理。

#### struct ccw1

通道命令字

**定义**

struct ccw1 {

\_\_u8 cmd\_code;

\_\_u8 flags;

\_\_u16 count;

\_\_u32 cda;

};

**成员**

cmd\_code

命令代码

flags

标志，如IDA寻址等。

count

字节数

cda

数据地址

**说明**

CCW是构建与设备或控制单元执行操作的通道程序的基本结构。仅支持格式1通道命令字。

#### struct ccw0

通道命令字

**定义**

struct ccw0 {

\_\_u8 cmd\_code;

\_\_u32 cda : 24;

\_\_u8 flags;

\_\_u8 reserved;

\_\_u16 count;

};

**成员**

cmd\_code

命令代码

cda

数据地址

flags

标志，如IDA寻址等

reserved

将被忽略

count

字节数

**说明**

format-0 ccw结构

#### struct erw

扩展报告字

**定义**

struct erw {

\_\_u32 res0 : 3;

\_\_u32 auth : 1;

\_\_u32 pvrf : 1;

\_\_u32 cpt : 1;

\_\_u32 fsavf : 1;

\_\_u32 cons : 1;

\_\_u32 scavf : 1;

\_\_u32 fsaf : 1;

\_\_u32 scnt : 6;

\_\_u32 res16 : 16;

};

**成员**

res0

预定的

auth

授权检查

pvrf

需要路径验证的标志

cpt

通道路径超时

fsavf

失败的存储地址有效性标志

cons

并行感知

scavf

次要CCW地址有效标志

fsaf

失败存储地址格式

scnt

感知计数，如果cons == 1

res16

预定的

#### struct erw\_eadm

EADM子通道扩展报告字

**定义**

struct erw\_eadm {

\_\_u32 : 16;

\_\_u32 b : 1;

\_\_u32 r : 1;

\_\_u32 : 14;

};

**成员**

b

aob 错误

r

arsb 错误

#### struct sublog

子通道注销区域

**定义**

struct sublog {

\_\_u32 res0 : 1;

\_\_u32 esf : 7;

\_\_u32 lpum : 8;

\_\_u32 arep : 1;

\_\_u32 fvf : 5;

\_\_u32 sacc : 2;

\_\_u32 termc : 2;

\_\_u32 devsc : 1;

\_\_u32 serr : 1;

\_\_u32 ioerr : 1;

\_\_u32 seqc : 3;

};

**成员**

res0

保留

esf

扩展状态标志

lpum

最后使用的路径掩码

arep

附属报告

fvf

字段有效性标志

sacc

存储访问代码

termc

终止码

devsc

设备状态检查

serr

次要错误

ioerr

I/O错误警报

seqc

序列码

#### struct esw0

格式0扩展状态字 (ESW)

**定义**

struct esw0 {

struct sublog sublog;

struct erw erw;

\_\_u32 faddr[2];

\_\_u32 saddr;

};

**成员**

sublog

子通道注销

erw

扩展报告字

faddr

失败存储地址

saddr

次要CCW地址

#### struct esw1

格式1扩展状态字 (ESW)

**定义**

struct esw1 {

\_\_u8 zero0;

\_\_u8 lpum;

\_\_u16 zero16;

struct erw erw;

\_\_u32 zeros[3];

};

**成员**

zero0

保留零

lpum

最后使用的路径掩码

zero16

保留零

erw

扩展报告字

zeros

三个满字的零

#### struct esw2

格式2扩展状态字 (ESW)

**定义**

struct esw2 {

\_\_u8 zero0;

\_\_u8 lpum;

\_\_u16 dcti;

struct erw erw;

\_\_u32 zeros[3];

};

**成员**

zero0

保留零

lpum

最后使用的路径掩码

dcti

设备连接时间间隔

erw

扩展报告字

zeros

三个满字的零

#### struct esw3

格式3扩展状态字(ESW)

**定义**

struct esw3 {

\_\_u8 zero0;

\_\_u8 lpum;

\_\_u16 res;

struct erw erw;

\_\_u32 zeros[3];

};

**成员**

zero0

保留零

lpum

最后使用的路径掩码

res

保留

erw

扩展报告字

zeros

三个满字的零

#### struct esw\_eadm

EADM子通道扩展状态字(ESW)

**定义**

struct esw\_eadm {

\_\_u32 sublog;

struct erw\_eadm erw;

\_\_u32 : 32;

\_\_u32 : 32;

\_\_u32 : 32;

};

**成员**

sublog

子通道注销

erw

扩展报告字

#### struct irb

中断响应块

**定义**

struct irb {

union scsw scsw;

union {

struct esw0 esw0;

struct esw1 esw1;

struct esw2 esw2;

struct esw3 esw3;

struct esw\_eadm eadm;

} esw;

\_\_u8 ecw[32];

};

**成员**

scsw

子通道状态字

esw

扩展状态字

ecw

扩展控制字

**说明**

发生中断时传递给设备驱动程序的IRB。对于征求意见的中断，通用I/O层已经执行了字段是否有效的检查;字段不有效始终传递为0。如果发生单元检查，ECW可能包含传感器数据;如果设备不支持并行感知，则通用I/O层自己检索这些数据（因此设备驱动程序永远不需要执行基本感知）。对于非请求的中断，IRB按原样传递（除了感知数据（如果适用））。

#### struct ciw

命令信息字（CIW）布局

**定义**

struct ciw {

\_\_u32 et : 2;

\_\_u32 reserved : 2;

\_\_u32 ct : 4;

\_\_u32 cmd : 8;

\_\_u32 count : 16;

};

**成员**

et

输入类型

reserved

保留位

ct

命令类型

cmd

命令代码

count

命令计数

#### struct ccw\_dev\_id

CCW设备的唯一标识符

**定义**

struct ccw\_dev\_id {

u8 ssid;

u16 devno;

};

**成员**

ssid

ssid

子通道集ID

devno

设备编号

**说明**

此结构不是基于任何硬件结构直接构建的。硬件通过其设备编号和子通道来识别设备，子通道又由其ID标识。为了获取跨子通道集的ccw设备的唯一标识符，引入了struct ccw\_dev\_id。

#### int ccw\_dev\_id\_is\_equal(struct [ccw\_dev\_id](https://www.kernel.org/doc/html/latest/driver-api/s390-drivers.html" \l "c.ccw_dev_id" \o "ccw_dev_id) \*dev\_id1, struct [ccw\_dev\_id](https://www.kernel.org/doc/html/latest/driver-api/s390-drivers.html" \l "c.ccw_dev_id" \o "ccw_dev_id) \*dev\_id2)

比较两个ccw\_dev\_ids

**参数**

struct ccw\_dev\_id \*dev\_id1

ccw\_dev\_id

struct ccw\_dev\_id \*dev\_id2

另一个ccw\_dev\_id

**返回**

如果两个结构逐个字段相等，则为1，否则为0。

**上下文**

任何

#### u8 pathmask\_to\_pos(u8 mask)

查找路径掩码中最左侧位的位置

**参数**

u8 mask

至少设置了一个位的路径掩码

### CCW设备

要发起通道I/O的设备需要连接到ccw总线。与驱动程序核心的交互通过通用I/O层完成，该层提供了ccw设备和ccw设备驱动程序的抽象。

发起或终止通道I/O的函数都作用于ccw设备结构。设备驱动程序不得绕过这些函数，否则可能会发生奇怪的副作用

#### struct ccw\_device

通道附属设备

**定义**

struct ccw\_device {

spinlock\_t \*ccwlock;

struct ccw\_device\_id id;

struct ccw\_driver \*drv;

struct device dev;

int online;

void (\*handler) (struct ccw\_device \*, unsigned long, struct irb \*);

};

**成员**

ccwlock

设备锁指针

ID

此设备的ID

drv

此设备的ccw驱动程序

dev

嵌入设备结构

online

设备的在线状态

handler

中断处理程序

**说明**

处理程序是设备的成员，而不是驱动程序，因为驱动程序可以为不同的ccw设备（多子通道驱动程序）拥有不同的中断处理程序。

#### struct ccw\_driver

通道附加设备的设备驱动程序

**定义**

struct ccw\_driver {

struct ccw\_device\_id \*ids;

int (\*probe) (struct ccw\_device \*);

void (\*remove) (struct ccw\_device \*);

int (\*set\_online) (struct ccw\_device \*);

int (\*set\_offline) (struct ccw\_device \*);

int (\*notify) (struct ccw\_device \*, int);

void (\*path\_event) (struct ccw\_device \*, int \*);

void (\*shutdown) (struct ccw\_device \*);

int (\*prepare) (struct ccw\_device \*);

void (\*complete) (struct ccw\_device \*);

int (\*freeze)(struct ccw\_device \*);

int (\*thaw) (struct ccw\_device \*);

int (\*restore)(struct ccw\_device \*);

enum uc\_todo (\*uc\_handler) (struct ccw\_device \*, struct irb \*);

struct device\_driver driver;

enum interruption\_class int\_class;

};

**成员**

ids

由此驱动程序支持的ID

probe

探测时调用的函数

remove

移除时调用的函数

set\_online

在设置设备在线时调用

set\_offline

在设置设备离线时调用

notify

将设备状态更改通知驱动程序

path\_event

将通道路径事件通知驱动程序

shutdown

关闭设备时调用

prepare

为pm状态转换做准备

complete

撤消在准备时所做的工作

freeze

在休眠快照期间冻结的回调

thaw

撤消在freeze中所做的工作

restore

在休眠后的恢复回调

uc\_handler

单元检查处理程序的回调

driver

嵌入式设备驱动程序结构

int\_class

用于记帐中断的中断类别

#### int ccw\_device\_set\_offline(struct ccw\_device \*cdev)

禁用ccw设备进行I/O

**参数**

struct ccw\_device \* cdev

目标ccw设备

**说明**

如果有的话，此函数为cdev调用驱动程序的set\_offline（）函数，然后禁用cdev。

**返回**

成功返回0，失败返回负错误值。

**上下文**

启用，未持有ccw设备锁定

#### int ccw\_device\_set\_online(struct ccw\_device \*cdev)

启用ccw设备进行I/O

**参数**

struct ccw\_device \* cdev

目标ccw设备

**说明**

此函数首先启用cdev，然后调用驱动程序的set\_online（）函数，如果有的话，则禁用cdev 。如果set\_online（）返回错误，则再次禁用cdev。

**返回**

成功返回0，失败返回负错误值。

**上下文**

启用，未持有ccw设备锁定

#### struct ccw\_device \* get\_ccwdev\_by\_dev\_id（struct ccw\_dev\_id \* dev\_id）

从CCW设备ID获取设备

**参数**

struct ccw\_dev\_id \* dev\_id

要搜索的设备的ID

**说明**

此函数搜索连接到ccw总线的所有设备，查找与dev\_id匹配的设备。

**返回**

如果找到设备，则增加其引用计数并返回；否则返回NULL。

#### struct ccw\_device \* get\_ccwdev\_by\_busid（struct ccw\_driver \* cdrv，const char \* bus\_id）

从总线ID获取设备

**参数**

struct ccw\_driver \* cdrv

驱动程序所拥有的设备

const char \* bus\_id

要搜索的设备的总线ID

**说明**

此函数搜索cdrv所拥有的所有设备，查找具有与bus\_id匹配的总线ID的设备。

**返回**

如果找到匹配项，则会增加找到的设备的引用计数，并将其返回；否则返回NULL。

#### int ccw\_driver\_register（struct ccw\_driver \* cdriver）

注册ccw驱动程序

**参数**

struct ccw\_driver \* cdriver

要注册的驱动程序

**说明**

此函数主要是driver\_register（）的包装器。

**返回**

成功返回0，失败返回负错误值。

#### void ccw\_driver\_unregister（struct ccw\_driver \* cdriver）

取消注册ccw驱动程序

**参数**

struct ccw\_driver \* cdriver

要注销的驱动程序

**说明**

此函数主要是driver\_unregister（）的包装器。

#### int ccw\_device\_siosl（struct ccw\_device \* cdev）

启动记录

**参数**

struct ccw\_device \* cdev

ccw设备

**说明**

此函数用于在通道子系统中调用模型相关日志记录。

#### int ccw\_device\_set\_options\_mask（struct ccw\_device \* cdev，unsigned long flags）

设置一些选项并取消剩余选项

**参数**

struct ccw\_device \* cdev

要设定选项的设备

unsigned long flags

要设置的选项

**说明**

所有在flags中指定的标志都被设置，未在flags中指定的所有标志都被清除。

**返回**

成功返回0，-EINVAL返回无效的标志组合。

#### int ccw\_device\_set\_options（struct ccw\_device \* cdev，unsigned long flags）

设置一些选项

**参数**

struct ccw\_device \* cdev

要设定选项的设备

unsigned long flags

要设置的选项

**说明**

所有在flags中指定的标志都被设置，其他标志则不变。

**返回**

成功返回0，如果会导致无效的标志组合，则返回-EINVAL。

#### void ccw\_device\_clear\_options（struct ccw\_device \* cdev，unsigned long flags）

清除一些选项

**参数**

struct ccw\_device \* cdev

要清除选项的设备

unsigned long flags

要清除的选项

**说明**

所有在flags中指定的标志都被清除，未指定的标志则不变。

#### int ccw\_device\_is\_pathgroup（struct ccw\_device \* cdev）

确定该设备的路径是否分组

**参数**

struct ccw\_device \* cdev

ccw设备

**说明**

如果有路径组，则返回非零；否则返回零。

#### int ccw\_device\_is\_multipath（struct ccw\_device \* cdev）

确定设备是否以多路径模式运行

**参数**

struct ccw\_device \* cdev

ccw设备

**说明**

如果该设备以多路径模式运行，则返回非零；否则返回零。

#### int ccw\_device\_clear（struct ccw\_device \*cdev，unsigned long intparm）

终止I/O请求处理

**参数**

struct ccw\_device \* cdev

目标ccw设备

unsigned long intparm

在csch结束时返回的中断参数

**说明**

ccw\_device\_clear（）调用cdev的子通道上的csch。

**返回**

成功返回0，设备不可用返回-ENODEV，设备状态无效返回-EINVAL。

**上下文**

中断禁用，ccw设备锁保持

#### int ccw\_device\_start\_timeout\_key（struct ccw\_device \* cdev，struct ccw1 \* cpa，unsigned long intparm，\_\_u8 lpm，\_\_u8 key，unsigned long flags，int expires）

启动带超时和键的s390通道程序

**参数**

struct ccw\_device \* cdev

目标ccw设备

struct ccw1 \* cpa

通道程序的逻辑起始地址

unsigned long intparm

用户特定的中断参数；将被呈现回cdev的中断处理程序。允许设备驱动程序将中断与特定I/O请求关联起来。

\_\_u8 lpm

定义用于特定I/O请求的通道路径。值为0将使cio使用opm。

\_\_u8 key

用于I/O的存储密钥

unsigned long flags

附加标志；定义要执行的I/O处理操作。

int expires

jiffies的超时值

**说明**

启动S / 390通道程序。当中断到达时，将调用IRQ处理程序，立即，延迟（dev-end丢失或需要感知）或从未（未注册IRQ处理程序）。如果在超时指定的时间内通道程序未完成，则此函数通知设备驱动程序。如果超时，则通过xsch，hsch或csch终止通道程序，并使用ERR\_PTR（-ETIMEDOUT）调用设备的中断处理程序。中断处理程序将在此处指定回显intparm，除非通过后续调用ccw\_device\_halt（）或ccw\_device\_clear（）指定了另一个中断参数。

**返回**

如果操作成功，则返回0；如果设备繁忙或状态挂起，则返回-EBUSY；如果在lpm中未指定路径，则返回-EACCES；如果设备不可用，则返回-ENODEV。

**上下文**

中断禁用，ccw设备锁保持

#### int ccw\_device\_start\_key（struct ccw\_device \* cdev，struct ccw1 \* cpa，unsigned long intparm，\_\_u8 lpm，\_\_u8 key，unsigned long flags）

使用键开始s390通道程序

**参数**

struct ccw\_device \* cdev

目标ccw设备

struct ccw1 \* cpa

通道程序的逻辑起始地址

unsigned long intparm

用户特定的中断参数；将被呈现回cdev的中断处理程序。允许设备驱动程序将中断与特定I/O请求关联起来。

\_\_u8 lpm

定义用于特定I/O请求的通道路径。值为0将使cio使用opm。

\_\_u8 key

用于I/O的存储密钥

unsigned long flags

附加标志；定义要执行的I/O处理操作。

**说明**

启动S / 390通道程序。当中断到达时，将调用IRQ处理程序，立即，延迟（dev-end丢失或需要感知）或从未（未注册IRQ处理程序）。中断处理程序将在此处指定回显intparm，除非通过后续调用ccw\_device\_halt（）或ccw\_device\_clear（）指定了另一个中断参数。

**返回**

如果操作成功，则返回0；如果设备繁忙或状态挂起，则返回-EBUSY；如果在lpm中未指定路径，则返回-EACCES；如果设备不可用，则返回-ENODEV。

**上下文**

中断禁用，ccw设备锁保持

#### int ccw\_device\_start（struct ccw\_device \* cdev，struct ccw1 \* cpa，unsigned long intparm，\_\_u8 lpm，unsigned long flags）

开始s390通道程序

**参数**

struct ccw\_device \* cdev

目标ccw设备

struct ccw1 \* cpa

通道程序的逻辑起始地址

unsigned long intparm

用户特定的中断参数；将被呈现回cdev的中断处理程序。允许设备驱动程序将中断与特定I/O请求关联起来。

\_\_u8 lpm

定义用于特定I/O请求的通道路径。值为0将使cio使用opm。

unsigned long flags

附加标志；定义要执行的I/O处理操作。

**说明**

启动S / 390通道程序。当中断到达时，将调用IRQ处理程序，立即，延迟（dev-end丢失或需要感知）或从未（未注册IRQ处理程序）。中断处理程序将在此处指定回显intparm，除非通过后续调用ccw\_device\_halt（）或ccw\_device\_clear（）指定了另一个中断参数。

**返回**

如果操作成功，则返回0；如果设备繁忙或状态挂起，则返回-EBUSY；如果在lpm中未指定路径，则返回-EACCES；如果设备不可用，则返回-ENODEV。

**上下文**

中断禁用，ccw设备锁保持

#### int ccw\_device\_start\_timeout（struct ccw\_device \* cdev，struct ccw1 \* cpa，unsigned long intparm，\_\_u8 lpm，unsigned long flags，int expires）

开始一个带有超时的S/390通道程序。

**参数**：

struct ccw\_device \*cdev

目标ccw设备

struct ccw1 \*cpa

通道程序的逻辑开始地址

unsigned long intparm

用户指定的中断参数，将被呈现回CDEV的中断处理程序。允许设备驱动程序将中断与特定的I/O请求关联起来。

\_\_u8 lpm

定义用于特定I/O请求的通道路径。值为0将使CIO使用OPM。

unsigned long flags

附加标志；定义I/O处理的操作。

int expires

以jiffies为单位的超时值。

**说明**

开始一个S/390通道程序。当中断到达时，将调用IRQ处理程序，立即，延迟（dev-end丢失或需要感知）或永远（未注册IRQ处理程序）。如果通道程序在expires指定的时间内未完成，则此函数会通知设备驱动程序。如果超时发生，则通过xsch、hsch或csch终止通道程序，并使用包含ERR\_PTR（-ETIMEDOUT）的IRB调用设备的中断处理程序。中断处理程序将通过此处指定的intparm进行回显，除非通过后续调用ccw\_device\_halt()或ccw\_device\_clear()指定其他中断参数。

**返回**

如果操作成功，则返回0；如果设备忙或状态挂起，则返回-EBUSY；如果在LPM中没有指定路径是操作的，则返回-EACCES；如果设备不可操作，则返回-ENODEV。

**上下文**：

禁用中断，锁定ccw设备

#### int ccw\_device\_halt(struct ccw\_device \*cdev, unsigned long intparm)

停止I/O请求处理

**参数**：

struct ccw\_device \*cdev

目标ccw设备

unsigned long intparm

在hsch结束后返回的中断参数

**说明**

ccw\_device\_halt()在CDEV的子通道上调用hsch。中断处理程序将通过此处指定的intparm进行回显，除非通过后续调用ccw\_device\_clear()指定其他中断参数。

**返回**

成功返回0，设备不可操作返回-ENODEV，无效设备状态返回-EINVAL，设备忙或中断挂起返回-EBUSY。

**上下文**

禁用中断，锁定ccw设备

#### int ccw\_device\_resume(struct ccw\_device \*cdev)

恢复通道程序的执行

**参数**：

struct ccw\_device \*cdev

目标ccw设备

**说明**

ccw\_device\_resume()在CDEV的子通道上调用rsch。

**返回**

成功返回0，设备不可操作返回-ENODEV，无效设备状态返回-EINVAL，设备忙或中断挂起返回-EBUSY。

**上下文**

禁用中断，锁定ccw设备

#### struct ciw \*ccw\_device\_get\_ciw(struct ccw\_device \*cdev, \_\_u32 ct)

在扩展感知数据中搜索CIW命令。

**参数**

struct ccw\_device \*cdev

要检查的ccw设备

\_\_u32 ct

要查找的命令类型

**说明**

在SenseID期间，说明可用于设备的特殊命令的命令信息字（CIW）可能已存储在扩展感知数据中。此函数在扩展感知数据中搜索指定命令类型的CIW。

**返回**

如果没有存储扩展感知数据或找不到指定命令类型的CIW，则返回NULL，否则返回指向指定命令类型的CIW的指针。

#### \_\_u8 ccw\_device\_get\_path\_mask(struct ccw\_device \*cdev)

获取当前可用路径

**参数**

struct ccw\_device \*cdev

要查询的ccw设备

**返回**

如果没有设备的子通道可用，则返回0，否则返回ccw设备子通道当前可用路径的掩码。

#### struct channel\_path\_desc\_fmt0 \*ccw\_device\_get\_chp\_desc(struct ccw\_device \*cdev, int chp\_idx)

返回新分配的通道路径说明符

**参数**：

struct ccw\_device \*cdev

要获取说明符的设备

int chp\_idx

通道路径的索引

**说明**

成功返回与给定通道路径关联的通道路径说明数据的新分配副本。出错时返回NULL。

#### u8 \*ccw\_device\_get\_util\_str(struct ccw\_device \*cdev, int chp\_idx)

返回新分配的实用程序字符串

**参数**

struct ccw\_device \*cdev

要获取实用程序字符串的设备

int chp\_idx

通道路径的索引

**说明**

成功返回与给定通道路径关联的实用程序字符串的新分配副本。出错时返回NULL。

#### void ccw\_device\_get\_id(struct ccw\_device \*cdev, struct ccw\_dev\_id \*dev\_id)

获取ccw设备ID

**参数**

struct ccw\_device \*cdev

要获取ID的设备

struct ccw\_dev\_id \*dev\_id

要填充值的位置

#### int ccw\_device\_tm\_start\_timeout\_key(struct ccw\_device \*cdev, struct tcw \*tcw, unsigned long intparm, u8 lpm, u8 key, int expires)

执行启动函数

**参数**

struct ccw\_device \*cdev

要在其上执行启动函数的ccw设备

struct tcw \*tcw

要启动的传输命令字

unsigned long intparm

传递给中断处理程序的用户定义参数

u8 lpm

要使用的路径掩码

u8 key

用于存储访问的存储键

int expires

在多少个jiffies后中止请求的时间跨度

**说明**

在给定的ccw设备上启动tcw。成功返回零，否则返回非零。

#### int ccw\_device\_tm\_start\_key(struct ccw\_device \*cdev, struct tcw \*tcw, unsigned long intparm, u8 lpm, u8 key)

执行开始功能

**参数**

struct ccw\_device \*cdev

要执行开始功能的ccw设备

struct tcw \*tcw

要启动的传输命令字

unsigned long intparm

传递给中断处理程序的用户定义参数

u8 lpm

要使用的路径掩码

u8 key

用于存储访问的存储键

**说明**

在给定的ccw设备上启动tcw。成功返回零，否则返回非零。

#### int ccw\_device\_tm\_start(struct ccw\_device \* cdev，struct tcw \* tcw，unsigned long intparm，u8 lpm)

执行开始功能

**参数**

struct ccw\_device \*cdev

要执行开始功能的ccw设备

struct tcw \*tcw

要启动的传输命令字

unsigned long intparm

传递给中断处理程序的用户定义参数

u8 lpm

要使用的路径掩码

**说明**

在给定的ccw设备上启动tcw。成功返回零，否则返回非零。

#### int ccw\_device\_tm\_start\_timeout(struct ccw\_device \* cdev，struct tcw \* tcw，unsigned long intparm，u8 lpm，int expires)

执行开始功能

**参数**

struct ccw\_device \*cdev

要执行开始功能的ccw设备

struct tcw \*tcw

要启动的传输命令字

unsigned long intparm

传递给中断处理程序的用户定义参数

u8 lpm

要使用的路径掩码

int expires

超时后的时段，以jiffies为单位，用于中止请求

**说明**

在给定的ccw设备上启动tcw。成功返回零，否则返回非零。

#### int ccw\_device\_get\_mdc(struct ccw\_device \* cdev，u8 mask)

累计最大数据计数

**参数**

struct ccw\_device \*cdev

累计最大数据计数的ccw设备

u8 mask

要使用的路径掩码

**说明**

返回所有路径至少支持传输命令的64K字节块数。返回为0表示失败。

#### int ccw\_device\_tm\_intrg(struct ccw\_device \*cdev)

执行询问功能

**参数**

struct ccw\_device \*cdev

要执行询问功能的ccw设备

**说明**

在给定的ccw设备上执行询问功能。成功返回零，否则返回非零。

#### void ccw\_device\_get\_schid(struct ccw\_device \*cdev，struct subchannel\_id \*schid)

获取子通道ID

**参数**

struct ccw\_device \*cdev

要获取ID的设备

struct subchannel\_id \*schid

填充值的位置

### 通道测量设施

通道测量设施提供了一种收集测量数据的方式，该数据由通道子系统为每个通道附加设备提供。

#### struct cmbdata

用户空间的通道测量块数据

**定义**

struct cmbdata {

\_\_u64 size;

\_\_u64 elapsed\_time;

\_\_u64 ssch\_rsch\_count;

\_\_u64 sample\_count;

\_\_u64 device\_connect\_time;

\_\_u64 function\_pending\_time;

\_\_u64 device\_disconnect\_time;

\_\_u64 control\_unit\_queuing\_time;

\_\_u64 device\_active\_only\_time;

\_\_u64 device\_busy\_time;

\_\_u64 initial\_command\_response\_time;

};

**成员**

size

存储数据的大小

elapsed\_time

上次采样后的时间

ssch\_rsch\_count

ssch和rsch的数量

sample\_count

样本数

device\_connect\_time

设备连接时间

function\_pending\_time

函数挂起时间

device\_disconnect\_time

设备断开连接时间

control\_unit\_queuing\_time

控制单元排队时间

device\_active\_only\_time

设备仅处于活动状态的时间

device\_busy\_time

设备忙时的时间（ext.格式）

initial\_command\_response\_time

初始命令响应时间（ext.格式）

**说明**

所有值都以64位存储，以便在32位仿真模式下简化，所有时间值都规范化为纳秒。目前已知两种格式，其差异在于该结构的大小，即只有当激活扩展通道测量设施（首次在z990机器中提供）时，才设置最后两个成员。可能会添加更多字段，这将导致新的ioctl编号。

#### int enable\_cmf(struct ccw\_device \*cdev)

为特定设备开启通道测量

**参数**

struct ccw\_device \*cdev

要启用的ccw设备

**说明**

为cdev启用通道测量。如果对已启用通道测量的设备调用此函数，则会触发对测量数据的重置。

**返回**

成功返回0或负错误值。

**上下文**

非原子性的

#### int disable\_cmf(struct ccw\_device \*cdev)

为特定设备关闭通道测量

**参数**

struct ccw\_device \*cdev

要禁用的ccw设备

**返回**

成功返回0或负错误值。

**上下文**

非原子性的

#### u64 cmf\_read(struct ccw\_device \*cdev，int index)

从当前通道测量块读取一个值

**参数**

struct ccw\_device \*cdev

要读取的通道

int index

要读取的值的索引

**返回**

读取的值，如果无法读取，则为0。

**上下文**

任何

#### int cmf\_readall(str·uct ccw\_device \*cdev，struct cmbdata \*data)

读取当前通道测量块

**参数**

struct ccw\_device \*cdev

要读取的通道

struct cmbdata \*data

将要填充的数据块的指针

**返回**

成功时为0，否则为负错误值。

**上下文**

任何

## ccwgroup总线

ccwgroup总线仅包含由用户创建的人工设备。许多网络设备（例如qeth）实际上由几个ccw设备（如qeth的读取，写入和数据通道）组成。ccwgroup总线提供一种机制，可以将这些ccw设备作为从设备包含在其中，并与netdevice关联起来。

### ccw组设备

#### struct ccwgroup\_device

ccw组设备

#### 定义

struct ccwgroup\_device {

enum {

CCWGROUP\_OFFLINE,

CCWGROUP\_ONLINE,

} state;

unsigned int count;

struct device dev;

struct work\_struct ungroup\_work;

struct ccw\_device \*cdev[];

};

#### 成员

state

在线/离线状态

count

已连接从设备的数量

dev

嵌入式设备结构

ungroup\_work

用于解除ccw组设备

cdev

可变数量的从设备，按需分配

#### struct ccwgroup\_driver

ccw组设备驱动程序

#### 定义

struct ccwgroup\_driver {

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

void (\*remove) (struct ccwgroup\_device \*);

int (\*set\_online) (struct ccwgroup\_device \*);

int (\*set\_offline) (struct ccwgroup\_device \*);

void (\*shutdown)(struct ccwgroup\_device \*);

int (\*prepare) (struct ccwgroup\_device \*);

void (\*complete) (struct ccwgroup\_device \*);

int (\*freeze)(struct ccwgroup\_device \*);

int (\*thaw) (struct ccwgroup\_device \*);

int (\*restore)(struct ccwgroup\_device \*);

struct device\_driver driver;

struct ccw\_driver \*ccw\_driver;

};

#### 成员

setup

在设备创建期间调用以设置设备

remove

在删除时调用的函数

set\_online

在设备设置为在线时调用的函数

set\_offline

在设备设置为离线时调用的函数

shutdown

在设备关闭时调用的函数

prepare

准备PM状态转换

complete

撤消准备工作

freeze

冬眠快照期间的冻结回调

thaw

撤消在freeze中完成的工作

restore

在休眠后恢复的回调

driver

嵌入式驱动程序结构

ccw\_driver

支持的ccw\_driver（可选）

#### int ccwgroup\_set\_online(struct ccwgroup\_device \*gdev)

启用ccw组设备

**参数**

struct ccwgroup\_device \*gdev

目标ccw组设备

**说明**

此函数尝试将ccw组设备置于在线状态。

**返回**

成功返回0，失败返回负错误值。

#### int ccwgroup\_set\_offline(struct ccwgroup\_device \*gdev, bool call\_gdrv)

禁用ccw组设备

**参数**

struct ccwgroup\_device \*gdev

目标ccw组设备

bool call\_gdrv

调用已注册的gdrv set\_offline函数

**说明**

此函数尝试将ccw组设备置于离线状态。

**返回**

成功返回0，失败返回负错误值。

#### int ccwgroup\_create\_dev(struct device \*parent, struct ccwgroup\_driver \*gdrv, int num\_devices, const char \*buf)

创建并注册ccw组设备

**参数**

struct device \*parent

新设备的父设备

struct ccwgroup\_driver \*gdrv

新组设备的驱动程序

int num\_devices

从设备数量

const char \*buf

包含从设备总线ID的逗号分隔符的缓冲区

**说明**

创建并注册一个新的ccw组设备，作为父设备的子设备。从设备从buf中给定的总线ID列表中获取。

**返回**

成功返回0，失败返回错误代码。

**上下文**

非原子操作

#### int ccwgroup\_driver\_register(struct ccwgroup\_driver \*cdriver)

注册ccw组驱动程序

**参数**

struct ccwgroup\_driver \*cdriver

要注册的驱动程序

**说明**

此函数主要是driver\_register的封装。

#### void ccwgroup\_driver\_unregister(struct ccwgroup\_driver \*cdriver)

取消注册ccw组驱动程序

**参数**

struct ccwgroup\_driver \*cdriver

要取消注册的驱动程序

**说明**

此函数主要是driver\_unregister的封装。

#### int ccwgroup\_probe\_ccwdev(struct ccw\_device \*cdev)

从设备探测函数

**参数**

struct ccw\_device \*cdev

要检测的ccw设备

**说明**

这是一个虚拟探测函数，用于将ccw设备作为ccw组设备中的从设备。

**返回**

始终为0

#### void ccwgroup\_remove\_ccwdev(struct ccw\_device \*cdev)

从设备删除函数

**参数**

struct ccw\_device \*cdev

要移除的ccw设备

**说明**

这是一个删除函数，用于将ccw设备作为ccw组设备中的从设备。它将ccw设备设置为离线，并注销嵌入式ccw组设备。

## 通用接口

以下部分包含不仅由处理ccw设备的驱动程序使用，还由处理其他s390硬件的驱动程序使用的接口。

### 适配器中断

通用I/O层提供了用于处理适配器中断和中断向量的帮助程序函数。

#### int register\_adapter\_interrupt(struct airq\_struct \*airq)

注册适配器中断处理程序

**参数**

struct airq\_struct \*airq

指向适配器中断说明符的指针

**说明**

成功返回0，失败返回-EINVAL。

#### void unregister\_adapter\_interrupt(struct airq\_struct \*airq)

注销适配器中断处理程序

**参数**

struct airq\_struct \*airq

指向适配器中断说明符的指针

#### struct airq\_iv \*airq\_iv\_create(unsigned long bits, unsigned long flags, unsigned long \*vec)

创建中断向量

**参数**

unsigned long bits

中断向量中的位数

unsigned long flags

分配标志

unsigned long \*vec

如果AIRQ\_IV\_GUESTVEC，则为指向固定宿主内存的指针

**说明**

返回指向中断向量结构的指针

#### void airq\_iv\_release(struct airq\_iv \*iv)

释放中断向量

**参数**

struct airq\_iv \*iv

指向中断向量结构的指针

#### unsigned long airq\_iv\_alloc(struct airq\_iv \*iv, unsigned long num)

从中断向量中分配中断位

**参数**

struct airq\_iv \*iv

指向中断向量结构的指针

unsigned long num

要分配的连续中断位的数量

**说明**

返回分配的中断块中第一个中断的位数，如果没有位可用或未指定AIRQ\_IV\_ALLOC标志，则返回-1UL。

#### void airq\_iv\_free(struct airq\_iv \*iv, unsigned long bit, unsigned long num)

释放中断向量的中断位

**参数**

struct airq\_iv \*iv

指向中断向量结构的指针

unsigned long bit

释放第一个irq位的编号

unsigned long num

要释放的连续irq位的数量

#### unsigned long airq\_iv\_scan( struct airq\_iv \*  iv , unsigned long  start , unsigned long  end )

扫描中断向量以查找非零位

**参数**

struct airq\_iv \*iv

指向中断向量结构的指针

unsigned long start

开始搜索的位数

unsigned long end

结束搜索的位数

**说明**

返回下一个非零中断位的位数，如果扫描完成而未找到任何非零位，则返回-1UL。

# VME 设备驱动程序

## 驱动注册

与Linux内核中的其他子系统一样，VME设备驱动程序通过对VME子系统的调用进行注册，通常从设备init例程中调用。这是通过调用vme\_register\_driver()实现的。

必须向注册函数提供struct vme\_driver类型的结构体指针。以及你的驱动程序能够支持的设备的最大数量。

至少，应正确设置struct vme\_driver的“.name”、“.match”和“.probe”元素。'name'元素是指向保存设备驱动程序名称的字符串的指针。

'.match'函数允许控制应将哪些VME设备注册到驱动程序中。如果应该探测设备，则匹配函数应返回1，否则返回0。此示例匹配函数（来自vme\_user.c）将探测的设备数量限制为一个:

#define USER\_BUS\_MAX 1

...

static int vme\_user\_match(struct vme\_dev \*vdev)

{

if (vdev->id.num >= USER\_BUS\_MAX)

return 0;

return 1;

}

'.probe'元素应包含指针指向探针例程。探针程序作为struct vme\_dev指针传递。

在这里，“num”字段是该特定驱动程序的顺序设备ID。桥（或总线）号可使用dev->bridge->num访问。

还提供了一个函数从VME内核中取消注册驱动程序，该函数称为vme\_unregister\_driver()，通常应从设备驱动程序的退出例程中调用。

## 资源管理

驱动程序注册到VME核心后，提供的匹配例程将调用在注册期间指定的次数。如果匹配成功，则应返回非零值。零返回表示失败。对于所有成功的匹配，将调用相应驱动程序的探测例程。将指针传递给设备的设备结构。应保存此指针，因为它将需要请求VME资源。

驱动程序可以请求所有或多个主窗口（vme\_master\_request()）、从窗口（vme\_slave\_request()）和/或dma通道（vme\_dma\_request()）的所有权。与其允许设备驱动程序请求特定窗口或DMA通道（可能被不同的驱动程序使用）不同，API允许基于所需的驱动程序属性分配资源。对于从窗口，这些属性被分成需要在“aspace”中访问的VME地址空间和需要进行VME总线周期的“cycle”。主窗口在“width”中添加了另一组属性，指定所需的数据传输宽度。这些属性被定义为位掩码，因此可以请求单个窗口的任意属性组合，核心将分配满足要求的窗口，返回类型为vme\_resource的指针，当使用分配的资源时，应使用它来标识已分配的资源。对于DMA控制器，请求函数需要在路由属性中提供任何传输的可能方向。这通常是VME到MEM和/或MEM到VME，尽管某些硬件还支持VME到VME和MEM到MEM的传输，以及测试模式生成。如果找不到符合要求的未分配窗口，将返回NULL指针。

一旦不再需要它们，还提供了用于释放窗口分配的函数。这些函数（vme\_master\_free()、vme\_slave\_free()和vme\_dma\_free()）应传递分配资源时提供的指针。

## 主窗口

主窗口提供从本地处理器[s]到VME总线的访问。可用的窗口数量和可用的访问模式取决于底层芯片组。必须在使用窗口之前对其进行配置。

### 主窗口配置

一旦分配了主窗口，可使用vme\_master\_set()对其进行配置，使用vme\_master\_get()检索当前设置。地址空间、传输宽度和循环类型与资源管理中说明的相同，但其中一些选项是互斥的。例如，只能指定一个地址空间。

### 主窗口访问

函数vme\_master\_read()可用于从配置的主窗口中读取，vme\_master\_write()用于写入配置的主窗口。

除了简单的读写操作外，还提供了vme\_master\_rmw()函数用于执行读-修改-写事务。VME窗口的一部分也可以通过vme\_master\_mmap()映射到用户空间内存中。

## 从窗口

从属窗口为 VME 总线上的设备提供对本地内存映射部分的访问。可用窗口的数量和可以使用的访问模式取决于底层芯片组。窗口必须先配置才能使用。

### 从窗口配置

安排了从窗口后，就可以使用vme\_slave\_set()来配置它，使用vme\_slave\_get()来检索当前设置。

地址空间、传输宽度和周期类型与资源管理中说明的相同，但某些选项是互斥的。例如，只能指定一个地址空间。

### 从窗口缓冲区分配

提供了函数允许用户分配（vme\_alloc\_consistent()）和释放（vme\_free\_consistent()）连续缓冲区，这些缓冲区将可以被VME桥访问。这些函数不一定非得使用，也可以使用其他方法来分配缓冲区，但必须注意确保它们是连续的并且可以被VME桥访问。

### 从窗口访问

从窗口映射本地内存到VME总线上，应使用标准方法来访问内存。

## DMA通道

VME DMA传输提供了运行链接列表DMA传输的功能。API引入了DMA列表的概念。每个DMA列表是一个可传递给DMA控制器的链接列表。可以创建、扩展、执行、重用和销毁多个列表。

### 列表管理

提供了vme\_new\_dma\_list()函数来创建和vme\_dma\_list\_free()函数来销毁DMA列表。执行列表不会自动销毁列表，因此可以重复使用列表进行重复任务。

### 列表入口

可以使用vme\_dma\_list\_add()将项目添加到列表中（在调用此函数之前需要创建源和目标属性，这在“传输属性”中已介绍）。

注意：直到将条目添加到DMA列表中才会检查传输源和目标的详细属性，请求DMA通道仅检查控制器预期传输数据的方向。因此，这个调用可能会返回错误，例如如果源或目标在不支持的VME地址空间中。

### 传输属性

源和目标的属性与将项目添加到列表中的操作分别处理。这是因为不同类型的源和目标需要不同的属性。有尽可能多的函数可用于创建PCI、VME和模式源和目标的属性：

PCI源或目标：vme\_dma\_pci\_attribute()

VME源或目标：vme\_dma\_vme\_attribute()

模式源：vme\_dma\_pattern\_attribute()

应使用vme\_dma\_free\_attribute()函数来释放属性。

### 列表执行

vme\_dma\_list\_exec()函数将列表排队执行，只有列表执行完毕后，该函数才会返回。

## 中断

VME API提供了将回调附加到特定VME级别和状态ID组合以及生成具有特定VME级别和状态ID的VME中断的函数。

### 附加中断处理程序

vme\_irq\_request()函数可用于附加，vme\_irq\_free()函数可用于释放特定的VME级别和状态ID组合。任何给定的组合只能分配单个回调函数。提供了一个空指针参数，其值将传递给回调函数，此指针的使用由用户定义。回调参数如下所示。编写回调函数时要注意，回调函数在中断上下文中运行：

void callback(int level, int statid, void \*priv);

### 中断产生

vme\_irq\_generate()函数可用于在给定的VME级别和VME状态ID上生成VME中断。

## 位置监视器

VME API提供以下功能来配置位置监视器。

### 位置监视器管理

vme\_lm\_request()函数用于请求位置监视器的使用，vme\_lm\_free()函数用于在不需要它们时释放它们。每个块可以提供多个位置监视器，监视相邻的位置。vme\_lm\_count()函数可用于确定提供了多少个位置。

### 位置**监视器**配置

一旦分配了位置监视器的块，vme\_lm\_set()函数用于配置位置监视器的位置和模式。vme\_lm\_get()函数可用于检索现有设置。

### 位置监视器使用

vme\_lm\_attach()函数允许将回调附加到每个位置监视器位置，并且vme\_lm\_detach()函数允许将回调从每个位置监视器位置中分离。每个位置监视器可以监视多个相邻位置。回调函数声明如下：

void callback(void \*data);

## 插槽检测

函数vme\_slot\_num() 返回所提供桥的插槽ID。

## 总线检测

函数vme\_bus\_num() 返回所提供桥的总线ID。

## 虚拟机接口

### struct vme\_dev

代表虚拟机设备的结构体

**定义**

struct vme\_dev {

int num;

struct vme\_bridge \*bridge;

struct device dev;

struct list\_head drv\_list;

struct list\_head bridge\_list;

};

**成员**

num

设备编号

bridge

指向此设备所在桥设备的指针

dev

内部设备结构体

drv\_list

设备列表（按驱动程序）

bridge\_list

设备列表（按桥）

### struct vme\_driver

代表虚拟机驱动程序的结构体

**定义**

struct vme\_driver {

const char \*name;

int (\*match)(struct vme\_dev \*);

int (\*probe)(struct vme\_dev \*);

void (\*remove)(struct vme\_dev \*);

struct device\_driver driver;

struct list\_head devices;

};

**成员**

name

驱动程序名称，应在VME驱动程序中唯一，通常与模块名称相同。

match

用于确定是否应运行探测的回调。

probe

用于设备绑定的回调，在检测到新设备时调用。

remove

回调，在设备移除时调用。

driver

基础通用设备驱动程序结构体。

devices

与此驱动程序关联的VME设备（struct vme\_dev）列表。

### void \*vme\_alloc\_consistent(struct vme\_resource \*resource, size\_t size, dma\_addr\_t \*dma)

分配连续内存

**参数**

struct vme\_resource \*resource

指向VME资源的指针。

size\_t size

所需分配的大小。

dma\_addr\_t \*dma

指向变量以存储分配的物理地址。

**说明**

为驱动程序分配连续的内存块。这用于创建从设备到客户机窗口的缓冲区。

**返回**

成功时为分配的虚拟地址，失败时为NULL

### void vme\_free\_consistent(struct vme\_resource \*resource, size\_t size, void \*vaddr, dma\_addr\_t dma)

释放先前分配的内存

**参数**

struct vme\_resource \*resource

指向VME资源的指针。

size\_t size

要释放的分配的大小。

void \*vaddr

分配的虚拟地址。

dma\_addr\_t dma

分配的物理地址。

**说明**

释放以前分配的连续内存块。

size\_t vme\_get\_size（struct vme\_resource \*resource）

返回VME窗口的大小的辅助函数。

**参数**

struct vme\_resource \*resource

指向VME从站或主站资源的指针。

**说明**

确定所提供的VME窗口的大小。这是一个辅助函数，包装对vme\_master\_get或vme\_slave\_get的调用，取决于交给它的窗口资源类型。

**返回**

成功时窗口的大小，失败时为零。

#### struct vme\_resource \*vme\_slave\_request（struct vme\_dev \*vdev，u32 address，u32 cycle）

请求VME从站窗口资源。

**参数**

struct vme\_dev \*vdev

指向VME设备结构体vme\_dev的指针，分配给驱动程序实例。

u32 address

所需的VME地址空间。

u32 cycle

所需的VME数据传输周期类型。

**说明**

请求使用可设置为所请求的地址空间和数据传输周期的VME窗口资源。

**返回**

成功时为指向VME资源的指针，失败时为NULL。

#### int vme\_slave\_set（struct vme\_resource \*resource，int enabled，unsigned long long vme\_base，unsigned long long size，dma\_addr\_t buf\_base，u32 aspace，u32 cycle）

设置VME从站窗口配置。

**参数**

struct vme\_resource \*resource

指向VME从站资源的指针。

int enabled

应配置窗口的状态。

unsigned long long vme\_base

窗口的基地址。

unsigned long long size

VME窗口的大小。

dma\_addr\_t buf\_base

用于提供VME从站窗口存储的缓冲区的基地址。

u32 aspace

VME窗口的地址空间。

u32 cycle

VME窗口的数据传输周期类型。

**说明**

为提供的VME从站窗口设置配置。

**返回**

成功为零，如果在此设备上不支持操作，则为-EINVAL，如果提供了无效资源或提供了无效属性。也可能返回特定于硬件的错误。

#### int vme\_slave\_get（struct vme\_resource \*resource，int \*enabled，unsigned long long \*vme\_base，unsigned long long \*size，dma\_addr\_t \*buf\_base，u32 \*aspace，u32 \*cycle）

检索VME从站窗口配置。

**参数**

struct vme\_resource \*resource

指向VME从站资源的指针。

int \*enabled

指向用于存储状态的变量的指针。

unsigned long long \*vme\_base

指向存储窗口基地址的变量的指针。

unsigned long long \*size

指向存储窗口大小的变量的指针。

dma\_addr\_t \*buf\_base

指向存储从站缓冲区基地址的变量的指针。

u32 \*aspace

指向存储VME地址空间的变量的指针。

u32 \*cycle

指向存储VME数据传输周期类型的变量的指针。

**说明**

返回所提供的VME从站窗口的配置。

**返回**

成功时为零，如果在此设备上不支持操作，则为-EINVAL或如果提供了无效资源。

#### void vme\_slave\_free（struct vme\_resource \*resource）

释放VME从站窗口

**参数**

struct vme\_resource \*resource

指向VME从站资源的指针。

**说明**

释放所提供的从站资源，以便可以重新分配。

#### struct vme\_resource \*vme\_master\_request（struct vme\_dev \*vdev，u32 address，u32 cycle，u32 dwidth）

请求VME主站窗口资源。

**参数**

struct vme\_dev \*vdev

指向VME设备结构体vme\_dev的指针，分配给驱动程序实例。

u32 address

请求所需的VME地址空间。

u32周期

所需的VME数据传输周期类型。

u32 dwidth

所需的VME数据传输宽度。

**说明**

请求使用可设置请求的地址空间、数据传输周期和宽度的VME窗口资源。

**返回**

成功时返回VME资源指针，失败时返回NULL。

#### int vme\_master\_set（struct vme\_resource \*resource，int enabled，unsigned long long vme\_base，unsigned long long size，u32 aspace，u32周期，u32 dwidth）

设置VME主窗口配置。

**参数**

struct vme\_resource \*resource

指向VME主资源的指针。

int enabled

应配置窗口的状态。

unsigned long long vme\_base

窗口的基地址。

unsigned long long size

VME窗口的大小。

u32 aspace

VME地址空间。

u32周期

用于VME窗口的VME数据传输周期类型。

u32 dwidth

用于VME窗口的VME数据传输宽度。

**说明**

设置提供的VME主窗口的配置。

**返回**

成功时返回零，如果在此设备上不支持操作、提供了无效的资源或提供了无效属性，则返回-EINVAL。还可以返回硬件特定错误。

#### int vme\_master\_get（struct vme\_resource \*resource，int \*enabled，unsigned long long \*vme\_base，unsigned long long \*size，u32 \*aspace，u32 \*周期，u32 \*dwidth）

检索VME主窗口配置。

**参数**

struct vme\_resource \*resource

指向VME主资源的指针。

int \*enabled

指向状态变量的指针。

unsigned long long \*vme\_base

指向窗口基地址的指针。

unsigned long long \*size

指向窗口大小的指针。

u32 \*aspace

指向VME地址空间的指针。

u32 \*cycle

指向VME数据传输周期类型的指针。

u32 \*dwidth

指向VME数据传输宽度的指针。

**说明**

返回提供的VME主窗口的配置。

**返回**

成功时返回零，如果在此设备上不支持操作或提供了无效资源，则返回-EINVAL。

#### ssize\_t vme\_master\_read（struct vme\_resource \*resource，void \*buf，size\_t count，loff\_t offset）

从VME空间读取数据到缓冲区。

**参数**

struct vme\_resource \*resource

指向VME主资源的指针。

void \*buf

指向数据应传输到的缓冲区的指针。

size\_t count

要传输的字节数。

loff\_t offset

开始传输的VME主窗口偏移量。

**说明**

从VME总线上映射到VME主窗口的位置读取count字节数据到buf中。

**返回**

返回读取的字节数；如果resource不是VME主资源或不支持读取操作，则返回-EINVAL。如果提供了无效偏移量，则返回-EFAULT。还可以返回硬件特定错误。

#### ssize\_t vme\_master\_write（struct vme\_resource \*resource，void \*buf，size\_t count，loff\_t offset）

从缓冲区向VME空间写入数据。

**参数**

struct vme\_resource \*resource

指向VME主资源的指针。

void \*buf

指向要传输数据的缓冲区的指针。

size\_t count

要传输的字节数。

loff\_t offset

开始传输的VME主窗口偏移量。

**说明**

从buf中写入count字节数据到VME总线上映射到VME主窗口的位置。

**返回**

返回已写入的字节数；如果resource不是VME主资源或不支持写操作，则返回-EINVAL。如果提供了无效偏移量，则返回-EFAULT。还可以返回硬件特定错误。

#### unsigned int vme\_master\_rmw（struct vme\_resource \*resource，unsigned int mask，unsigned int compare，unsigned int swap，loff\_t offset）

执行读-修改-写周期。

**参数**

struct vme\_resource \*resource

指向VME主资源的指针。

unsigned int mask

在操作中要比较和交换的位。

unsigned int compare

与从偏移量读取的数据进行比较的位。

unsigned int swap

要从偏移量读取的要交换的位。

loff\_t offset

要执行操作的VME主窗口偏移量。

**说明**

在提供的位置执行读-修改-写操作： - 从VME总线读取位置。 - 由掩码选择的位与比较中的匹配的位一旦在交换中被选择，则进行交换。 - 将结果写回VME总线上的位置。

**返回**

成功时返回写入的字节数；如果resource不是VME主资源或不支持RMW操作，则返回-EINVAL。还可以返回硬件特定错误。

#### int vme\_master\_mmap（struct vme\_resource \*resource，struct vm\_area\_struct \*vma）

将VME主窗口的一段区域映射到用户空间。

**参数**

struct vme\_resource \*resource

指向VME主资源的指针。

struct vm\_area\_struct \*vma

指向用户映射定义的指针。

**说明**

将VME主窗口的一段区域内存映射到用户空间。

**返回**

成功时返回零；如果resource不是VME主资源或映射超出了窗口大小，则返回-EINVAL。还可能返回其他通用的mmap错误。

#### void vme\_master\_free（struct vme\_resource \*resource）

释放VME主窗口

**参数**

struct vme\_resource \*resource

指向VME主资源的指针。

**说明**

释放提供的主资源以便重新分配。

#### struct vme\_resource \*vme\_dma\_request（struct vme\_dev \*vdev，u32 route）

请求DMA控制器。

**参数**

struct vme\_dev \*vdev

指向分配给驱动程序实例的VME设备结构 vme\_dev的指针。

u32 route

所需的src / destination组合。

**说明**

请求具有执行所需源/目标组合之间传输能力的VME DMA控制器。

**返回**

成功时VME DMA资源指针，失败时为NULL。

#### struct vme\_dma\_list \*vme\_new\_dma\_list（struct vme\_resource \*resource）

创建新的VME DMA列表。

**参数**

struct vme\_resource \*resource

指向VME DMA资源的指针。

**说明**

创建新的VME DMA列表。一旦不再需要，用户有责任自由使用vme\_dma\_list\_free（）。

**返回**

指向新VME DMA列表的指针;分配失败或无效时为NULL

VME DMA资源。

#### struct vme\_dma\_attr \*vme\_dma\_pattern\_attribute（u32 pattern，u32 type）

创建“Pattern”类型VME DMA列表属性。

**参数**

u32 pattern

用作模式的值

u32 type

要写入的模式类型。

**说明**

为模式生成创建VME DMA列表属性。用户有责任使用vme\_dma\_free\_attribute（）释放使用的属性。

**返回**

指向VME DMA属性的指针;失败时为NULL。

#### struct vme\_dma\_attr \*vme\_dma\_pci\_attribute（dma\_addr\_t address）

创建“PCI”类型VME DMA列表属性。

**参数**

dma\_addr\_t address

DMA传输的PCI基地址。

**说明**

创建指向PCI上的位置的VME DMA列表属性，以进行DMA传输。用户有责任使用vme\_dma\_free\_attribute（）释放已使用的属性。

**返回**

指向VME DMA属性的指针;失败时为NULL。

#### struct vme\_dma\_attr \*vme\_dma\_vme\_attribute（unsigned long long address，u32 aspace，u32 cycle，u32 dwidth）

创建“VME”类型VME DMA列表属性。

**参数**

unsigned long long address

DMA传输的VME基地址。

u32 aspace

用于DMA传输的VME地址空间。

u32 cycle

用于DMA传输的VME总线周期。

u32 dwidth

用于DMA传输的VME数据宽度。

**说明**

创建指向VME总线上的位置的VME DMA列表属性，以进行DMA传输。用户有责任使用vme\_dma\_free\_attribute（）释放已使用的属性。

**返回**

指向VME DMA属性的指针;失败时为NULL。

#### void vme\_dma\_free\_attribute（struct vme\_dma\_attr \*attributes）

释放DMA列表属性。

**参数**

struct vme\_dma\_attr \*attributes

指向DMA列表属性的指针。

**说明**

释放VME DMA列表属性。 VME DMA列表属性可以在vme\_dma\_list\_add（）返回后安全释放。

#### int vme\_dma\_list\_add（struct vme\_dma\_list \*list，struct vme\_dma\_attr \*src，struct vme\_dma\_attr \*dest，size\_t count）

将条目添加到VME DMA列表中。

**参数**

struct vme\_dma\_list \*list

指向VME列表的指针。

struct vme\_dma\_attr \*src

用作源的DMA列表属性的指针。

struct vme\_dma\_attr \*dest

用作目标的DMA列表属性的指针。

size\_t count

要传输的字节数。

**说明**

向提供的VME DMA列表中添加条目。条目要求指向源和目标DMA属性的指针以及计数。

请注意，用于传输的支持源和目的地的属性是硬件相关的。

**返回**

成功时为零，如果在此设备上不支持操作或链接列表已提交执行，则为-EINVAL。硬件特定错误也可能存在。

#### int vme\_dma\_list\_exec（struct vme\_dma\_list \*list）

将VME DMA列表排队以执行。

**参数**

struct vme\_dma\_list \*list

指向VME列表的指针。

**说明**

将提供的VME DMA列表排队以执行。调用将在列表执行后返回。

**返回**

成功时为零，如果在此设备上不支持操作，则为-EINVAL。硬件特定错误也可能存在。

#### int vme\_dma\_list\_free（struct vme\_dma\_list \*list）

释放VME DMA列表。

**参数**

struct vme\_dma\_list \*list

指向VME列表的指针。

**说明**

释放提供的DMA列表及其所有条目。

**返回**

成功时为零，如果VME资源无效，则为-EINVAL，如果资源仍在使用，则为-EBUSY。硬件特定错误也可能存在。

#### int vme\_dma\_free（struct vme\_resource \*resource）

释放VME DMA资源。

**参数**

struct vme\_resource \*resource

指向VME DMA资源的指针。

**说明**

释放提供的DMA资源，以便重新分配。

**返回**

成功时为零，如果VME资源无效，则为-EINVAL，如果资源仍处于活动状态，则为-EBUSY。

#### int vme\_irq\_request（struct vme\_dev \*vdev，int level，int statid，void（\* callback）（int，int，void \*），void \*priv\_data）

请求特定的VME中断。

**参数**

struct vme\_dev \*vdev

指向分配给驱动程序实例的VME设备结构 vme\_dev的指针。

int level

正在请求的中断优先级。

int statid

正在请求的中断向量。

void（\* callback）（int，int，void \*）

当接收到VME中断/向量时调用的回调函数的指针。

void \*priv\_data

将传递给回调函数的通用指针。

**说明**

请求将回调附加为具有提供的级别和statid的VME中断处理程序。

**返回**

成功则返回0，无效的vme设备、优先级或不支持此函数则返回-EINVAL，如果优先级/statid组合已被使用则返回-EBUSY。也有可能出现硬件特定的错误。

#### void vme\_irq\_free(struct vme\_dev \*vdev, int level, int statid)

释放一个VME中断。

**参数**

struct vme\_dev \*vdev

指向分配给驱动程序实例的VME设备结构体 vme\_dev的指针。

int level

正在释放的中断的中断优先级。

int statid

正在释放的中断的中断向量。

**说明**

从VME中断优先级/向量中删除先前附加的回调。

#### int vme\_irq\_generate(struct vme\_dev \*vdev, int level, int statid)

生成VME中断。

**参数**

struct vme\_dev \*vdev

指向分配给驱动程序实例的VME设备结构体 vme\_dev的指针。

int level

断言中断的中断优先级。

int statid

与中断关联的中断向量。

**说明**

生成具有提供的优先级和提供的statid的VME中断。

**返回**

成功则返回0，无效的vme设备、优先级或不支持此函数则返回-EINVAL。也有可能出现硬件特定的错误。

#### struct vme\_resource \*vme\_lm\_request(struct vme\_dev \*vdev)

请求一个VME位置监视器

**参数**

struct vme\_dev \*vdev

指向分配给驱动程序实例的VME设备结构体 vme\_dev的指针。

**说明**

将位置监视器资源分配给驱动程序。位置监视器允许驱动程序监视VME总线上一系列连续的地址。

**返回**

成功则返回指向VME资源的指针，否则返回NULL。

#### int vme\_lm\_count(struct vme\_resource \*resource)

确定受监视的VME地址数量

**参数**

struct vme\_resource \*resource

指向VME位置监视器资源的指针。

**说明**

受监视的连续地址数量取决于硬件。返回位置监视器监视的连续地址数量。

**返回**

地址计数或提供无效位置监视器资源时返回-EINVAL。

#### int vme\_lm\_set(struct vme\_resource \*resource, unsigned long long lm\_base, u32 aspace, u32 cycle)

配置位置监视器

**参数**

struct vme\_resource \*resource

指向VME位置监视器资源的指针。

unsigned long long lm\_base

要监视的基地址。

u32 aspace

要监视的VME地址空间。

u32 cycle

要监视的VME总线周期类型。

**说明**

设置要由位置监视器监视的访问的基地址、地址空间和周期类型。

**返回**

成功则返回0，提供无效的位置监视器资源或不支持此函数则返回-EINVAL。也有可能出现硬件特定的错误。

#### int vme\_lm\_get(struct vme\_resource \*resource, unsigned long long \*lm\_base, u32 \*aspace, u32 \*cycle)

检索位置监视器设置

**参数**

struct vme\_resource \*resource

指向VME位置监视器资源的指针。

unsigned long long \*lm\_base

用于输出所监视的基地址的指针。

u32 \*aspace

用于输出所监视的地址空间的指针。

u32 \*cycle

用于输出所监视的VME总线周期类型的指针。

**说明**

检索由位置监视器监视的访问的基地址、地址空间和周期类型。

**返回**

成功则返回0，提供无效的位置监视器资源或不支持此函数则返回-EINVAL。也有可能出现硬件特定的错误。

#### int vme\_lm\_attach(struct vme\_resource \*resource, int monitor, void (\*callback)(void\*), void \*data)

为位置监视器地址提供回调

**参数**

struct vme\_resource \*resource

指向VME位置监视器资源的指针。

int monitor

应附加回调的偏移量。

void (\*callback)(void \*)

触发时调用的回调函数的指针。

void \*data

将传递给回调函数的通用指针。

**说明**

将回调附加到位置监视器监视的地址的特定偏移量上。提供通用指针以允许数据在调用时传递到回调函数。

**返回**

成功则返回0，提供无效的位置监视器资源或不支持此函数则返回-EINVAL。也有可能出现硬件特定的错误。

#### int vme\_lm\_detach(struct vme\_resource \*resource, int monitor)

删除位置监视器地址回调

**参数**

struct vme\_resource \*resource

指向VME位置监视器资源的指针。

int monitor

要删除回调的偏移量。

**说明**

删除与位置监视器监视的地址的指定偏移量相关联的回调。

**返回**

成功则返回0，提供无效的位置监视器资源或不支持此函数则返回-EINVAL。也有可能出现硬件特定的错误。

#### void vme\_lm\_free(struct vme\_resource \*resource)

释放分配的VME位置监视器

**参数**

struct vme\_resource \*resource

指向VME位置监视器资源的指针。

**说明**

释放VME位置监视器的分配。

警告：该函数目前假定已删除任何已附加到位置监视器上的回调。

**返回**

成功则返回0，提供无效的位置监视器资源则返回-EINVAL。

#### int vme\_slot\_num(struct vme\_dev \*vdev)

检索插槽ID

**参数**

struct vme\_dev \*vdev

指向分配给驱动程序实例的VME设备结构体 vme\_dev的指针。

**说明**

检索与提供的VME设备相关联的插槽ID。

**返回**

成功则返回插槽ID，如果无法确定VME桥，则返回-EINVAL。

或功能不支持。也可能返回硬件特定的错误。

#### int vme\_bus\_num（struct vme\_dev \*vdev）

检索总线编号

**参数**

struct vme\_dev \*vdev

指向VME设备结构vme\_dev的指针，该结构分配给驱动程序实例。

**说明**

检索与提供的VME设备相关联的总线枚举。

**返回**

成功时总线号，如果无法确定VME桥，则为-EINVAL。

#### int vme\_register\_driver（struct vme\_driver \*drv，unsigned int ndevs）

注册VME驱动程序

**参数**

struct vme\_driver \*drv

要注册的VME驱动程序的指针。

unsigned int ndevs

允许枚举的最大设备数。

**说明**

向VME子系统注册VME设备驱动程序。

**返回**

成功时为零，注册失败时为错误值。

#### void vme\_unregister\_driver（struct vme\_driver \*drv）

注销VME驱动程序

**参数**

struct vme\_driver \*drv

要注销的VME驱动程序的指针。

**说明**

从VME子系统中注销VME设备驱动程序。

# Linux 802.11 驱动程序开发者指南

## 介绍

解释Linux内核中的无线802.11网络

版权 2007-2009 Johannes Berg

这些书试图对在Linux中扮演802.11无线网络角色的各个子系统进行说明。由于这些书是为内核开发人员编写的，因此它们尝试记录内核中使用的结构和函数，以及给出更高级别的概述。

读者应熟悉由IEEE在802.11-2007（或可能是更高版本）中发布的802.11标准。对该标准的引用将被称为“802.11-2007 8.1.5”。

## cfg80211子系统

cfg80211是Linux中用于802.11设备的配置API。它桥接用户空间和驱动程序，并提供与802.11相关的一些实用功能。所有现代Linux无线驱动程序都必须直接或间接通过mac80211使用cfg80211，以便它们通过nl80211提供一致的API。为了向后兼容，cfg80211还向用户空间提供无线扩展，但完全隐藏了它们对驱动程序的使用。

此外，cfg80211包含帮助执行规制频谱使用限制的代码。

### 设备注册

为了使驱动程序使用cfg80211，它必须向cfg80211注册硬件设备。这通过下面说明的多个硬件功能结构发生。

每个设备的基本结构是“wiphy”，其中每个实例说明连接到系统的一个物理无线设备。每个这样的wiphy可以有零个、一个或多个与之关联的虚拟接口，需要通过将网络接口的ieee80211\_ptr指针指向一个进一步说明接口无线部分的struct\_wireless\_dev来识别这些接口。通常，此结构嵌入在网络接口的私有数据区域中。驱动程序可以选择允许在飞行中创建或销毁虚拟接口，但如果没有至少一个接口或创建一些接口的能力，无线设备将无用。

每个wiphy结构包含设备功能信息，并且还具有指向驱动程序提供的各种操作的指针。这里的定义和结构详细说明了这些功能。

#### enum ieee80211\_channel\_flags

通道标志

**常量**

IEEE80211\_CHAN\_DISABLED

此通道已禁用。

IEEE80211\_CHAN\_NO\_IR

不进行辐射，这包括发送探测请求或beaconing。

IEEE80211\_CHAN\_RADAR

在此通道上需要雷达检测。

IEEE80211\_CHAN\_NO\_HT40PLUS

不允许在该通道上方使用扩展通道。

IEEE80211\_CHAN\_NO\_HT40MINUS

不允许在该通道下方使用扩展通道。

IEEE80211\_CHAN\_NO\_OFDM

不允许在此通道上使用OFDM。

IEEE80211\_CHAN\_NO\_80MHZ

如果驱动程序支持带宽上的80 MHz，则此标志指示80 MHz通道不能将此通道用作控制通道或任何辅助通道之一。这可能是由于驱动程序或由于监管带宽限制。

IEEE80211\_CHAN\_NO\_160MHZ

如果驱动程序支持带宽上的160 MHz，则此标志指示160 MHz通道不能将此通道用作控制通道或任何辅助通道之一。这可能是由于驱动程序或监管带宽限制。

IEEE80211\_CHAN\_INDOOR\_ONLY

请参见NL80211\_FREQUENCY\_ATTR\_INDOOR\_ONLY

IEEE80211\_CHAN\_IR\_CONCURRENT

请参见NL80211\_FREQUENCY\_ATTR\_IR\_CONCURRENT

IEEE80211\_CHAN\_NO\_20MHZ

20 MHz带宽在此通道上不允许。

IEEE80211\_CHAN\_NO\_10MHZ

在此通道上不允许10 MHz带宽。

**说明**

由规制控制代码设置的通道标志。

#### struct ieee80211\_channel

通道定义

**定义**

struct ieee80211\_channel {

enum nl80211\_band band;

u16 center\_freq;

u16 hw\_value;

u32 flags;

int max\_antenna\_gain;

int max\_power;

int max\_reg\_power;

bool beacon\_found;

u32 orig\_flags;

int orig\_mag, orig\_mpwr;

enum nl80211\_dfs\_state dfs\_state;

unsigned long dfs\_state\_entered;

unsigned int dfs\_cac\_ms;

};

**成员**

band

通道所属的频带。

center\_freq

中心频率（以MHz为单位）

hw\_value

通道的硬件专用值

flags

来自枚举ieee80211\_channel\_flags的通道标志。

max\_antenna\_gain

最大天线增益（以dBi为单位）

max\_power

最大传输功率（以dBm为单位）

max\_reg\_power

最大监管传输功率（以dBm为单位）

beacon\_found

用于规制代码的辅助功能，指示已在此通道上找到信标。使用regulatory\_hint\_found\_beacon（）启用此功能，这仅在5 GHz频段上有用。

orig\_flags

注册时的信道标志，用于支持具有附加限制的设备的监管代码

orig\_mag

内部使用

orig\_mpwr

内部使用

dfs\_state

该信道当前状态。仅当需要在此信道上使用雷达时才相关。

dfs\_state\_entered

DFS状态进入时的时间戳（jiffies）。

dfs\_cac\_ms

DFS CAC时间（毫秒），这对DFS信道有效。

**说明**

该结构说明了与cfg80211一起使用的单个信道。

#### enum ieee80211\_rate\_flags

速率标志

**常量**

IEEE80211\_RATE\_SHORT\_PREAMBLE

硬件可以以此速率发送短前导；仅在2.4 GHz频段和CCK速率中有意义。

IEEE80211\_RATE\_MANDATORY\_A

当802.11a（在5 GHz频带上）使用时，该速率是强制速率；在注册wiphy时由核心代码填充。

IEEE80211\_RATE\_MANDATORY\_B

当使用802.11b（在2.4 GHz频带上）时，该速率是强制速率；在注册wiphy时由核心代码填充。

IEEE80211\_RATE\_MANDATORY\_G

当802.11g（在2.4 GHz频带上）使用时，该速率是强制速率；在注册wiphy时由核心代码填充。

IEEE80211\_RATE\_ERP\_G

这是802.11g模式下的ERP速率。

IEEE80211\_RATE\_SUPPORTS\_5MHZ

速率可以在5 MHz模式下使用

IEEE80211\_RATE\_SUPPORTS\_10MHZ

速率可以在10 MHz模式下使用

**说明**

速率的硬件/规范标志。这些结构化的方式允许在不同的频段/物理模式下使用相同的比特率结构。

#### struct ieee80211\_rate

比特率定义

**定义**

struct ieee80211\_rate {

u32 flags;

u16 bitrate;

u16 hw\_value, hw\_value\_short;

};

**成员**

flags

速率特定标志

bitrate

单位为100 Kbps的比特率

hw\_value

此速率的驱动程序/硬件值

hw\_value\_short

使用短前导时此速率的驱动程序/硬件值

**说明**

该结构说明了802.11 PHY可以操作的比特率。当传递指向此结构的指针时，hw\_value和hw\_value\_short仅供驱动程序使用。

#### struct ieee80211\_sta\_ht\_cap

STA的HT功能

**定义**

struct ieee80211\_sta\_ht\_cap {

u16 cap;

bool ht\_supported;

u8 ampdu\_factor;

u8 ampdu\_density;

struct ieee80211\_mcs\_info mcs;

};

**成员**

cap

HT功能映射，如802.11n规范中所述

ht\_supported

STA是否支持HT

ampdu\_factor

最大A-MPDU长度系数

ampdu\_density

最小A-MPDU间距

mcs

支持的MCS速率

说明

该结构说明了一个STA的802.11n HT功能所需的最基本参数。

#### struct ieee80211\_supported\_band

频段定义

**定义**

struct ieee80211\_supported\_band {

struct ieee80211\_channel \*channels;

struct ieee80211\_rate \*bitrates;

enum nl80211\_band band;

int n\_channels;

int n\_bitrates;

struct ieee80211\_sta\_ht\_cap ht\_cap;

struct ieee80211\_sta\_vht\_cap vht\_cap;

u16 n\_iftype\_data;

const struct ieee80211\_sband\_iftype\_data \*iftype\_data;

};

**成员**

channels

channels

硬件可以在此频段内操作的信道数组。

bitrates

硬件可以在此频段内操作的速率数组。必须排序以获得有效的“支持速率”IE，即首先是CCK速率，然后是OFDM。

band

表示该结构的频段

n\_channels

信道中的信道数

n\_bitrates

速率中的速率数

ht\_cap

此频段中的HT功能

vht\_cap

此频段中的VHT功能

n\_iftype\_data

iftype数据条目数

iftype\_data

接口类型数据条目。请注意，此结构内的types\_mask中的位不能重叠（即，每种类型在iftype\_data的所有实例中只允许出现一次）。

**说明**

该结构说明了一个wiphy能够操作的频段。

#### enum cfg80211\_signal\_type

信号类型

**常量**

CFG80211\_SIGNAL\_TYPE\_NONE

没有可用的信号强度信息

CFG80211\_SIGNAL\_TYPE\_MBM

mBm中的信号强度（100×dBm）

CFG80211\_SIGNAL\_TYPE\_UNSPEC

信号强度，从0到100递增

#### enum wiphy\_params\_flags

set\_wiphy\_params位字段值

**常量**

WIPHY\_PARAM\_RETRY\_SHORT

wiphy->retry\_short已更改

WIPHY\_PARAM\_RETRY\_LONG

wiphy->retry\_long已更改

WIPHY\_PARAM\_FRAG\_THRESHOLD

wiphy->frag\_threshold已更改

WIPHY\_PARAM\_RTS\_THRESHOLD

wiphy->rts\_threshold已更改

WIPHY\_PARAM\_COVERAGE\_CLASS

覆盖范围类别已更改

WIPHY\_PARAM\_DYN\_ACK

已启用dynack

WIPHY\_PARAM\_TXQ\_LIMIT

TXQ数据包限制已更改

WIPHY\_PARAM\_TXQ\_MEMORY\_LIMIT

TXQ内存限制已更改

WIPHY\_PARAM\_TXQ\_QUANTUM

TXQ调度器量子

#### enum wiphy\_flags

wiphy能力标志

**常量**

WIPHY\_FLAG\_NETNS\_OK

如果未设置，则根本不允许更改此wiphy的netns

WIPHY\_FLAG\_PS\_ON\_BY\_DEFAULT

如果设置为true，则默认情况下将启用powersave-此标志将根据内核的默认情况在wiphy\_new()上设置，但如果驱动程序具有覆盖默认值的充分理由，则可以更改该标志

WIPHY\_FLAG\_4ADDR\_AP

即使在VLAN接口上有单个站点时，也支持4addr模式

WIPHY\_FLAG\_4ADDR\_STATION

即使作为站点，也支持4addr模式

WIPHY\_FLAG\_CONTROL\_PORT\_PROTOCOL

该设备支持设置控制端协议以太类型。该设备还考虑到control\_port\_no\_encrypt标志。

WIPHY\_FLAG\_IBSS\_RSN

该设备支持IBSS RSN。

WIPHY\_FLAG\_MESH\_AUTH

该设备支持通过将auth帧路由到用户空间来进行mesh身份验证。请参见NL80211\_MESH\_SETUP\_USERSPACE\_AUTH。

WIPHY\_FLAG\_SUPPORTS\_FW\_ROAM

该设备支持固件中的漫游功能。

WIPHY\_FLAG\_AP\_UAPSD

该设备支持AP上的uapsd。

WIPHY\_FLAG\_SUPPORTS\_TDLS

该设备支持TDLS（802.11z）操作。

WIPHY\_FLAG\_TDLS\_EXTERNAL\_SETUP

该设备不在内部处理TDLS（802.11z）链接设置/发现操作。应该通过NL80211\_CMD\_TDLS\_MGMT命令发送设置，发现和拆除数据包。当未设置此标志时，应使用NL80211\_CMD\_TDLS\_OPER来要求驱动程序/固件执行TDLS操作。

WIPHY\_FLAG\_HAVE\_AP\_SME

设备集成AP SME功能。

WIPHY\_FLAG\_REPORTS\_OBSS

当虚拟接口处于AP模式时，设备将通过调用cfg80211\_report\_obss\_beacon()报告来自其他BSS的信标帧。

WIPHY\_FLAG\_AP\_PROBE\_RESP\_OFFLOAD

作为AP运行时，设备可以在硬件中响应探测请求。

WIPHY\_FLAG\_OFFCHAN\_TX

设备支持直接离线发送数据包（off-channel TX）。

WIPHY\_FLAG\_HAS\_REMAIN\_ON\_CHANNEL

设备支持保留通道（remain-on-channel）操作调用。

WIPHY\_FLAG\_SUPPORTS\_5\_10\_MHZ

设备支持5 MHz和10 MHz频段的无线信道。

WIPHY\_FLAG\_HAS\_CHANNEL\_SWITCH

在beaconing模式下（如AP、IBSS、Mesh等），该设备支持信道切换操作。

WIPHY\_FLAG\_HAS\_STATIC\_WEP

该设备支持连接前安装静态 WEP 密钥。

#### struct wiphy

说明了无线硬件信息。

**定义**

struct wiphy {

u8 perm\_addr[ETH\_ALEN];

u8 addr\_mask[ETH\_ALEN];

struct mac\_address \*addresses;

const struct ieee80211\_txrx\_stypes \*mgmt\_stypes;

const struct ieee80211\_iface\_combination \*iface\_combinations;

int n\_iface\_combinations;

u16 software\_iftypes;

u16 n\_addresses;

u16 interface\_modes;

u16 max\_acl\_mac\_addrs;

u32 flags, regulatory\_flags, features;

u8 ext\_features[DIV\_ROUND\_UP(NUM\_NL80211\_EXT\_FEATURES, 8)];

u32 ap\_sme\_capa;

enum cfg80211\_signal\_type signal\_type;

int bss\_priv\_size;

u8 max\_scan\_ssids;

u8 max\_sched\_scan\_reqs;

u8 max\_sched\_scan\_ssids;

u8 max\_match\_sets;

u16 max\_scan\_ie\_len;

u16 max\_sched\_scan\_ie\_len;

u32 max\_sched\_scan\_plans;

u32 max\_sched\_scan\_plan\_interval;

u32 max\_sched\_scan\_plan\_iterations;

int n\_cipher\_suites;

const u32 \*cipher\_suites;

u8 retry\_short;

u8 retry\_long;

u32 frag\_threshold;

u32 rts\_threshold;

u8 coverage\_class;

char fw\_version[ETHTOOL\_FWVERS\_LEN];

u32 hw\_version;

#ifdef CONFIG\_PM;

const struct wiphy\_wowlan\_support \*wowlan;

struct cfg80211\_wowlan \*wowlan\_config;

#endif;

u16 max\_remain\_on\_channel\_duration;

u8 max\_num\_pmkids;

u32 available\_antennas\_tx;

u32 available\_antennas\_rx;

u32 probe\_resp\_offload;

const u8 \*extended\_capabilities, \*extended\_capabilities\_mask;

u8 extended\_capabilities\_len;

const struct wiphy\_iftype\_ext\_capab \*iftype\_ext\_capab;

unsigned int num\_iftype\_ext\_capab;

const void \*privid;

struct ieee80211\_supported\_band \*bands[NUM\_NL80211\_BANDS];

void (\*reg\_notifier)(struct wiphy \*wiphy, struct regulatory\_request \*request);

const struct ieee80211\_regdomain \_\_rcu \*regd;

struct device dev;

bool registered;

struct dentry \*debugfsdir;

const struct ieee80211\_ht\_cap \*ht\_capa\_mod\_mask;

const struct ieee80211\_vht\_cap \*vht\_capa\_mod\_mask;

struct list\_head wdev\_list;

possible\_net\_t \_net;

#ifdef CONFIG\_CFG80211\_WEXT;

const struct iw\_handler\_def \*wext;

#endif;

const struct wiphy\_coalesce\_support \*coalesce;

const struct wiphy\_vendor\_command \*vendor\_commands;

const struct nl80211\_vendor\_cmd\_info \*vendor\_events;

int n\_vendor\_commands, n\_vendor\_events;

u16 max\_ap\_assoc\_sta;

u8 max\_num\_csa\_counters;

u8 max\_adj\_channel\_rssi\_comp;

u32 bss\_select\_support;

u64 cookie\_counter;

u8 nan\_supported\_bands;

u32 txq\_limit;

u32 txq\_memory\_limit;

u32 txq\_quantum;

char priv[0] ;

};

**成员**

perm\_addr

设备的永久 MAC 地址。

addr\_mask

如果设备通过掩码支持多个 MAC 地址，则将其设置为具有可变位设置为 1 的掩码，例如，如果最后四位是可变的，则将其设置为 00-00-00-00-00-0f。实际的可变位应由添加的接口确定，不匹配掩码的接口将被拒绝启动。

addresses

如果设备有多个地址，请将此指针设置为地址列表（每个地址 6 字节）。默认情况下，第一个地址将用于 perm\_addr。在这种情况下，应该将掩码设置为全零。在这种情况下，假定设备可以处理相同数量的任意 MAC 地址。

mgmt\_stypes

可以通过 nl80211 订阅或传输帧子类型的位掩码，在接口类型索引数组中指向一个数组。

iface\_combinations

有效接口组合数组，不应列出单一接口类型。

n\_iface\_combinations

iface\_combinations 数组中条目数。

software\_iftypes

软件界面类型比特掩码, 这些没有受到任何限制, 因为它们纯粹是在 SW 中管理.

n\_addresses

addresses 中地址数.

interface\_modes

适用于此 wiphy 的界面类型比特掩码, 必须由驱动程序进行设置.

max\_acl\_mac\_addrs

ACL 支持的最大 MAC 地址数.

flags

wiphy 标志, 请参见 enum wiphy\_flags

regulatory\_flags

wiphy 法规标志，请参见 enum ieee80211\_regulatory\_flags

features

广告到 nl80211 的功能，请参阅枚举 nl80211\_feature\_flags。

ext\_features

广告到 nl80211 的扩展功能，请参阅枚举 nl80211\_ext\_feature\_index。

ap\_sme\_capa

AP SME 功能，来自枚举 nl80211\_ap\_sme\_features 中的标志。

signal\_type

在 struct cfg80211\_bss 中报告的信号类型。

bss\_priv\_size

每个 BSS 结构都有分配给它的私有数据，此变量确定其大小.

max\_scan\_ssids

设备可以在任何给定扫描中扫描 SSID 的最大数量.

max\_sched\_scan\_reqs

设备可以同时运行的计划扫描请求的最大数量.

max\_sched\_scan\_ssids

设备可以在任何给定计划扫描中扫描 SSID 的最大数量.

max\_match\_sets

执行计划扫描时设备可处理匹配集合的最大数量，如果不支持过滤，则为 0。

max\_scan\_ie\_len

设备可以添加到进行扫描期间传输的探测请求帧中用户可控 IE 的最大长度，不能包括固定 IE（如支持速率）。

max\_sched\_scan\_ie\_len

与 max\_scan\_ie\_len 相同，但用于计划扫描.

max\_sched\_scan\_plans

设备支持的计划扫描的最大扫描计划数（扫描间隔和迭代次数）。

max\_sched\_scan\_plan\_interval

设备支持的单个扫描计划的最大间隔（以秒为单位）。

max\_sched\_scan\_plan\_iterations

设备支持的单个扫描计划的最大迭代次数。

n\_cipher\_suites

受支持密码套件数量.

cipher\_suites

受支持密码套件.

retry\_short

短帧重试限制 (dot11ShortRetryLimit).

retry\_long

长帧重试限制 (dot11LongRetryLimit).

frag\_threshold

分片阈值 (dot11FragmentationThreshold); -1 = 禁用分片，仅使用 >= 256 的奇数值.

rts\_threshold

RTS 阈值（dot11RTSThreshold）；-1 = 禁用 RTS/CTS

coverage\_class

当前覆盖类别

fw\_version

以太网工具报告的固件版本

hw\_version

用于ethtool报告的硬件版本

wowlan

WoWLAN支持信息

wowlan\_config

当前WoWLAN配置；这通常不应该使用，因为访问它必须是并发的，而应该使用传递给挂起（suspend（））操作的参数。

max\_remain\_on\_channel\_duration

如果已实现，则可以请求Remain-on-channel操作的最大时间。

max\_num\_pmkids

设备支持的PMKID的最大数量

available\_antennas\_tx

可以配置为TX天线的天线位图。除非设置了此项或available\_antennas\_rx，否则将拒绝天线配置命令。

available\_antennas\_rx

可以配置为RX天线的天线位图。除非设置了此项或available\_antennas\_tx，否则将拒绝天线配置命令。

probe\_resp\_offload

用于探测响应卸载的支持协议的位图。请参见枚举nl80211\_probe\_resp\_offload\_support\_attr。仅在wiphy标志WIPHY\_FLAG\_AP\_PROBE\_RESP\_OFFLOAD设置时有效。

extended\_capabilities

驱动程序支持的扩展功能，用户空间可能支持其他功能。这些是802.11扩展功能（“扩展功能元素”），并且与信息元素中的格式相同。请参阅802.11-2012 8.4.2.29以获取定义的字段。如果未为特定接口类型的iftype\_ext\_capab指定功能，则将使用默认扩展功能。

extended\_capabilities\_mask

有效值的掩码

extended\_capabilities\_len

扩展功能的长度

iftype\_ext\_capab

每种接口类型的扩展功能数组

num\_iftype\_ext\_capab

指定单独指定扩展功能的接口类型的数量。

privid

驱动程序可以使用的指针，用于标识任意wiphy是否为其拥有的，例如在全局通知程序中。

bands

有关此设备支持的频段/信道的信息

reg\_notifier

驱动程序的法规通知回调，请注意，如果驱动程序使用wiphy\_apply\_custom\_regulatory（），则reg\_notifier的请求可以传递为NULL

regd

驱动程序的法规域（如果通过regulatory\_hint（）API请求），如果驱动程序选择忽略其他驱动程序引起的未来法规域更改，则可以在reg\_notifier（）上使用它。

dev

（虚拟）此wiphy的设备结构

registered

帮助同步suspend / resume与wiphy取消注册

debugfsdir

用于此wiphy的debugfs目录，在wiphy重命名时将自动重命名

ht\_capa\_mod\_mask

指定哪些ht\_cap值可以被覆盖。如果为null，则不能覆盖任何值。

vht\_capa\_mod\_mask

指定哪些VHT功能可以被覆盖。如果为null，则不能覆盖任何值。

wdev\_list

关联（虚拟）接口的列表；驱动程序不得修改此列表，但可以使用RTNL / RCU保护进行读取。

\_net

该wiphy当前所在的网络命名空间

wext

无线扩展处理程序

coalesce

数据包合并支持信息

vendor\_commands

硬件支持的供应商命令数组

vendor\_events

硬件支持的供应商事件数组

n\_vendor\_commands

供应商命令数

n\_vendor\_events

供应商事件数

max\_ap\_assoc\_sta

AP模式中支持的最大关联站点数（包括P2P GO），或0表示不提供此类限制的广告。驱动程序允许广告可以在某些情况下达到的理论极限，但可能不总是达到。

max\_num\_csa\_counters

使用广播和探测响应消息中的支持的csa\_counters计数器的数量。如果驱动程序希望限制csa计数器的数量，则应设置此值。默认值（0）表示无限。

max\_adj\_channel\_rssi\_comp

发件频道和听到帧的频道之间的最大偏移量，该偏移量仍然有效，并且可以补偿所报告的RSSI。如果驱动程序能够在听到不同频道上的帧时补偿低RSSI，则应将此变量设置为可以补偿的最大偏移量。该值应以MHz为单位设置。

bss\_select\_support

表示驱动程序在.:c:func:connect()回调中支持的BSS选择条件的位掩码。位位置映射到枚举nl80211\_bss\_select\_attr中定义的属性索引。

cookie\_counter

唯一的通用cookie计数器，用于识别对象。

nan\_supported\_bands

设备在NAN模式下支持的频段，为enum nl80211\_band值的位图。例如，对于NL80211\_BAND\_2GHZ，会设置第0位（即BIT（NL80211\_BAND\_2GHZ））。

txq\_limit

内部TX队列帧限制的配置

txq\_memory\_limit

内部TX队列内存限制的配置

txq\_quantum

内部TX队列调度程序量子的配置

priv

驱动程序的私有数据（根据wiphy\_new（）参数的大小）

#### struct wireless\_dev

无线设备状态

**定义**

struct wireless\_dev {

struct wiphy \*wiphy;

enum nl80211\_iftype iftype;

struct list\_head list;

struct net\_device \*netdev;

u32 identifier;

struct list\_head mgmt\_registrations;

spinlock\_t mgmt\_registrations\_lock;

struct mutex mtx;

bool use\_4addr, is\_running;

u8 address[ETH\_ALEN] ;

u8 ssid[IEEE80211\_MAX\_SSID\_LEN];

u8 ssid\_len, mesh\_id\_len, mesh\_id\_up\_len;

struct cfg80211\_conn \*conn;

struct cfg80211\_cached\_keys \*connect\_keys;

enum ieee80211\_bss\_type conn\_bss\_type;

u32 conn\_owner\_nlportid;

struct work\_struct disconnect\_wk;

u8 disconnect\_bssid[ETH\_ALEN];

struct list\_head event\_list;

spinlock\_t event\_lock;

struct cfg80211\_internal\_bss \*current\_bss;

struct cfg80211\_chan\_def preset\_chandef;

struct cfg80211\_chan\_def chandef;

bool ibss\_fixed;

bool ibss\_dfs\_possible;

bool ps;

int ps\_timeout;

int beacon\_interval;

u32 ap\_unexpected\_nlportid;

u32 owner\_nlportid;

bool nl\_owner\_dead;

bool cac\_started;

unsigned long cac\_start\_time;

unsigned int cac\_time\_ms;

#ifdef CONFIG\_CFG80211\_WEXT;

struct {

struct cfg80211\_ibss\_params ibss;

struct cfg80211\_connect\_params connect;

struct cfg80211\_cached\_keys \*keys;

const u8 \*ie;

size\_t ie\_len;

u8 bssid[ETH\_ALEN], prev\_bssid[ETH\_ALEN];

u8 ssid[IEEE80211\_MAX\_SSID\_LEN];

s8 default\_key, default\_mgmt\_key;

bool prev\_bssid\_valid;

} wext;

#endif;

struct cfg80211\_cqm\_config \*cqm\_config;

};

**成员**

wiphy

指向硬件说明的指针

iftype

接口类型

list

（私有的）用于收集接口

netdev

（私有的）用于参考到netdev，可能为 NULL

identifier

（私有的）如果没有netdev，使用nl80211中的标识符识别此无线设备

mgmt\_registrations

管理帧注册的列表

mgmt\_registrations\_lock

列表的锁

mtx

用于锁定此结构中的数据的互斥体，可能由驱动程序使用，某些API函数需要它保持

use\_4addr

表示此接口上使用4addr模式，必须由驱动程序在registering netdev之前在add\_interface上设置（如果支持的话），否则可能可读取，将由cfg80211在change\_interface上进行更新

is\_running

如果这是已启动的非netdev设备（例如P2P Device），则为true。

address

该设备的地址，仅当netdev为NULL时有效

ssid

（私有的）内部配置代码使用

ssid\_len

（私有的）内部配置代码使用

mesh\_id\_len

（私有的）内部配置代码使用

mesh\_id\_up\_len

（私有的）内部配置代码使用

conn

(cfg80211软件SME连接状态机数据的) cfg80211 software SME connection state machine data

connect\_keys

建立连接后要设置的密钥

conn\_bss\_type

连接的BSS类型

conn\_owner\_nlportid

（私有的）连接所有者套接字端口ID

disconnect\_wk

（私有的）自动断开连接工作

disconnect\_bssid

自动断开连接时要使用的BSSID

event\_list

（私有的）用于内部事件处理的列表

event\_lock

事件列表的锁

current\_bss

（私有的）内部配置代码使用

preset\_chandef

（私有的）内部配置代码使用，以便稍后用于AP的跟踪频道

chandef

（私有的）内部配置代码使用，以便跟踪用户设置的频道定义。

ibss\_fixed

（私有的）IBSS采用固定BSSID

ibss\_dfs\_possible

（私有的）IBSS可以更改为DFS频道

ps

启用powersave模式

ps\_timeout

动态powersave超时

beacon\_interval

用于在此设备上传输信标的信标间隔，无效时为0

ap\_unexpected\_nlportid

（私有的）已注册用于意外类3帧(AP模式)的应用程序的netlink端口ID

owner\_nlportid

（私有的）所有者套接字端口ID

nl\_owner\_dead

（私有的）所有者套接字关闭了

cac\_started

如果已启动DFS频道可用性检查，则为true

cac\_start\_time

进入dfs状态时的时间戳（jiffies）。

cac\_time\_ms

CAC时间（以毫秒为单位）

wext

（私有的）内部无线扩展兼容代码使用

cqm\_config

（私有的）nl80211 RSSI监视器状态

**说明**

对于netdev，驱动程序必须分配此结构，该结构使用struct net\_device的ieee80211\_ptr字段（这是有意的，因此它可以与netdev一起分配。）然后不需要注册netdev，因为cfg80211将拦截netdev注册以查看新的无线设备。

对于非netdev使用，驱动程序也必须根据需要分配它作为回调操作的响应，并进行必要的分配，因为在该情况下没有netdev注册，所以不能在回调操作之外分配。

#### struct wiphy \*wiphy\_new(const struct cfg80211\_ops \*ops, int sizeof\_priv)

创建一个新的 wiphy 以与 cfg80211 一起使用

**参数**

const struct cfg80211\_ops \*ops

此设备的配置操作

int sizeof\_priv

用于分配私有区域的大小

**说明**

创建一个新的wiphy并将给定操作与之关联。为私有使用分配sizeof\_priv字节。

**返回**

指向新wiphy的指针。必须将此指针分配给每个netdev的ieee80211\_ptr以进行正确操作。

#### void wiphy\_read\_of\_freq\_limits(struct wiphy \* wiphy)

从设备树中读取频率限制

**参数**

struct wiphy \* wiphy

要获取额外限制的无线设备

**说明**

某些设备可能在DT中指定了额外的限制。这对于通常支持更多频段但由于板设计（例如天线或外部功率放大器）而受限的芯片组可能很有用。

此函数从DT中读取信息并使用它来修改通道（禁用不可用通道）。在具有共享通道数据的驱动程序中使用它通常是一个坏主意，因为DT限制是设备特定的。您应确保仅在对wiphy中的频道进行复制并且可以修改而不影响其他设备的情况下调用它。

由于此函数访问设备节点，因此必须在set\_wiphy\_dev之后调用它。它还修改通道，因此首先必须设置它们。如果正在使用此助手程序，请在wiphy\_register（）之前调用它。

#### int wiphy\_register(struct wiphy \*wiphy)

向cfg80211注册wiphy

**参数**

struct wiphy \*wiphy

要注册的wiphy。

**返回**

非负的wiphy索引或负的错误代码。

#### void wiphy\_unregister(struct wiphy \*wiphy)

从cfg80211中注销wiphy

**参数**

struct wiphy \*wiphy

要注销的wiphy。

**说明**

调用此函数后，不能再使用该priv指针进行更多请求，但是该调用可以休眠以等待正在处理的未处理请求。

#### void wiphy\_free(struct wiphy \*wiphy)

释放wiphy

**参数**

struct wiphy \*wiphy

要释放的wiphy。

#### const char \* wiphy\_name(const struct wiphy \* wiphy)

获取wiphy名称

**参数**

const struct wiphy \* wiphy

要返回名称的wiphy

**返回**

wiphy的名称。

#### struct device \* wiphy\_dev(struct wiphy \* wiphy)

获取wiphy dev指针

**参数**

struct wiphy \* wiphy

要查找其设备结构的wiphy

**返回**

wiphy的dev。

#### void \* wiphy\_priv(struct wiphy \* wiphy)

从wiphy返回priv

**参数**

struct wiphy \* wiphy

要返回其priv指针的wiphy

**返回**

wiphy的priv。

#### struct wiphy \* priv\_to\_wiphy(void \* priv)

返回包含特权的wiphy

**参数**

void \* priv

先前由wiphy\_priv返回的指针

**返回**

特权的wiphy。

#### void set\_wiphy\_dev(struct wiphy \* wiphy, struct device \* dev)

为wiphy设置设备指针

**参数**

struct wiphy \* wiphy

要绑定设备的wiphy

struct device \* dev

要将其指定为父项的设备

#### void \* wdev\_priv(struct wireless\_dev \* wdev)

从wireless\_dev返回wiphy priv

**参数**

struct wireless\_dev \* wdev

要返回其wiphy的特权指针的无线设备

**返回**

wdev的wiphy priv。

#### struct ieee80211\_iface\_limit

特定接口类型的限制

**定义**

struct ieee80211\_iface\_limit {

u16 max;

u16 types;

};

**成员**

max

这些类型的接口的最大数量

types

接口类型（位）

#### struct ieee80211\_iface\_combination

可能的接口组合

**定义**

struct ieee80211\_iface\_combination {

const struct ieee80211\_iface\_limit \*limits;

u32 num\_different\_channels;

u16 max\_interfaces;

u8 n\_limits;

bool beacon\_int\_infra\_match;

u8 radar\_detect\_widths;

u8 radar\_detect\_regions;

u32 beacon\_int\_min\_gcd;

};

**成员**

limits

给定接口类型的限制

num\_different\_channels

可以使用最多这么多不同的通道

max\_interfaces

此组中允许的接口的最大数量

n\_limits

限制数量

beacon\_int\_infra\_match

在此组合中，基础设施和AP类型之间的信标间隔必须匹配。这仅在特定情况下需要。

radar\_detect\_widths

支持雷达检测的通道宽度位图

radar\_detect\_regions

支持雷达检测的区域位图

beacon\_int\_min\_gcd

此接口组合支持不同的信标间隔。

= 0

不同接口的所有信标间隔必须相同。

> 0

此组合部分的接口的任何信标间隔以及来自该组合中信标接口的所有信标间隔的GCD必须大于或等于此值。

**说明**

使用此结构，驱动程序可以说明其支持的并发接口组合。

**示例**

1）允许#STA <= 1，#AP <= 1，匹配BI，通道= 1，总计2个:

struct ieee80211\_iface\_limit limits1[] = {

{ .max = 1, .types = BIT(NL80211\_IFTYPE\_STATION), },

{ .max = 1, .types = BIT(NL80211\_IFTYPE\_AP}, },

};

struct ieee80211\_iface\_combination combination1 = {

.limits = limits1,

.n\_limits = ARRAY\_SIZE(limits1),

.max\_interfaces = 2,

.beacon\_int\_infra\_match = true,

};

2）允许#{AP，P2P-GO} <= 8，通道= 1，总共8个:

struct ieee80211\_iface\_limit limits2[] = {

{ .max = 8, .types = BIT(NL80211\_IFTYPE\_AP) |

BIT(NL80211\_IFTYPE\_P2P\_GO), },

};

struct ieee80211\_iface\_combination combination2 = {

.limits = limits2,

.n\_limits = ARRAY\_SIZE(limits2),

.max\_interfaces = 8,

.num\_different\_channels = 1,

};

3）允许#STA <= 1，#{P2P-client,P2P-GO} <= 3在两个通道上，总共4个。这可以允许基础设施连接和三个P2P连接。

struct ieee80211\_iface\_limit limits3[] = {

{ .max = 1, .types = BIT(NL80211\_IFTYPE\_STATION), },

{ .max = 3, .types = BIT(NL80211\_IFTYPE\_P2P\_GO) |

BIT(NL80211\_IFTYPE\_P2P\_CLIENT),},

};

struct ieee80211\_iface\_combination combination3 = {

.limits = limits3,

.n\_limits = ARRAY\_SIZE(limits3),

.max\_interfaces = 4,

.num\_different\_channels = 2,

};

#### int cfg80211\_check\_combinations(struct [wiphy](https://www.kernel.org/doc/html/latest/driver-api/80211/cfg80211.html" \l "c.cfg80211_check_combinations" \o "wiphy) \*wiphy, struct iface\_combination\_params \*params)

检查接口组合

**参数**

struct wiphy \*wiphy

the wiphy

struct iface\_combination\_params \*params

接口组合参数

**说明**

驱动程序可以通过调用此函数来检查接口和其类型的组合是否符合接口组合的要求。

### 操作和配置

每个无线设备和每个虚拟接口都提供一组配置操作和其他由用户空间调用的操作。每个操作都在操作结构中说明，这些操作使用的参数单独说明。

此外，一些操作是异步的，并期望驱动程序通过一些函数调用获取状态信息。

扫描和带BSS列表处理的相关功能在另一章节中说明。

#### struct cfg80211\_ops

无线配置的后端说明

**定义**

struct cfg80211\_ops {

int (\*suspend)(struct wiphy \*wiphy, struct cfg80211\_wowlan \*wow);

int (\*resume)(struct wiphy \*wiphy);

void (\*set\_wakeup)(struct wiphy \*wiphy, bool enabled);

struct wireless\_dev \* (\*add\_virtual\_intf)(struct wiphy \*wiphy,const char \*name,unsigned char name\_assign\_type,enum nl80211\_iftype type, struct vif\_params \*params);

int (\*del\_virtual\_intf)(struct wiphy \*wiphy, struct wireless\_dev \*wdev);

int (\*change\_virtual\_intf)(struct wiphy \*wiphy,struct net\_device \*dev,enum nl80211\_iftype type, struct vif\_params \*params);

int (\*add\_key)(struct wiphy \*wiphy, struct net\_device \*netdev,u8 key\_index, bool pairwise, const u8 \*mac\_addr, struct key\_params \*params);

int (\*get\_key)(struct wiphy \*wiphy, struct net\_device \*netdev,u8 key\_index, bool pairwise, const u8 \*mac\_addr,void \*cookie, void (\*callback)(void \*cookie, struct key\_params\*));

int (\*del\_key)(struct wiphy \*wiphy, struct net\_device \*netdev, u8 key\_index, bool pairwise, const u8 \*mac\_addr);

int (\*set\_default\_key)(struct wiphy \*wiphy,struct net\_device \*netdev, u8 key\_index, bool unicast, bool multicast);

int (\*set\_default\_mgmt\_key)(struct wiphy \*wiphy,struct net\_device \*netdev, u8 key\_index);

int (\*start\_ap)(struct wiphy \*wiphy, struct net\_device \*dev, struct cfg80211\_ap\_settings \*settings);

int (\*change\_beacon)(struct wiphy \*wiphy, struct net\_device \*dev, struct cfg80211\_beacon\_data \*info);

int (\*stop\_ap)(struct wiphy \*wiphy, struct net\_device \*dev);

int (\*add\_station)(struct wiphy \*wiphy, struct net\_device \*dev,const u8 \*mac, struct station\_参数 \*params);

int (\*del\_station)(struct wiphy \*wiphy, struct net\_device \*dev, struct station\_del\_参数 \*params);

int (\*change\_station)(struct wiphy \*wiphy, struct net\_device \*dev,const u8 \*mac, struct station\_参数 \*params);

int (\*get\_station)(struct wiphy \*wiphy, struct net\_device \*dev, const u8 \*mac, struct station\_info \*sinfo);

int (\*dump\_station)(struct wiphy \*wiphy, struct net\_device \*dev, int idx, u8 \*mac, struct station\_info \*sinfo);

int (\*add\_mpath)(struct wiphy \*wiphy, struct net\_device \*dev, const u8 \*dst, const u8 \*next\_hop);

int (\*del\_mpath)(struct wiphy \*wiphy, struct net\_device \*dev, const u8 \*dst);

int (\*change\_mpath)(struct wiphy \*wiphy, struct net\_device \*dev, const u8 \*dst, const u8 \*next\_hop);

int (\*get\_mpath)(struct wiphy \*wiphy, struct net\_device \*dev, u8 \*dst, u8 \*next\_hop, struct mpath\_info \*pinfo);

int (\*dump\_mpath)(struct wiphy \*wiphy, struct net\_device \*dev,int idx, u8 \*dst, u8 \*next\_hop, struct mpath\_info \*pinfo);

int (\*get\_mpp)(struct wiphy \*wiphy, struct net\_device \*dev, u8 \*dst, u8 \*mpp, struct mpath\_info \*pinfo);

int (\*dump\_mpp)(struct wiphy \*wiphy, struct net\_device \*dev,int idx, u8 \*dst, u8 \*mpp, struct mpath\_info \*pinfo);

int (\*get\_mesh\_config)(struct wiphy \*wiphy,struct net\_device \*dev, struct mesh\_config \*conf);

int (\*update\_mesh\_config)(struct wiphy \*wiphy,struct net\_device \*dev, u32 mask, const struct mesh\_config \*nconf);

int (\*join\_mesh)(struct wiphy \*wiphy, struct net\_device \*dev,const struct mesh\_config \*conf, const struct mesh\_setup \*setup);

int (\*leave\_mesh)(struct wiphy \*wiphy, struct net\_device \*dev);

int (\*join\_ocb)(struct wiphy \*wiphy, struct net\_device \*dev, struct ocb\_setup \*setup);

int (\*leave\_ocb)(struct wiphy \*wiphy, struct net\_device \*dev);

int (\*change\_bss)(struct wiphy \*wiphy, struct net\_device \*dev, struct bss\_参数 \*params);

int (\*set\_txq\_params)(struct wiphy \*wiphy, struct net\_device \*dev, struct ieee80211\_txq\_params \*params);

int (\*libertas\_set\_mesh\_channel)(struct wiphy \*wiphy,struct net\_device \*dev, struct ieee80211\_channel \*chan);

int (\*set\_monitor\_channel)(struct wiphy \*wiphy, struct cfg80211\_chan\_def \*chandef);

int (\*scan)(struct wiphy \*wiphy, struct cfg80211\_scan\_request \*request);

void (\*abort\_scan)(struct wiphy \*wiphy, struct wireless\_dev \*wdev);

int (\*auth)(struct wiphy \*wiphy, struct net\_device \*dev, struct cfg80211\_auth\_request \*req);

int (\*assoc)(struct wiphy \*wiphy, struct net\_device \*dev, struct cfg80211\_assoc\_request \*req);

int (\*deauth)(struct wiphy \*wiphy, struct net\_device \*dev, struct cfg80211\_deauth\_request \*req);

int (\*disassoc)(struct wiphy \*wiphy, struct net\_device \*dev, struct cfg80211\_disassoc\_request \*req);

int (\*connect)(struct wiphy \*wiphy, struct net\_device \*dev, struct cfg80211\_connect\_params \*sme);

int (\*update\_connect\_params)(struct wiphy \*wiphy,struct net\_device \*dev,struct cfg80211\_connect\_params \*sme, u32 changed);

int (\*disconnect)(struct wiphy \*wiphy, struct net\_device \*dev, u16 reason\_code);

int (\*join\_ibss)(struct wiphy \*wiphy, struct net\_device \*dev, struct cfg80211\_ibss\_params \*params);

int (\*leave\_ibss)(struct wiphy \*wiphy, struct net\_device \*dev);

int (\*set\_mcast\_rate)(struct wiphy \*wiphy, struct net\_device \*dev, int rate[NUM\_NL80211\_BANDS]);

int (\*set\_wiphy\_params)(struct wiphy \*wiphy, u32 changed);

int (\*set\_tx\_power)(struct wiphy \*wiphy, struct wireless\_dev \*wdev, enum nl80211\_tx\_power\_setting type, int mbm);

int (\*get\_tx\_power)(struct wiphy \*wiphy, struct wireless\_dev \*wdev, int \*dbm);

int (\*set\_wds\_peer)(struct wiphy \*wiphy, struct net\_device \*dev, const u8 \*addr);

void (\*rfkill\_poll)(struct wiphy \*wiphy);

#ifdef CONFIG\_NL80211\_TESTMODE;

int (\*testmode\_cmd)(struct wiphy \*wiphy, struct wireless\_dev \*wdev, void \*data, int len);

int (\*testmode\_dump)(struct wiphy \*wiphy, struct sk\_buff \*skb,struct netlink\_callback \*cb, void \*data, int len);

#endif;

int (\*set\_bitrate\_mask)(struct wiphy \*wiphy,struct net\_device \*dev,const u8 \*peer, const struct cfg80211\_bitrate\_mask \*mask);

int (\*dump\_survey)(struct wiphy \*wiphy, struct net\_device \*netdev, int idx, struct survey\_info \*info);

int (\*set\_pmksa)(struct wiphy \*wiphy, struct net\_device \*netdev, struct cfg80211\_pmksa \*pmksa);

int (\*del\_pmksa)(struct wiphy \*wiphy, struct net\_device \*netdev, struct cfg80211\_pmksa \*pmksa);

int (\*flush\_pmksa)(struct wiphy \*wiphy, struct net\_device \*netdev);

int (\*remain\_on\_channel)(struct wiphy \*wiphy,struct wireless\_dev \*wdev,struct ieee80211\_channel \*chan,unsigned int duration, u64 \*cookie);

int (\*cancel\_remain\_on\_channel)(struct wiphy \*wiphy,struct wireless\_dev \*wdev, u64 cookie);

int (\*mgmt\_tx)(struct wiphy \*wiphy, struct wireless\_dev \*wdev,struct cfg80211\_mgmt\_tx\_params \*params, u64 \*cookie);

int (\*mgmt\_tx\_cancel\_wait)(struct wiphy \*wiphy,struct wireless\_dev \*wdev, u64 cookie);

int (\*set\_power\_mgmt)(struct wiphy \*wiphy, struct net\_device \*dev, bool enabled, int timeout);

int (\*set\_cqm\_rssi\_config)(struct wiphy \*wiphy,struct net\_device \*dev, s32 rssi\_thold, u32 rssi\_hyst);

int (\*set\_cqm\_rssi\_range\_config)(struct wiphy \*wiphy,struct net\_device \*dev, s32 rssi\_low, s32 rssi\_high);

int (\*set\_cqm\_txe\_config)(struct wiphy \*wiphy,struct net\_device \*dev, u32 rate, u32 pkts, u32 intvl);

void (\*mgmt\_frame\_register)(struct wiphy \*wiphy,struct wireless\_dev \*wdev, u16 frame\_type, bool reg);

int (\*set\_antenna)(struct wiphy \*wiphy, u32 tx\_ant, u32 rx\_ant);

int (\*get\_antenna)(struct wiphy \*wiphy, u32 \*tx\_ant, u32 \*rx\_ant);

int (\*sched\_scan\_start)(struct wiphy \*wiphy,struct net\_device \*dev, struct cfg80211\_sched\_scan\_request \*request);

int (\*sched\_scan\_stop)(struct wiphy \*wiphy, struct net\_device \*dev, u64 reqid);

int (\*set\_rekey\_data)(struct wiphy \*wiphy, struct net\_device \*dev, struct cfg80211\_gtk\_rekey\_data \*data);

int (\*tdls\_mgmt)(struct wiphy \*wiphy, struct net\_device \*dev,const u8 \*peer, u8 action\_code, u8 dialog\_token,u16 status\_code, u32 peer\_capability, bool initiator, const u8 \*buf, size\_t len);

int (\*tdls\_oper)(struct wiphy \*wiphy, struct net\_device \*dev, const u8 \*peer, enum nl80211\_tdls\_operation oper);

int (\*probe\_client)(struct wiphy \*wiphy, struct net\_device \*dev, const u8 \*peer, u64 \*cookie);

int (\*set\_noack\_map)(struct wiphy \*wiphy,struct net\_device \*dev, u16 noack\_map);

int (\*get\_channel)(struct wiphy \*wiphy,struct wireless\_dev \*wdev, struct cfg80211\_chan\_def \*chandef);

int (\*start\_p2p\_device)(struct wiphy \*wiphy, struct wireless\_dev \*wdev);

void (\*stop\_p2p\_device)(struct wiphy \*wiphy, struct wireless\_dev \*wdev);

int (\*set\_mac\_acl)(struct wiphy \*wiphy, struct net\_device \*dev, const struct cfg80211\_acl\_data \*params);

int (\*start\_radar\_detection)(struct wiphy \*wiphy,struct net\_device \*dev,struct cfg80211\_chan\_def \*chandef, u32 cac\_time\_ms);

int (\*update\_ft\_ies)(struct wiphy \*wiphy, struct net\_device \*dev, struct cfg80211\_update\_ft\_ies\_params \*ftie);

int (\*crit\_proto\_start)(struct wiphy \*wiphy,struct wireless\_dev \*wdev,enum nl80211\_crit\_proto\_id protocol, u16 duration);

void (\*crit\_proto\_stop)(struct wiphy \*wiphy, struct wireless\_dev \*wdev);

int (\*set\_coalesce)(struct wiphy \*wiphy, struct cfg80211\_coalesce \*coalesce);

int (\*channel\_switch)(struct wiphy \*wiphy,struct net\_device \*dev, struct cfg80211\_csa\_settings \*params);

int (\*set\_qos\_map)(struct wiphy \*wiphy,struct net\_device \*dev, struct cfg80211\_qos\_map \*qos\_map);

int (\*set\_ap\_chanwidth)(struct wiphy \*wiphy, struct net\_device \*dev, struct cfg80211\_chan\_def \*chandef);

int (\*add\_tx\_ts)(struct wiphy \*wiphy, struct net\_device \*dev,u8 tsid, const u8 \*peer, u8 user\_prio, u16 admitted\_time);

int (\*del\_tx\_ts)(struct wiphy \*wiphy, struct net\_device \*dev, u8 tsid, const u8 \*peer);

int (\*tdls\_channel\_switch)(struct wiphy \*wiphy,struct net\_device \*dev,const u8 \*addr, u8 oper\_class, struct cfg80211\_chan\_def \*chandef);

void (\*tdls\_cancel\_channel\_switch)(struct wiphy \*wiphy,struct net\_device \*dev, const u8 \*addr);

int (\*start\_nan)(struct wiphy \*wiphy, struct wireless\_dev \*wdev, struct cfg80211\_nan\_conf \*conf);

void (\*stop\_nan)(struct wiphy \*wiphy, struct wireless\_dev \*wdev);

int (\*add\_nan\_func)(struct wiphy \*wiphy, struct wireless\_dev \*wdev, struct cfg80211\_nan\_func \*nan\_func);

void (\*del\_nan\_func)(struct wiphy \*wiphy, struct wireless\_dev \*wdev, u64 cookie);

int (\*nan\_change\_conf)(struct wiphy \*wiphy,struct wireless\_dev \*wdev,struct cfg80211\_nan\_conf \*conf, u32 changes);

int (\*set\_multicast\_to\_unicast)(struct wiphy \*wiphy,struct net\_device \*dev, const bool enabled);

int (\*get\_txq\_stats)(struct wiphy \*wiphy,struct wireless\_dev \*wdev, struct cfg80211\_txq\_stats \*txqstats);

int (\*set\_pmk)(struct wiphy \*wiphy, struct net\_device \*dev, const struct cfg80211\_pmk\_conf \*conf);

int (\*del\_pmk)(struct wiphy \*wiphy, struct net\_device \*dev, const u8 \*aa);

int (\*external\_auth)(struct wiphy \*wiphy, struct net\_device \*dev, struct cfg80211\_external\_auth\_params \*params);

int (\*tx\_control\_port)(struct wiphy \*wiphy,struct net\_device \*dev,const u8 \*buf, size\_t len,const u8 \*dest, const \_\_be16 proto, const bool noencrypt);

};

**成员**

suspend

需要挂起 wiphy 设备。变量wow将为NULL或包含为设备配置的启用的 Wake-on-Wireless 触发器

resume

需要恢复 wiphy 设备

set\_wakeup

当启用/禁用 WoWLAN 时调用此回调函数，使用 device\_set\_wakeup\_enable() 来启用/禁用从设备的唤醒。

add\_virtual\_intf

使用给定名称创建新虚拟接口，必须设置结构体 wireless\_dev 的 iftype。注意：您必须在 wiphy 的网络名称空间中创建新的 netdev！返回 struct wireless\_dev 或 ERR\_PTR。对于 P2P 设备 wdevs，驱动程序还必须在 wdev 中设置地址成员。

del\_virtual\_intf

删除虚拟接口

change\_virtual\_intf

更改虚拟接口的类型/配置，请一直更新结构体 wireless\_dev 的 iftype。

add\_key

使用给定参数添加密钥。添加组密钥时，mac\_addr 将为 NULL。

get\_key

获取给定参数密钥的信息。要请求组密钥的信息时，mac\_addr 将为NULL。回调函数给出的所有指针在返回后不需要有效。如果不可能检索密钥，则此函数应返回错误，如果不存在，则返回-ENOENT。

del\_key

删除给定 mac\_addr（为组密钥时为 NULL）和 key\_index 的密钥，如果密钥不存在，则返回-ENOENT。

set\_default\_key

在接口上设置默认密钥

set\_default\_mgmt\_key

在接口上设置默认管理帧密钥

start\_ap

启动由参数定义的 AP 模式。

change\_beacon

更改接入点模式接口的信标参数。如果没有启动 AP 模式，则应拒绝呼叫。

stop\_ap

停止作为 AP，包括停止发出信标。

add\_station

添加新站点。

del\_station

删除一个站点

change\_station

修改给定站点。请注意，cfg80211 中没有对标志更改进行验证，特别是 auth/assoc/authorized 标志可能以无效的组合形式传递到驱动程序 - 请确保检查它们，也与现有状态进行比较！驱动程序必须调用 cfg80211\_check\_station\_change() 来验证信息。

get\_station

获取由 mac 标识的站点的信息

dump\_station

转储站点回调 - 在索引 idx 上恢复转储

add\_mpath

添加一个固定的网格路径

del\_mpath

删除给定的网格路径

change\_mpath

更改给定的网格路径

get\_mpath

获取给定参数的网格路径

dump\_mpath

转存网格路径回调 - 在索引idx处恢复转存

get\_mpp

获取给定参数的网格代理路径

dump\_mpp

转存网格代理路径回调 - 在索引idx处恢复转存

get\_mesh\_config

获取当前网格配置

update\_mesh\_config

在运行网格上更新网格参数。掩码是一个位字段，告诉我们哪些参数要设置，哪些要保留。

join\_mesh

使用指定的参数加入网格网络（带有无线设备互斥锁调用）

leave\_mesh

离开当前网格网络（带有无线设备互斥锁调用）

join\_ocb

使用指定的参数加入OCB网络（带有无线设备互斥锁调用）

leave\_ocb

离开当前OCB网络（带有无线设备互斥锁调用）

change\_bss

修改给定BSS的参数。

set\_txq\_params

设置TX队列参数

libertas\_set\_mesh\_channel

仅用于自由城的向后兼容性，因为它没有实现join\_mesh，需要设置频道以加入网格。

set\_monitor\_channel

为设备设置监视模式频道。如果其他接口处于活动状态，则此回调应拒绝配置。如果没有接口处于活动状态，或者设备处于关闭状态，则应将通道存储以供监视接口变为活动时使用。

scan

请求进行扫描。如果返回零，则将扫描请求传递给驱动程序，并且将有效直到传递给cfg80211\_scan\_done()。要获取扫描结果，请调用cfg80211\_inform\_bss()；您也可以在扫描/scan\_done括号之外调用此函数。

abort\_scan

告诉驱动程序中止正在进行的扫描。驱动程序应通过cfg80211\_scan\_done()指示扫描的状态。

auth

请求与指定的对等体进行身份验证（带有无线设备互斥锁调用）

assoc

请求与指定的对等体（重新）关联（带有无线设备互斥锁调用）

deauth

请求从指定的对等体进行去认证（带有无线设备互斥锁调用）

disassoc

请求从指定对等体进行取消关联（带有无线设备互斥锁调用）

connect

以指定的参数连接ESS。连接后，使用状态码WLAN\_STATUS\_SUCCESS调用cfg80211\_connect\_result()/cfg80211\_connect\_bss()。如果连接因某种原因而失败，则使用来自AP的状态代码调用cfg80211\_connect\_result()/cfg80211\_connect\_bss()或cfg80211\_connect\_timeout()如果未收到状态代码框。当驱动程序与其他BSS匹配连接参数时，允许驱动程序漫游到ESS中的其他BSS。在驱动程序发起的漫游和新的connect()调用发起的漫游案例中，在cfg80211\_roamed()中调用漫游的结果（带有无线设备互斥锁调用）。

update\_connect\_params

在连接到BSS时更新连接参数。更新后的参数可以由驱动程序/固件用于后续BSS选择（漫游）决策并形成Authentication/（Re）协会请求帧。此调用不请求立即与当前BSS进行取消关联或重新关联，即仅在后续（重新）关联时受到影响。 枚举cfg80211\_connect\_params\_changed中定义了changed中的位。（带有无线设备互斥锁调用）

disconnect

断开与BSS/ESS的连接或停止连接尝试（如果正在进行连接）。完成后，如果已经建立连接，则调用cfg80211\_disconnected()，否则调用cfg80211\_connect\_timeout()。（带有无线设备互斥锁调用）

join\_ibss

加入指定的IBSS（如果需要则创建）。完成后，调用cfg80211\_ibss\_joined()，当由于合并而更改BSSID时也调用该函数。（带有无线设备互斥锁调用）

leave\_ibss

离开IBSS。（带有无线设备互斥锁调用）

set\_mcast\_rate

设置指定的多播速率（仅在vif处于ADHOC或MESH模式时）

set\_wiphy\_params

通知wiphy参数已更改；changed位字段（参见枚举wiphy\_params\_flags）说明哪些值已更改。实际参数值可在struct wiphy中获得。如果返回错误，则不应更改任何值。

set\_tx\_power

根据参数设置传输功率，传递的功率单位为mBm，要获取dBm，请使用MBM\_TO\_DBM()。如果没有广告的nl80211功能标志，wdev可以为NULL，始终为NULL，除非驱动程序支持每个vif TX功率。

get\_tx\_power

将当前TX功率存储到dbm变量中；如果成功，则返回0

set\_wds\_peer

设置 WDS 接口的 WDS 对等端点

rfkill\_poll

轮询硬件 rfkill 线路，使用 cfg80211 报告函数来调节 rfkill 硬件状态

testmode\_cmd

运行测试模式命令；wdev 可能为 NULL

testmode\_dump

实现一个测试模式转储。该函数可以使用 cb->args[2] 和上面的参数，但 0 和 1 必须不被触及。此外，返回除 -ENOBUFS 和 -ENOENT 之外的错误代码将终止 转储并返回到用户空间带有一个错误，所以小心控制。如果有任何数据从用户空间传入，那么数据/长度参数将存在并指向包含在 NL80211\_ATTR\_TESTDATA 中的数据。

set\_bitrate\_mask

设置比特率掩码配置

dump\_survey

获取站点调查信息。

set\_pmksa

为 BSSID 缓存 PMKID。这对于运行能够生成（重新）关联 RSN IE 固件的 fullmac 设备非常有用。它允许更快地在 WPA2 BSSIDs 之间漫游。

del\_pmksa

删除缓存的 PMKID。

flush\_pmksa

刷新所有缓存的 PMKIDs。

remain\_on\_channel

请求驱动程序在指定通道上保持唤醒状态，以完成离线通道操作（例如，公共操作帧交换）指定时长。当驱动程序已准备好请求的通道时，它必须通过调用 cfg80211\_ready\_on\_channel() 发出事件通知。

cancel\_remain\_on\_channel

取消正在进行的保留在频道上的操作。这允许在根据持续时间值超时之前终止操作。

mgmt\_tx

传输管理帧。

mgmt\_tx\_cancel\_wait

取消在另一个频道上传输管理帧的等待时间

set\_power\_mgmt

配置 WLAN 电源管理。-1 的超时值允许驱动程序调整动态 ps 超时值。

set\_cqm\_rssi\_config

配置连接质量监控 RSSI 阈值。配置后，驱动程序应该（很快）发送一个事件，指示当前级别是否高于/低于已配置的阈值。当更改配置（而没有首先禁用它）时，可能需要一些小心处理。

set\_cqm\_rssi\_range\_config

在 connection quality monitor 中配置两个 RSSI 阈值。仅在信号电平被发现在两个值之外时才发送事件。如果实现了此方法，则驱动程序应设置 NL80211\_EXT\_FEATURE\_CQM\_RSSI\_LIST。如果提供了它，那么提供 set\_cqm\_rssi\_config 就没有意义了。

set\_cqm\_txe\_config

配置连接质量监控 TX 错误阈值。

mgmt\_frame\_register

通知驱动程序注册了管理帧类型。回调程序允许睡眠。

set\_antenna

在设备上设置天线配置（tx\_ant，rx\_ant）。参数是用于 TX/RX 的允许天线的位图。驱动程序可以通过返回 -EINVAL 拒绝它无法支持的 TX/RX 掩码组合（也请参阅 nl80211.h NL80211\_ATTR\_WIPHY\_ANTENNA\_TX）。

get\_antenna

从设备中获取当前的天线配置（tx\_ant，rx\_ant）。

sched\_scan\_start

告诉驱动程序开始定期扫描。

sched\_scan\_stop

告诉驱动程序停止正在进行的定期扫描，并提供请求 ID。此调用必须停止定期扫描，并准备好在返回之前再次启动，即随后可以立即调用 sched\_scan\_start，并且在该情况下不应失败。驱动程序不应为请求的停止调用 cfg80211\_sched\_scan\_stopped()（当此方法返回 0 时）。

set\_rekey\_data

将 GTK 重键数据提供给驱动程序

tdls\_mgmt

传输 TDLS 管理帧。

tdls\_oper

执行高级 TDLS 操作（例如 TDLS 链路设置）。

probe\_client

探测一个关联的客户端，必须返回后期通过 cfg80211\_probe\_status() 传递的 Cookie。

set\_noack\_map

为 TID 设置 NoAck Map。

get\_channel

获取虚拟接口的当前操作通道。对于监控接口，除非有单个当前监控通道，否则应返回 NULL。

start\_p2p\_device

启动给定的 P2P 设备。

stop\_p2p\_device

停止给定的 P2P 设备。

set\_mac\_acl

在 AP 和 P2P GO 模式下设置 MAC 地址控制列表。参数包括 ACL 策略、站点的 MAC 地址数组和 MAC 地址的数量。如果驱动程序已经具有已在驱动程序中存在的列表，则此新列表替换现有列表。当 MAC 地址条目数传递为 0 时，驱动程序必须清除其 ACL。广告支持基于 MAC 的 ACL 的驱动程序必须实现此回调。

start\_radar\_detection

在驱动程序中开始雷达检测。

update\_ft\_ies

为驱动程序提供更新的快速 BSS 过渡信息。如果 SME 在驱动程序/固件中，则可以在构建身份验证和重新关联请求帧时使用此信息。

crit\_proto\_start

表示需要给定时间（毫秒）内更多的链接可靠性的关键协议。提供协议以便驱动程序可以采取最适当的行动。

crit\_proto\_stop

表示关键协议不再需要更高的链接可靠性。此操作不会失败。

set\_coalesce

设置合并参数。

channel\_switch

启动 带 CSA 的 Channel-Switch 过程。驱动程序负责验证是否可以切换。由于这本质上是棘手的，驱动程序可能决定稍后断开一个接口，并使用cfg80211\_stop\_iface()。这并不意味着驱动程序可以接受任何内容。它应该尽力验证请求，并尽快拒绝它们。

set\_qos\_map

设置QoS映射信息给驱动程序。

set\_ap\_chanwidth

为给定接口设置AP（包括P2P GO）模式的通道宽度。这可用于BSS生存期内动态HT 20/40 MHz通道宽度更改。

add\_tx\_ts

验证（如果admitted\_time为0）或为具有给定参数的设备添加TX TS；用户空间已处理动作帧交换，因此只需修改TX路径以考虑TS。如果承认时间为0，则只需验证参数以确保可以创建会话；对于这个，总是返回成功是有效的，但可能会导致低效行为（与对等方握手，稍后添加被拒绝时立即拆除）。

del\_tx\_ts

删除现有的TX TS。

tdls\_channel\_switch

与TDLS对等体开始切换通道。驱动程序负责持续发起通道切换操作并返回到基本通道以与AP进行通讯。

tdls\_cancel\_channel\_switch

停止与TDLS对等体的通道切换。调用完成时，两个对等体都必须在基本通道上。

start\_nan

启动NAN界面。

stop\_nan

停止NAN接口。

add\_nan\_func

添加NAN函数。失败返回负值。成功后，nan\_func所有权将转移给驱动程序，并且它可以在此函数的作用域之外访问它。驱动程序在不再需要nan\_func时应通过调用cfg80211\_free\_nan\_func()来释放nan\_func。成功后，驱动程序应在所提供的nan\_func中 分配一个instance\_id。

del\_nan\_func

删除一个NAN函数。

nan\_change\_conf

更改NAN配置。在changes中必须指定更改的参数（使用enum cfg80211\_nan\_conf\_changes）；所有其他参数必须被忽略。

set\_multicast\_to\_unicast

为BSS配置组播到单播转换。

get\_txq\_stats

获取接口或PHY的TXQ统计信息。如果wdev为NULL，则此函数应返回PHY统计信息，否则返回接口统计信息。

set\_pmk

配置用于离线802.1x 4路握手的PMK。如果未通过del\_pmk删除，则PMK将保持有效，直到断开连接，此时驱动程序应将其清除。（在持有wireless\_dev互斥锁的情况下调用）

del\_pmk

为给定的认证器删除先前配置的PMK。 （在持有wireless\_dev互斥锁的情况下调用）

external\_auth

表示用户空间的离线认证处理结果

tx\_control\_port

TX控制端口帧（EAPoL）。noencrypt参数告诉驱动程序应该不加密帧。

**说明**

这个结构由fullmac卡驱动程序和/或无线堆栈注册，以处理它们的接口上的配置请求。

除非另有说明，否则所有回调应在成功时返回0或负错误代码。

目前，所有操作都在rtnl下调用，以保持与无线扩展的一致性，但是这将在此代码更广泛使用并且我们有第一个没有wext的用户时重新评估。

#### struct vif\_params

说明虚拟接口参数

**定义**

struct vif\_params {

u32 flags;

int use\_4addr;

u8 macaddr[ETH\_ALEN];

const u8 \*vht\_mumimo\_groups;

const u8 \*vht\_mumimo\_follow\_addr;

};

**成员**

flags

监视器接口标志，如果为0则不变，否则将设置MONITOR\_FLAG\_CHANGED

use\_4addr

使用4地址帧

macaddr

用于此虚拟接口的地址。如果将该参数设置为零地址，则驱动程序可以根据需要确定地址。只有完全支持 NL80211\_FEATURE\_MAC\_ON\_CREATE 标志的驱动程序才完全支持此功能。其他人可能仅支持使用指定的MAC创建仅p2p设备。

vht\_mumimo\_groups

MU-MIMO groupID，用于监视属于该MU-MIMO groupID的MU-MIMO数据包；如果未更改，则为NULL

vht\_mumimo\_follow\_addr

MU-MIMO跟随地址，用于监视发往指定站点的MU-MIMO数据包；如果未更改，则为NULL

#### struct key\_params

密钥信息

**定义**

struct key\_params {

const u8 \*key;

const u8 \*seq;

int key\_len;

int seq\_len;

u32 cipher;

};

**成员**

key

密钥材料

seq

TKIP和CCMP密钥的序列计数器（IV / PN），仅与get\_key（）回调一起使用，必须以小端表示，长度由seq\_len给出。

key\_len

密钥材料长度

seq\_len

seq的长度。

cipher

密码套件选择器

**说明**

关于密钥的信息

#### enum survey\_info\_flags

调查信息标志

**常量**

SURVEY\_INFO\_NOISE\_DBM

噪声（以dBm为单位）已填写

SURVEY\_INFO\_IN\_USE

频道当前正在使用

SURVEY\_INFO\_TIME

活动时间（以毫秒为单位）已填写

SURVEY\_INFO\_TIME\_BUSY

忙时间已填写

SURVEY\_INFO\_TIME\_EXT\_BUSY

已填写延长通道忙碌时间

SURVEY\_INFO\_TIME\_RX

接收时间已填写

SURVEY\_INFO\_TIME\_TX

传输时间已填写

SURVEY\_INFO\_TIME\_SCAN

扫描时间已填写

**说明**

驱动程序用于指示在get\_survey（）中填充了struct survey\_info中的哪些信息。

#### struct survey\_info

频道调查响应调查信息标志

**常量**

SURVEY\_INFO\_NOISE\_DBM

噪声（以dBm为单位）已填写

SURVEY\_INFO\_IN\_USE

频道当前正在使用

SURVEY\_INFO\_TIME

活动时间（以毫秒为单位）已填写

SURVEY\_INFO\_TIME\_BUSY

忙时间已填写

SURVEY\_INFO\_TIME\_EXT\_BUSY

已填写延长通道忙碌时间

SURVEY\_INFO\_TIME\_RX

接收时间已填写

SURVEY\_INFO\_TIME\_TX

传输时间已填写

SURVEY\_INFO\_TIME\_SCAN

扫描时间已填写

**说明**

驱动程序用于指示在get\_survey（）中填充了struct survey\_info中的哪些信息。

#### struct survey\_info

频道调查响应

**定义**

struct survey\_info {

struct ieee80211\_channel \*channel;

u64 time;

u64 time\_busy;

u64 time\_ext\_busy;

u64 time\_rx;

u64 time\_tx;

u64 time\_scan;

u32 filled;

s8 noise;

};

**成员**

channel

此调查记录报告的通道，对于单个记录报告全局统计数据可能为NULL

time

无线电打开的时间（在通道上）的毫秒数

time\_busy

主通道被感知为忙碌的时间

time\_ext\_busy

扩展通道被感知为忙碌的时间

time\_rx

无线电接收数据所花费的时间

time\_tx

无线电发送数据所花费的时间

time\_scan

无线电扫描所花费的时间

filled

枚举调查信息标志位的位标志

noise

通道噪声（分贝）,此字段及后续字段均为可选项

**说明**

用于dump\_survey()以报告每个通道的调查信息。

此结构体以后可以通过增加像通道占空比之类的东西进行扩展。

#### struct cfg80211\_beacon\_data

信标数据

**定义**

struct cfg80211\_beacon\_data {

const u8 \*head, \*tail;

const u8 \*beacon\_ies;

const u8 \*proberesp\_ies;

const u8 \*assocresp\_ies;

const u8 \*probe\_resp;

size\_t head\_len, tail\_len;

size\_t beacon\_ies\_len;

size\_t proberesp\_ies\_len;

size\_t assocresp\_ies\_len;

size\_t probe\_resp\_len;

};

**成员**

head

信标的头部（在 TIM IE 之前）或者NULL如果没有改变

tail

信标的尾部（在 TIM IE 之后）或NULL如果未更改

beacon\_ies

添加到信标帧中的额外信息元素或NULL

proberesp\_ies

添加到探测响应帧中的额外信息元素或NULL

assocresp\_ies

添加到（重新）关联响应帧或中的额外信息元素NULL

probe\_resp

探测响应模板（仅限 AP 模式）

head\_len

头长

tail\_len

尾巴的长度

beacon\_ies\_len

beacon\_ies 的八位字节长度

proberesp\_ies\_len

proberesp\_ies 的八位字节长度

assocresp\_ies\_len

以八位字节为单位的关联长度

probe\_resp\_len

探测响应模板的长度 ( probe\_resp )

#### struct cfg80211\_ap\_settings

AP 配置

**定义**

struct cfg80211\_ap\_settings {

struct cfg80211\_chan\_def chandef;

struct cfg80211\_beacon\_data beacon;

int beacon\_interval, dtim\_period;

const u8 \*ssid;

size\_t ssid\_len;

enum nl80211\_hidden\_ssid hidden\_ssid;

struct cfg80211\_crypto\_settings crypto;

bool privacy;

enum nl80211\_auth\_type auth\_type;

enum nl80211\_smps\_mode smps\_mode;

int inactivity\_timeout;

u8 p2p\_ctwindow;

bool p2p\_opp\_ps;

const struct cfg80211\_acl\_data \*acl;

bool pbss;

struct cfg80211\_bitrate\_mask beacon\_rate;

const struct ieee80211\_ht\_cap \*ht\_cap;

const struct ieee80211\_vht\_cap \*vht\_cap;

bool ht\_required, vht\_required, he\_required, sae\_h2e\_required;

};

**成员**

chandef

定义要使用的通道

beacon

信标数据

beacon\_interval

信标间隔

dtim\_period

DTIM 期间

ssid

BSS 中使用的 SSID（注意：NULL如果用户空间未提供，则可能是）

ssid\_len

ssid的长度

hidden\_ssid

是否在 Beacon/Probe Response 帧中隐藏 SSID

crypto

加密设置

privacy

BSS 使用隐私

auth\_type

身份验证类型（算法）

smps\_mode

开关电源模式

inactivity\_timeout

以秒为单位的时间来确定站的不活动。

p2p\_ctwindow

P2P CT窗口

p2p\_opp\_ps

P2P机会主义PS

acl

支持基于 MAC 地址的访问控制的驱动程序使用的 ACL 配置

pbss

如果设置，则作为 PCP 而不是 AP 开始。与 DMG 网络相关。

beacon\_rate

用于信标的比特率

ht\_cap

HT 功能（或者NULL如果 HT 未启用）

vht\_cap

VHT 功能（或者NULL如果未启用 VHT）

ht\_required

站点必须支持 HT

vht\_required

站点必须支持 VHT

**说明**

用于配置AP接口。

#### struct station\_parameters

站参数

**定义**

struct station\_parameters {

const u8 \*supported\_rates;

struct net\_device \*vlan;

u32 sta\_flags\_mask, sta\_flags\_set;

u32 sta\_modify\_mask;

int listen\_interval;

u16 aid;

u16 peer\_aid;

u8 supported\_rates\_len;

u8 plink\_action;

u8 plink\_state;

const struct ieee80211\_ht\_cap \*ht\_capa;

const struct ieee80211\_vht\_cap \*vht\_capa;

u8 uapsd\_queues;

u8 max\_sp;

enum nl80211\_mesh\_power\_mode local\_pm;

u16 capability;

const u8 \*ext\_capab;

u8 ext\_capab\_len;

const u8 \*supported\_channels;

u8 supported\_channels\_len;

const u8 \*supported\_oper\_classes;

u8 supported\_oper\_classes\_len;

u8 opmode\_notif;

bool opmode\_notif\_used;

int support\_p2p\_ps;

const struct ieee80211\_he\_cap\_elem \*he\_capa;

u8 he\_capa\_len;

};

**成员**

supported\_rates

IEEE 802.11 格式支持的速率（或 NULL 表示没有变化）

vlan

vlan接口站应该属于

sta\_flags\_mask

更改的站标志（BIT（NL80211\_STA\_FLAG\_...）的位掩码）

sta\_flags\_set

站标志值（BIT（NL80211\_STA\_FLAG\_...）的位掩码）

sta\_modify\_mask

指示哪些参数已更改的位图（对于那些没有自然“无更改”值的参数），请参阅enum station\_parameters\_apply\_mask

listen\_interval

监听间隔或 -1 表示没有变化

aid

AID 或零表示没有变化

peer\_aid

网状对等 AID 或零表示无变化

supported\_rates\_len

支持的速率数

plink\_action

要采取的 plink 操作

plink\_state

设置站点的对等链路状态

ht\_capa

站点HT能力

vht\_capa

站点的VHT能力

uapsd\_queues

为 uapsd 配置的队列位图。与 QoS 信息字段中的 AC 位图格式相同

max\_sp

最长服务期。与 QoS 信息字段中的 MAX\_SP 相同的格式（但已经下移）

local\_pm

本地链路特定网状网络省电模式（设置为未知时无变化）

capability

站点能力

ext\_capab

站的扩展能力

ext\_capab\_len

扩展能力的数量

supported\_channels

IEEE 802.11 格式支持的频道

supported\_channels\_len

支持频道数

supported\_oper\_classes

支持 IEEE 802.11 格式的操作类

supported\_oper\_classes\_len

支持的操作类别数

opmode\_notif

来自操作模式通知的操作模式字段

opmode\_notif\_used

使用操作模式字段时的信息

support\_p2p\_ps

站是否支持P2P PS机制的信息

he\_capa

站的HE能力

he\_capa\_len

HE能力的长度

**说明**

用于更改和创建新站

#### enum rate\_info\_flags

比特率信息标志

**常量**

RATE\_INFO\_FLAGS\_MCS

mcs 字段填充了 HT MCS

RATE\_INFO\_FLAGS\_VHT\_MCS

mcs 字段填充了 VHT MCS

RATE\_INFO\_FLAGS\_SHORT\_GI

400ns 保护间隔

RATE\_INFO\_FLAGS\_60G

60GHz 主控系统

RATE\_INFO\_FLAGS\_HE\_MCS

HE MCS 信息

**说明**

驱动程序用于表示802.11n传输的特定速率类型

#### struct rate\_info

比特率信息

**定义**

struct rate\_info {

u8 flags;

u8 mcs;

u16 legacy;

u8 nss;

u8 bw;

u8 he\_gi;

u8 he\_dcm;

u8 he\_ru\_alloc;

};

**成员**

flags

枚举rate\_info\_flags的标志位

mcs

如果结构说明HT/VHT/HE速率，则为mcs索引

legacy

802.11abg的100kbit/s比特率

nss

流的数量（仅限VHT和HE）

bw

带宽（来自枚举rate\_info\_bw）

he\_gi

HE保护间隔（来自枚举nl80211\_he\_gi）

he\_dcm

HE DCM值

he\_ru\_alloc

HE RU分配（来自枚举nl80211\_he\_ru\_alloc，仅当bw为RATE\_INFO\_BW\_HE\_RU时才有效）

**说明**

有关接收或传输比特率的信息。

#### struct station\_info

站点信息

**定义**

struct station\_info {

u64 filled;

u32 connected\_time;

u32 inactive\_time;

u64 rx\_bytes;

u64 tx\_bytes;

u16 llid;

u16 plid;

u8 plink\_state;

s8 signal;

s8 signal\_avg;

u8 chains;

s8 chain\_signal[IEEE80211\_MAX\_CHAINS];

s8 chain\_signal\_avg[IEEE80211\_MAX\_CHAINS];

struct rate\_info txrate;

struct rate\_info rxrate;

u32 rx\_packets;

u32 tx\_packets;

u32 tx\_retries;

u32 tx\_failed;

u32 rx\_dropped\_misc;

struct sta\_bss\_parameters bss\_param;

struct nl80211\_sta\_flag\_update sta\_flags;

int generation;

const u8 \*assoc\_req\_ies;

size\_t assoc\_req\_ies\_len;

u32 beacon\_loss\_count;

s64 t\_offset;

enum nl80211\_mesh\_power\_mode local\_pm;

enum nl80211\_mesh\_power\_mode peer\_pm;

enum nl80211\_mesh\_power\_mode nonpeer\_pm;

u32 expected\_throughput;

u64 rx\_beacon;

u64 rx\_duration;

u8 rx\_beacon\_signal\_avg;

struct cfg80211\_tid\_stats \*pertid;

s8 ack\_signal;

s8 avg\_ack\_signal;

};

**成员**

filled

使用nl80211\_sta\_info位的枚举的标志位的位标志，用于表示此结构体的相关值

connected\_time

自站点上次连接以来的时间（以秒为单位）

inactive\_time

上次站点活动（tx/rx）以来的时间（以毫秒为单位）

rx\_bytes

从此站点接收的字节（MPDU的大小）

tx\_bytes

传输到此站点的字节（MPDU的大小）

llid

网状本地链路ID

plid

网状对等链路ID

plink\_state

网状对等链路状态

signal

信号强度，类型取决于wiphy的signal\_type。对于CFG80211\_SIGNAL\_TYPE\_MBM，值以dBm为单位。

signal\_avg

平均信号强度，类型取决于wiphy的signal\_type。对于CFG80211\_SIGNAL\_TYPE\_MBM，值以dBm为单位。

chains

chain\_signal和chain\_signal\_avg中填充值的位掩码

chain\_signal

最后接收到的数据包的每个链路信号强度（以dBm为单位）

chain\_signal\_avg

每个链路信号强度的平均值（以dBm为单位）

txrate

从此站点的当前单播比特率

rxrate

发往此站点的当前单播比特率

rx\_packets

从此站点接收的数据包（MSDU和MMPDU）

tx\_packets

实际传输到此站点的数据包（MSDU和MMPDU）

tx\_retries

累计重试次数（MPDU）

tx\_failed

传输失败的数量（MPDU）（超过重试次数，未收到ACK）

rx\_dropped\_misc

未指定原因而被丢弃的数量。

bss\_param

当前BSS参数

sta\_flags

站点标志掩码和值

generation

nl80211转储的代数编号。此编号应在每次站点列表更改时增加，即添加或删除站点时，以便用户空间可以确定是否获得一致的快照。

assoc\_req\_ies

（重新）关联请求的IE。仅在具有不使用用户空间MLME/SME实现的驱动程序的AP模式下使用。提供信息用于cfg80211\_new\_sta（）调用，以通知用户空间IE。

assoc\_req\_ies\_len

assoc\_req\_ies缓冲区的长度（以八位字节为单位）。

beacon\_loss\_count

信标丢失事件触发的次数。

t\_offset

相对于此主机的站点的时间偏移量。

local\_pm

本地网状STA省电模式

peer\_pm

对等网状STA省电模式

nonpeer\_pm

非对等网状STA省电模式

expected\_throughput

即将升级至此站点的kbps预期吞吐量（包括802.11头）。

rx\_duration

所有来自对等体的帧的聚合PPDU持续时间（微秒）

rx\_beacon

从此对等体接收的信标数

rx\_beacon\_signal\_avg

从此对等体接收的信标的信号强度平均值（以dBm为单位）

pertid

每个TID的统计信息，请参见struct cfg80211\_tid\_stats，使用最后一个（IEEE80211\_NUM\_TIDS）索引用于未封装在QoS-MPDUs中的MSDU。请注意，这不使用填充位，但如果非空则使用。

ack\_signal

最后一帧ACK的信号强度（以dBm为单位）。

avg\_ack\_signal

通过已发送的MSDU数量计算出来的平均rssi值的ack数据包。

**说明**

驱动程序填充的站点信息，用于get\_station（）和dump\_station。

#### enum monitor\_flags

监视器标志

**常量**

MONITOR\_FLAG\_CHANGED

如果更改了标志，则设置

MONITOR\_FLAG\_FCSFAIL

传输带有错误FCS的帧

MONITOR\_FLAG\_PLCPFAIL

传输带有错误PLCP的帧

MONITOR\_FLAG\_CONTROL

传输控制帧

MONITOR\_FLAG\_OTHER\_BSS

禁用BSSID过滤

MONITOR\_FLAG\_COOK\_FRAMES

在处理后报告帧

MONITOR\_FLAG\_ACTIVE

主动监视器，对其MAC地址的帧进行确认

**说明**

监视器接口配置标志。请注意，这些必须按照nl80211标志的位。

#### enum mpath\_info\_flags

网状路径信息标志

**常量**

MPATH\_INFO\_FRAME\_QLEN

frame\_qlen 填充

MPATH\_INFO\_SN

sn 填充

MPATH\_INFO\_METRIC

metric 填充

MPATH\_INFO\_EXPTIME

exptime 填充

MPATH\_INFO\_DISCOVERY\_TIMEOUT

discovery\_timeout 填充

MPATH\_INFO\_DISCOVERY\_RETRIES

discovery\_retries 填充

MPATH\_INFO\_FLAGS

flags 填充

**说明**

驱动程序用来指示它在 get\_station() 或 dump\_station() 中已经填充了 struct mpath\_info 中的哪些信息。

#### struct mpath\_info

网状路径信息

**定义**

struct mpath\_info {

u32 filled;

u32 frame\_qlen;

u32 sn;

u32 metric;

u32 exptime;

u32 discovery\_timeout;

u8 discovery\_retries;

u8 flags;

int generation;

};

**成员**

filled

来自枚举 mpath\_info\_flags 的标志位域

frame\_qlen

此目的地的排队帧数

sn

目标序列号

metric

此网状路径的度量（成本）

exptime

从现在开始网状路径的过期时间，以毫秒为单位

discovery\_timeout

总的网状路径发现超时时间，以毫秒为单位

discovery\_retries

网状路径发现重试

flags

网状路径标志

generation

nl80211 dumps 的生成号。这个数字应该在网状路径列表发生变化时增加，即当添加或删除一个站点时，因此用户空间可以知道它是否得到了一致的快照。

**说明**

驱动程序用于获取 get\_mpath() 和 dump\_mpath() 的网状路径信息。

#### struct bss\_parameters

BSS 参数

**定义**

struct bss\_parameters {

int use\_cts\_prot;

int use\_short\_preamble;

int use\_short\_slot\_time;

const u8 \*basic\_rates;

u8 basic\_rates\_len;

int ap\_isolate;

int ht\_opmode;

s8 p2p\_ctwindow, p2p\_opp\_ps;

};

**成员**

use\_cts\_prot

是否使用 CTS 保护（0 = 否，1 = 是，-1 = 不更改）

use\_short\_preamble

是否允许使用短前导码（0 = 否，1 = 是，-1 = 不更改）

use\_short\_slot\_time

是否允许使用短时隙时间（0 = 否，1 = 是，-1 = 不更改）

basic\_rates

基本速率以 IEEE 802.11 格式表示（或 NULL 表示不更改）

basic\_rates\_len

基本速率数

ap\_isolate

不转发连接的站点之间的数据包（0 = 否，1 = 是，-1 = 不更改）

ht\_opmode

HT 操作模式（u16 = opmode，-1 = 不更改）

p2p\_ctwindow

P2P CT 窗口（-1 = 不更改）

p2p\_opp\_ps

P2P 机会式 PS（-1 = 不更改）

**说明**

用于更改 BSS 参数（主要用于 AP 模式）。

#### struct ieee80211\_txq\_params

TX队列参数

**定义**

struct ieee80211\_txq\_params {

enum nl80211\_ac ac;

u16 txop;

u16 cwmin;

u16 cwmax;

u8 aifs;

};

**成员**

ac

AC 标识符

txop

以 32 微秒为单位的最大爆发时间，0 表示禁用

cwmin

最小争用窗口[一个形式为 2^n-1 的值，在 1..32767 范围内]

cwmax

最大争用窗口[一个形式为 2^n-1 的值，在 1..32767 范围内]

aifs

仲裁间隔[0..255]

#### struct cfg80211\_crypto\_settings

加密设置

**定义**

struct cfg80211\_crypto\_settings {

u32 wpa\_versions;

u32 cipher\_group;

int n\_ciphers\_pairwise;

u32 ciphers\_pairwise[NL80211\_MAX\_NR\_CIPHER\_SUITES];

int n\_akm\_suites;

u32 akm\_suites[NL80211\_MAX\_NR\_AKM\_SUITES];

bool control\_port;

\_\_be16 control\_port\_ethertype;

bool control\_port\_no\_encrypt;

bool control\_port\_over\_nl80211;

struct key\_params \*wep\_keys;

int wep\_tx\_key;

const u8 \*psk;

};

**成员**

wpa\_versions

表示启用了哪些（如果有）WPA 版本（来自 enum nl80211\_wpa\_versions）

cipher\_group

组密钥密码套件（或 0 如果未设置）

n\_ciphers\_pairwise

AP 支持的单播密码的数量

ciphers\_pairwise

单播密钥密码套件

n\_akm\_suites

AKM 套件数量

akm\_suites

AKM 套件

control\_port

用户空间是否控制 IEEE 802.1X 端口，即设置/清除 NL80211\_STA\_FLAG\_AUTHORIZED。如果为 true，则驱动程序需要假定该端口未经授权，直到由用户空间授权为止。否则，默认情况下端口被标记为经过授权。

control\_port\_ethertype

应允许通过未经授权的端口的控制端口协议

control\_port\_no\_encrypt

TRUE 以防止加密控制端口协议帧。

control\_port\_over\_nl80211

TRUE 如果用户空间希望通过 NL80211 交换控制端口帧，而不是通过网络接口。

wep\_keys

静态 WEP 密钥，如果不为 NULL，则指向 CFG80211\_MAX\_WEP\_KEYS WEP 密钥数组

wep\_tx\_key

默认 TX 静态 WEP 密钥的键索引（0..3）

psk

PSK（对于支持 4 方握手卸载的设备）

#### struct cfg80211\_auth\_request

身份验证请求数据

**定义**

struct cfg80211\_auth\_request {

struct cfg80211\_bss \*bss;

const u8 \*ie;

size\_t ie\_len;

enum nl80211\_auth\_type auth\_type;

const u8 \*key;

u8 key\_len;

s8 key\_idx;

const u8 \*auth\_data;

size\_t auth\_data\_len;

};

**成员**

bss

要进行身份验证的 BSS，如果需要保留它，则必须在需要时获得对它的引用。

ie

要添加到身份验证帧的附加 IE 或 NULL

ie\_len

ie 缓冲区的长度（以八位字节为单位）

auth\_type

身份验证类型（算法）

key

用于共享密钥身份验证的 WEP 密钥

key\_len

用于共享密钥身份验证的 WEP 密钥长度

key\_idx

用于共享密钥身份验证的 WEP 密钥索引

auth\_data

身份验证帧中的字段和元素。这个变量包含身份验证帧主体（非 IE 和 IE 数据），不包括身份验证算法号，即从身份验证事务序列号字段开始。

auth\_data\_len

auth\_data 缓冲区的长度（以八位字节为单位）

**说明**

此结构提供了完成 IEEE 802.11 身份验证所需的信息。

#### struct cfg80211\_assoc\_request

（重新）关联请求数据

**定义**

struct cfg80211\_assoc\_request {

struct cfg80211\_bss \*bss;

const u8 \*ie, \*prev\_bssid;

size\_t ie\_len;

struct cfg80211\_crypto\_settings crypto;

bool use\_mfp;

u32 flags;

struct ieee80211\_ht\_cap ht\_capa;

struct ieee80211\_ht\_cap ht\_capa\_mask;

struct ieee80211\_vht\_cap vht\_capa, vht\_capa\_mask;

const u8 \*fils\_kek;

size\_t fils\_kek\_len;

const u8 \*fils\_nonces;

};

**成员**

bss

要进行关联的BSS。如果调用成功，驱动程序会获得一个引用，必须将其返回给cfg80211\_send\_rx\_assoc（）或cfg80211\_assoc\_timeout（）。为确保正确的引用计数，已经关联的新关联请求必须被拒绝。这也适用于links.bss参数，该参数用于MLO关联代替该参数（它为NULL）。

ie

额外的IE要添加到（重新）关联请求帧或NULL

prev\_bssid

先前的BSSID，如果不为空，请使用重新关联帧。这用于指示在ESS内重新关联的请求，而不是请求与ESS进行初始关联。当包括时，它设置为当前关联的BSSID，即包括在重新关联请求帧的当前AP地址字段中的值。

ie\_len

ie缓冲区的长度（以八位字节为单位）

crypto

加密设置

use\_mfp

在此关联中使用管理帧保护（IEEE 802.11w）

flags

参见枚举cfg80211\_assoc\_req\_flags

ht\_capa

HT能力重写。在ht\_capa\_mask中设置的值将在ht\_capa中使用。不支持的值将被忽略。

ht\_capa\_mask

要使用的ht\_capa位。

vht\_capa

VHT能力覆盖

vht\_capa\_mask

指示要使用哪些字段的VHT能力掩码

fils\_kek

用于保护（重新）关联请求/响应帧的FILS KEK，如果未使用FILS，则为NULL。

fils\_kek\_len

fils\_kek的长度（以八位字节为单位）

fils\_nonces

用于保护（重新）关联请求/响应帧的FILS nonce（AAD的一部分），如果未使用FILS，则为NULL。该字段以16个八位字节的STA nonce开头，后跟16个八位字节的AP nonce。

**说明**

该结构提供完成IEEE 802.11（重新）关联所需的信息。

#### struct cfg80211\_deauth\_request

注销请求数据

**定义**

struct cfg80211\_deauth\_request {

const u8 \*bssid;

const u8 \*ie;

size\_t ie\_len;

u16 reason\_code;

bool local\_state\_change;

};

**成员**

bssid

要注销的BSSID或AP MLD地址

ie

要将其添加到注销帧中的额外IE或NULL

ie\_len

ie缓冲区的长度（以八位字节为单位）

reason\_code

注销的原因代码

local\_state\_change

如果设置，仅更改本地状态，不设置注销帧

**说明**

该结构提供完成IEEE 802.11注销所需的信息。

#### struct cfg80211\_disassoc\_request

取消关联请求数据

**定义**

struct cfg80211\_disassoc\_request {

struct cfg80211\_bss \*bss;

const u8 \*ie;

size\_t ie\_len;

u16 reason\_code;

bool local\_state\_change;

};

**成员**

bss

要解除关联的 BSS

ie

额外的 IE 添加到解除关联框架或NULL

ie\_len

ie 缓冲区的八位字节长度

reason\_code

解除关联的原因代码

local\_state\_change

这只是对本地状态的请求，即不发送解除关联帧。

**说明**

该结构提供完成IEEE 802.11取消关联所需的信息。

#### struct cfg80211\_ibss\_params

IBSS 参数

**定义**

struct cfg80211\_ibss\_params {

const u8 \*ssid;

const u8 \*bssid;

struct cfg80211\_chan\_def chandef;

const u8 \*ie;

u8 ssid\_len, ie\_len;

u16 beacon\_interval;

u32 basic\_rates;

bool channel\_fixed;

bool privacy;

bool control\_port;

bool control\_port\_over\_nl80211;

bool userspace\_handles\_dfs;

int mcast\_rate[NUM\_NL80211\_BANDS];

struct ieee80211\_ht\_cap ht\_capa;

struct ieee80211\_ht\_cap ht\_capa\_mask;

struct key\_params \*wep\_keys;

int wep\_tx\_key;

};

**成员**

ssid

SSID，将始终为非null。

bssid

请求固定的BSSID，可以为NULL，如果设置了，则不会搜索具有不同BSSID的IBSS。

chandef

定义要使用的通道（如果找不到其他IBSS可加入）

ie

要包含在信标中的信息元素

ssid\_len

SSID的长度，将始终为非零。

ie\_len

长度

beacon\_interval

要使用的信标间隔

basic\_rates

创建IBSS时要使用的基本率的位图

channel\_fixed

通道应该固定-不要在其他频道上查找要加入的IBSS。

privacy

这是一个受保护的网络，在加入后将配置密钥

control\_port

用户空间是否控制IEEE 802.1X端口，即设置/清除NL80211\_STA\_FLAG\_AUTHORIZED。如果为真，则驱动程序需要假定该端口未经授权，直到由用户空间授权。否则，默认情况下将标记授权。

control\_port\_over\_nl80211

如果用户空间希望通过NL80211而不是网络接口交换控制端口帧，则为TRUE。

userspace\_handles\_dfs

用户空间是否控制DFS操作，即在检测到雷达时更改通道。这是操作DFS频道所必需的。

mcast\_rate

每个频带的多播速率索引+1（0：禁用）

ht\_capa

HT能力重写。在ht\_capa\_mask中设置的值将在ht\_capa中使用。不支持的值将被忽略。

ht\_capa\_mask

要使用的ht\_capa位。

wep\_keys

静态WEP密钥，如果不为NULL，则指向CFG80211\_MAX\_WEP\_KEYS WEP密钥数组

wep\_tx\_key

默认TX静态WEP密钥的密钥索引（0..3）

**说明**

该结构定义了加入\_ibss（）方法的IBSS参数。

#### struct cfg80211\_connect\_params

连接参数

**定义**

struct cfg80211\_connect\_params {

struct ieee80211\_channel \*channel;

struct ieee80211\_channel \*channel\_hint;

const u8 \*bssid;

const u8 \*bssid\_hint;

const u8 \*ssid;

size\_t ssid\_len;

enum nl80211\_auth\_type auth\_type;

const u8 \*ie;

size\_t ie\_len;

bool privacy;

enum nl80211\_mfp mfp;

struct cfg80211\_crypto\_settings crypto;

const u8 \*key;

u8 key\_len, key\_idx;

u32 flags;

int bg\_scan\_period;

struct ieee80211\_ht\_cap ht\_capa;

struct ieee80211\_ht\_cap ht\_capa\_mask;

struct ieee80211\_vht\_cap vht\_capa;

struct ieee80211\_vht\_cap vht\_capa\_mask;

bool pbss;

struct cfg80211\_bss\_selection bss\_select;

const u8 \*prev\_bssid;

const u8 \*fils\_erp\_username;

size\_t fils\_erp\_username\_len;

const u8 \*fils\_erp\_realm;

size\_t fils\_erp\_realm\_len;

u16 fils\_erp\_next\_seq\_num;

const u8 \*fils\_erp\_rrk;

size\_t fils\_erp\_rrk\_len;

bool want\_1x;

};

**成员**

channel

使用的频道，如果没有指定则为NULL（根据扫描结果自动选择）

channel\_hint

首选BSS的频道，如果没有指定则为NULL

bssid

AP BSSID或NULL（根据扫描结果自动选择）

bssid\_hint

连接到BSS的首选AP的BSSID，如果没有指定，则为NULL。与bssid参数不同，如果有更好的BSS可用，则驱动程序可以忽略此bssid\_hint。

ssid

SSID

ssid\_len

ssid的八位字节长度

auth\_type

身份验证类型（算法）

ie

关联请求的IEs

ie\_len

关联\_ie的八位字节长度

privacy

指示是否使用启用隐私的APs

mfp

指示是否使用管理帧保护

crypto

加密设置

key

共享密钥认证的WEP密钥

key\_len

共享密钥认证的WEP密钥长度

key\_idx

共享密钥认证的WEP密钥索引

flags

请参见枚举cfg80211\_assoc\_req\_flags

bg\_scan\_period

后台扫描周期（以秒为单位），如果为-1，则表示要使用默认值。

ht\_capa

HT Capability重写。在ht\_capa\_mask中设置的值将在ht\_capa中使用。未支持的值将被忽略。

ht\_capa\_mask

要使用的ht\_capa的位。

vht\_capa

VHT能力覆盖

vht\_capa\_mask

要使用的vht\_capa的位。

pbss

如果设置，连接到PCP而不是AP。仅适用于DMG网络。

bss\_select

用于BSS选择的标准。

prev\_bssid

上一BSSID，如果不为NULL，则使用重新关联帧。这用于指示在ESS内重新关联请求而不是在ESS内进行初始关联请求的请求。当包括时，将其设置为当前关联的BSSID，即包含在重新关联请求帧的当前AP地址域中的值。

fils\_erp\_username

EAP重新认证协议（ERP）NAI的用户名部分，如果未指定，则为NULL。这用于构建FILS包装的数据IE。

fils\_erp\_username\_len

fils\_erp\_username的八位字节长度。

fils\_erp\_realm

EAP重新认证协议（ERP）NAI的域部分，如果未指定，则为NULL。这指定了ER服务器的域名，并用于构建FILS包装的数据IE。

fils\_erp\_realm\_len

fils\_erp\_realm的八位字节长度。

fils\_erp\_next\_seq\_num

在FILS ERP消息中使用的下一个序列号。这也用于构建FILS包装的数据IE。

fils\_erp\_rrk

用于在FILS中派生其他密钥的ERP重新认证根密钥（rRK）或NULL（如果未指定）。

fils\_erp\_rrk\_len

fils\_erp\_rrk的八位字节长度。

want\_1x

指示用户空间支持并希望使用802.1X驱动程序卸载4路握手。

**说明**

该结构提供完成IEEE 802.11身份验证和关联所需的信息。

#### struct cfg80211\_pmksa

PMK安全关联

**定义**

struct cfg80211\_pmksa {

const u8 \*bssid;

const u8 \*pmkid;

const u8 \*pmk;

size\_t pmk\_len;

const u8 \*ssid;

size\_t ssid\_len;

const u8 \*cache\_id;

};

**成员**

bssid

AP的BSSID（可能为NULL）。

pmkid

用于引用PMKSA的标识符。

pmk

由pmkid标识的PMKSA的PMK。这用于FILS STA的密钥派生。否则，NULL。

pmk\_len

pmk的长度。pmk的长度可能因用于生成此长度的哈希算法而异。

ssid

使用FILS缓存标识符时指定PMKSA有效的ESS的SSID（可能为NULL）。

ssid\_len

ssid的八位字节长度。

cache\_id

由FILS AP广告的2个八位字节缓存标识符，标识PMKSA的范围。只有在ssid\_len不为零时才有效（可能为NULL）。

**说明**

将此结构传递给PMKSA缓存的set/del\_pmksa（）方法。

#### void cfg80211\_rx\_mlme\_mgmt(struct [net\_device](https://www.kernel.org/doc/html/latest/networking/kapi.html" \l "c.net_device" \o "net_device) \*dev, const u8 \*buf, size\_t len)

已处理的MLME管理帧的通知

**参数**

struct net\_device \*dev

网络设备

const u8 \*buf

认证帧(头+正文)

size\_t len

帧数据的长度

**说明**

每当接收和处理认证、解关联或取消认证帧时，在站点模式下都会调用此函数。在通过cfg80211\_ops::auth()被要求进行身份验证后，驱动程序必须调用此函数或cfg80211\_auth\_timeout()。在通过cfg80211\_ops::assoc()被要求进行关联后，驱动程序必须调用此函数或cfg80211\_auth\_timeout()。在连接状态下，驱动程序必须将接收和处理的解关联和取消认证帧呼叫此函数。如果不能使用帧，因为它是未受保护的，则驱动程序必须调用函数cfg80211\_rx\_unprot\_mlme\_mgmt()。

此函数可能会休眠。调用方必须持有相应wdev的互斥锁。

#### void cfg80211\_auth\_timeout(struct net\_device \*dev，const u8 \* addr)

超时认证的通知

**参数**

struct net\_device \*dev

网络设备

const u8 \*addr

身份验证超时设备的MAC地址

**说明**

此函数可能会休眠。调用方必须持有相应wdev的互斥锁。

#### void cfg80211\_rx\_assoc\_resp(struct net\_device \* dev，struct cfg80211\_bss \* bss，const u8 \* buf，size\_t len，int uapsd\_queues)

已处理关联响应的通知

**参数**

struct net\_device \* dev

网络设备

struct cfg80211\_bss \* bss

请求关联的BSS，所有权指针在此调用中移动到cfg80211。

const u8 \* buf

认证帧(头+正文)

size\_t len

帧数据的长度

int uapsd\_queues

已配置uapsd的队列位图，与QoS信息字段中的AC位图具有相同的格式

**说明**

在通过cfg80211\_ops::assoc()被要求关联后，驱动程序必须调用此函数或cfg80211\_auth\_timeout()。

此函数可能会休眠。调用方必须持有相应wdev的互斥锁。

#### void cfg80211\_assoc\_timeout(struct net\_device \* dev，struct cfg80211\_bss \* bss)

超时的协会通知

**参数**

struct net\_device \* dev

网络设备

struct cfg80211\_bss \* bss

协会超时的BSS条目。

**说明**

此函数可能会休眠。调用方必须持有相应wdev的互斥锁。

#### void cfg80211\_tx\_mlme\_mgmt(struct net\_device \*dev，const u8 \* buf，size\_t len)

发送deauth/disassoc帧的通知

**参数**

struct net\_device \*dev

网络设备

const u8 \*buf

802.11帧(头+正文)

size\_t len

帧数据的长度

**说明**

在站点模式下，每当取消认证已被处理时，都会调用此函数。这包括接收到的取消认证帧和本地生成的帧。此函数可能会休眠。调用方必须持有相应wdev的互斥锁。

#### void cfg80211\_ibss\_joined(struct net\_device \*dev，const u8 \*bssid，struct ieee80211\_channel \*channel，gfp\_t gfp)

通知cfg80211设备已加入IBSS

**参数**

struct net\_device \*dev

网络设备

const u8 \*bssid

已加入IBSS的BSSID

struct ieee80211\_channel \*channel

已加入IBSS的通道

gfp\_t gfp

分配标志

**说明**

此函数通知cfg80211设备已加入IBSS或切换到不同的BSSID。在调用此函数之前，必须从IBSS接收到信标，或者必须使用本地生成的信标之一调用cfg80211\_inform\_bss{,\_frame}函数，这保证了始终有此IBSS的扫描结果。cfg80211将处理其余部分。

#### struct cfg80211\_connect\_resp\_params

连接响应参数

**定义**

struct cfg80211\_connect\_resp\_params {

int status;

const u8 \*bssid;

struct cfg80211\_bss \*bss;

const u8 \*req\_ie;

size\_t req\_ie\_len;

const u8 \*resp\_ie;

size\_t resp\_ie\_len;

struct cfg80211\_fils\_resp\_params fils;

enum nl80211\_timeout\_reason timeout\_reason;

};

**成员**

status

状态码，WLAN\_STATUS\_SUCCESS表示连接成功，如果您的设备无法给出真实的故障状态码，请使用WLAN\_STATUS\_UNSPECIFIED\_FAILURE。如果此调用用于报告由于超时（例如未从AP接收身份验证帧）而导致的故障而不是由AP的明确拒绝，则使用-1来表示这是一个故障，但没有状态码。如果超时原因则使用timeout\_reason报告超时的原因。

bssid

AP的BSSID（可能为NULL）

bss

STA连接到的BSS的条目，可以通过cfg80211\_get\_bss()获得（可能为NULL）。仅需要在bssid和bss之间指定一个参数。

req\_ie

关联请求IE（可能为空）

req\_ie\_len

关联请求IE长度

resp\_ie

关联响应IE（可能为空）

resp\_ie\_len

关联响应IE长度

fils

FILS连接响应参数。

timeout\_reason

连接超时的原因。这在连接由于超时而失败而不是由于AP的明确拒绝时使用。NL80211\_TIMEOUT\_UNSPECIFIED用于当不知道超时原因时。只有在status < 0时才使用此值来指示失败是由于超时而不是由AP的明确拒绝导致的。在其他情况下（status>=0），此值将被忽略。

#### void cfg80211\_connect\_done(struct net\_device \*dev, struct cfg80211\_connect\_resp\_params \*params, gfp\_t gfp)

通知cfg80211连接结果

**参数**

struct net\_device \*dev

网络设备

struct cfg80211\_connect\_resp\_params \*params

连接响应参数

gfp\_t gfp

分配标志

**说明**

一旦来自connect()的连接请求的执行已经完成，底层驱动程序应该调用它。这类似于cfg80211\_connect\_bss()，但使用结构指针用于连接响应参数。仅应调用cfg80211\_connect\_bss()，cfg80211\_connect\_result()，cfg80211\_connect\_timeout()和cfg80211\_connect\_done()中的一个函数。

#### void cfg80211\_connect\_result(struct net\_device \* dev, const u8 \* bssid, const u8 \* req\_ie, size\_t req\_ie\_len, const u8 \* resp\_ie, size\_t resp\_ie\_len, u16 status, gfp\_t gfp)

通知cfg80211连接结果

**参数**

struct net\_device \* dev

网络设备

const u8 \* bssid

AP的BSSID

const u8 \* req\_ie

关联请求IE（可能为空）

size\_t req\_ie\_len

关联请求IE长度

const u8 \* resp\_ie

关联响应IE（可能为空）

size\_t resp\_ie\_len

关联响应IE长度

u16 status

状态码，WLAN\_STATUS\_SUCCESS表示连接成功，如果您的设备无法给出真实的故障状态码，请使用WLAN\_STATUS\_UNSPECIFIED\_FAILURE。

gfp\_t gfp

分配标志

**说明**

执行来自connect()的连接请求后，底层驱动程序应该调用它。这类似于cfg80211\_connect\_bss()，允许指定确切的bss条目。仅应调用cfg80211\_connect\_bss()，cfg80211\_connect\_result()，cfg80211\_connect\_timeout()和cfg80211\_connect\_done()中的一个函数。

#### void cfg80211\_connect\_bss(struct net\_device \*dev, const u8 \*bssid, struct cfg80211\_bss \*bss, const u8 \*req\_ie, size\_t req\_ie\_len, const u8 \*resp\_ie, size\_t resp\_ie\_len, int status, gfp\_t gfp, enum nl80211\_timeout\_reason timeout\_reason)

通知cfg80211连接结果

**参数**

struct net\_device \*dev

网络设备

const u8 \*bssid

AP的BSSID

struct cfg80211\_bss \*bss

STA连接到的BSS的条目，可以通过cfg80211\_get\_bss()获得（可能为NULL）。但是，建议将bss存储在connect\_request中并保存对其的引用，并通过此参数返回，以避免在连接期间出现警告，尤其是对于实现connect op的驱动程序。仅需要在bssid和bss之间指定一个参数。

const u8 \*req\_ie

关联请求IE（可能为空）

size\_t req\_ie\_len

关联请求IE长度

const u8 \*resp\_ie

关联响应IE（可能为空）

size\_t resp\_ie\_len

关联响应IE长度

int status

状态码，WLAN\_STATUS\_SUCCESS表示连接成功，如果您的设备无法给出真实的故障状态码，请使用WLAN\_STATUS\_UNSPECIFIED\_FAILURE。如果此调用用于报告由于超时（例如未从AP接收身份验证帧）而导致的故障而不是由AP的明确拒绝，则使用-1来表示这是一个故障，但没有状态码。在这种情况下，timeout\_reason用于报告超时的原因。

gfp\_t gfp

分配标志

enum nl80211\_timeout\_reason timeout\_reason

连接超时的原因。这在连接由于超时而失败而不是由于AP的明确拒绝时使用。NL80211\_TIMEOUT\_UNSPECIFIED用于当不知道超时原因时。只有在status < 0时才使用此值来指示失败是由于超时而不是由AP的明确拒绝导致的。在其他情况下（status>=0），此值将被忽略。

**说明**

一旦来自connect（）的连接请求执行完成，应该由底层驱动程序进行调用。这类似于cfg80211\_connect\_result（），但具有标识连接的确切bss条目的选项。在cfg80211\_connect\_bss（），cfg80211\_connect\_result（），cfg80211\_connect\_timeout（）和cfg80211\_connect\_done（）中，只应调用其中一个函数。

#### void cfg80211\_connect\_timeout（struct net\_device \* dev，const u8 \* bssid，const u8 \* req\_ie，size\_t req\_ie\_len，gfp\_t gfp，enum nl80211\_timeout\_reason timeout\_reason）

通知cfg80211连接超时

**参数**

struct net\_device \* dev

网络设备

const u8 \* bssid

AP的BSSID

const u8 \* req\_ie

关联请求IE（可能为NULL）

size\_t req\_ie\_len

协会请求IE长度

gfp\_t gfp

分配标志

enum nl80211\_timeout\_reason timeout\_reason

连接超时的原因。

**说明**

如果在未收到来自AP的显式身份验证/关联拒绝的情况下，连接（）已经在序列中失败，则应由底层驱动程序进行调用。例如，由于无法发送身份验证或关联请求帧或在等待响应期间超时而发生。在cfg80211\_connect\_bss（），cfg80211\_connect\_result（），cfg80211\_connect\_timeout（）和cfg80211\_connect\_done（）中，只应调用其中一个函数。

#### void cfg80211\_roamed（struct net\_device \* dev，struct cfg80211\_roam\_info \* info，gfp\_t gfp）

通知cfg80211漫游

**参数**

struct net\_device \* dev

网络设备

struct cfg80211\_roam\_info \* info

有关新BSS的信息。 struct cfg80211\_roam\_info。

gfp\_t gfp

分配标志

**说明**

该驱动程序可以使用新AP的BSSID或传递bss条目来避免bss条目超时的比赛。每当连接时从一个AP漫游到另一个驱动程序时，都应该由底层驱动程序进行调用。在固件中实现漫游的驱动程序应传递bss条目以避免bss条目超时的比赛，在驱动程序中查看新AP的bss条目，但由于\_\_cfg80211\_roamed（）中的rdev-> event\_work调度延迟而在访问时被定时的情况。如果发生任何故障，则在cfg80211\_roamed（）或\_\_cfg80211\_romed（）中释放引用;否则，将在从当前bss断开连接时释放引用。

#### void cfg80211\_disconnected（struct net\_device \* dev，u16 reason，const u8 \* ie，size\_t ie\_len，bool locally\_generated，gfp\_t gfp）

通知cfg80211连接已取消

**参数**

struct net\_device \* dev

网络设备

u16 reason

取消连接的原因代码，如果未知，则将其设置为0

const u8 \* ie

逆向/ disassoc帧的信息元素（可能为空）

size\_t ie\_len

IE的长度

bool locally\_generated

本地请求了断开连接

gfp\_t gfp

分配标志

**说明**

在调用此函数之后，驱动程序应进入空闲状态，不再尝试连接到任何AP。

#### void cfg80211\_ready\_on\_channel（struct wireless\_dev \* wdev，u64 cookie，struct ieee80211\_channel \* chan，unsigned int duration，gfp\_t gfp）

保留在通道上的开始通知

**参数**

struct wireless\_dev \* wdev

无线设备

u64 cookie

请求cookie

struct ieee80211\_channel \* chan

当前通道（来自remain\_on\_channel请求）

unsigned int duration

驱动程序打算停留在该通道的毫秒数

gfp\_t gfp

分配标志

#### void cfg80211\_remain\_on\_channel\_expired（struct wireless\_dev \* wdev，u64 cookie，struct ieee80211\_channel \* chan，gfp\_t gfp）

remain\_on\_channel持续时间已过期

**参数**

struct wireless\_dev \* wdev

无线设备

u64 cookie

请求cookie

struct ieee80211\_channel \* chan

当前通道（来自remain\_on\_channel请求）

gfp\_t gfp

分配标志

#### void cfg80211\_new\_sta（struct net\_device \* dev，const u8 \* mac\_addr，struct station\_info \* sinfo，gfp\_t gfp）

通知用户空间关于站点

**参数**

struct net\_device \* dev

the netdev

const u8 \* mac\_addr

站点的地址

struct station\_info \* sinfo

车站信息

gfp\_t gfp

分配标志

#### bool cfg80211\_rx\_mgmt（struct wireless\_dev \* wdev，int freq，int sig\_dbm，const u8 \* buf，size\_t len，u32 flags）

收到未处理的管理帧的通知

**参数**

struct wireless\_dev \* wdev

接收帧的无线设备

int freq

以MHz接收帧的频率

int sig\_dbm

以dBm为单位的信号强度，如果未知，则为0

const u8 \* buf

管理帧（头+正文）

size\_t len

帧数据的长度

u32 flags

标志，如enum nl80211\_rxmgmt\_flags中定义

**说明**

每当接收到一个站点模式接口的操作帧但未在内核中处理时，都会调用此函数。

**返回**

如果已注册用户空间应用程序以接收此框架，则为true。对于操作帧，这使其负责拒绝未识别的操作帧；否则为false，在这种情况下，对于操作帧，驱动程序负责拒绝帧。

#### void cfg80211\_mgmt\_tx\_status（struct wireless\_dev \* wdev，u64 cookie，const u8 \* buf，size\_t len，bool ack，gfp\_t gfp）

管理帧的TX状态通知

**参数**

struct wireless\_dev \* wdev

接收帧的无线设备

u64 cookie

cfg80211\_ops :: mgmt\_tx（）返回的Cookie

const u8 \* buf

管理帧（头+正文）

size\_t len

帧数据的长度

bool ack

帧是否被确认

gfp\_t gfp

上下文标识

**说明**

每当请求使用 cfg80211\_ops:: 传输管理帧时，都会调用此函数mgmt\_tx()以报告传输尝试的 TX 状态

#### void cfg80211\_cqm\_rssi\_notify(struct [net\_device](https://www.kernel.org/doc/html/latest/networking/kapi.html" \l "c.net_device" \o "net_device) \*dev, enum nl80211\_cqm\_rssi\_threshold\_event rssi\_event, s32 rssi\_level, gfp\_t gfp)

连接质量监测RSSI事件

**参数**

struct net\_device \* dev

网络设备

enum nl80211\_cqm\_rssi\_threshold\_event rssi\_event

触发的RSSI事件

s32 rssi\_level

新的RSSI级别值，如果不可用，则为0

gfp\_t gfp

上下文标识

**说明**

当发生配置的连接质量监测rssi阈值达到事件时，将调用此函数。

void cfg80211\_cqm\_pktloss\_notify（struct net\_device \* dev，const u8 \* peer，u32 num\_packets，gfp\_t gfp）

通知用户空间，对等方的数据包丢失

**参数**

struct net\_device \* dev

网络设备

const u8 \* peer

对等方的MAC地址

u32 num\_packets

丢失的数据包数量-应为固定的阈值，但可能不少于50，或可能为吞吐量相关的阈值（以考虑暂时的干扰）

gfp\_t gfp

上下文标识

#### void cfg80211\_michael\_mic\_failure（struct net\_device \* dev，const u8 \* addr，enum nl80211\_key\_type key\_type，int key\_id，const u8 \* tsc，gfp\_t gfp）

Michael MIC（TKIP）失败通知

**参数**

struct net\_device \* dev

网络设备

const u8 \* addr

帧的源MAC地址

enum nl80211\_key\_type key\_type

接收到的帧使用的密钥类型

int key\_id

键标识符（0..3）。如果缺少，可以为-1。

const u8 \* tsc

生成MIC失败的帧的TSC值（6个八位组）

gfp\_t gfp

分配标志

**说明**

每当本地MAC在接收到的帧中检测到MIC失败时，将调用此函数。这与MLME-MICHAELMICFAILURE.:c:func:indication（）原语匹配。

### 扫描和BSS列表处理

扫描过程本身相当简单，但是cfg80211提供了相当多的辅助功能。要开始扫描，将使用扫描定义调用扫描操作。该扫描定义包含要扫描的频道以及要发送探测请求的SSID（包括通配符，如果需要）。通过没有要探测的SSID表示被动扫描。此外，扫描请求可能包含应添加到探测请求的额外信息元素。保证IE是格式正确的，并且不会超过驱动程序在wiphy结构中广告的最大长度。

当扫描找到BSS时，cfg80211需要被通知，因为它负责维护BSS列表；驱动程序不应自己维护列表。为此通知，存在各种函数。

由于驱动程序不维护BSS列表，因此还有许多函数可用于搜索BSS并从cfg80211维护的BSS结构中获取有关其信息。还将BSS列表提供给用户空间。

struct cfg80211\_ssid

SSID说明

**定义**

struct cfg80211\_ssid {

u8 ssid[IEEE80211\_MAX\_SSID\_LEN];

u8 ssid\_len;

};

**成员**

ssid

SSID

ssid\_len

ssid的长度

#### struct cfg80211\_scan\_request

扫描请求说明

**定义**

struct cfg80211\_scan\_request {

struct cfg80211\_ssid \*ssids;

int n\_ssids;

u32 n\_channels;

enum nl80211\_bss\_scan\_width scan\_width;

const u8 \*ie;

size\_t ie\_len;

u16 duration;

bool duration\_mandatory;

u32 flags;

u32 rates[NUM\_NL80211\_BANDS];

struct wireless\_dev \*wdev;

u8 mac\_addr[ETH\_ALEN] ;

u8 mac\_addr\_mask[ETH\_ALEN] ;

u8 bssid[ETH\_ALEN] ;

struct wiphy \*wiphy;

unsigned long scan\_start;

struct cfg80211\_scan\_info info;

bool notified;

bool no\_cck;

struct ieee80211\_channel \*channels[0];

};

**成员**

ssids

要扫描的SSID（仅主动扫描）

n\_ssids

SSID的数量

n\_channels

要扫描的频道总数

scan\_width

扫描的通道宽度

ie

要添加到探测请求中的可选信息元素或NULL

ie\_len

ie的八位组长度

duration

每个信道上侦听多长时间，以TUs为单位。如果未设置duration\_mandatory，则为最大停留时间，并且实际停留时间可能会更短。

duration\_mandatory

如果设置，则扫描持续时间必须如字段duration指定。

flags

控制操作的标志位字段

rates

每个频带广告的速率位图

wdev

要扫描的无线设备

mac\_addr

用于随机化的MAC地址

mac\_addr\_mask

用于随机化的MAC地址掩码，应随机化掩码中为0的位，应从mac\_addr中获取为1的位

bssid

要扫描的BSSID（最常见的是通配符BSSID）

wiphy

此项是为了什么

scan\_start

扫描开始时的时间（以jiffies为单位）

info

（内部）有关完成的扫描的信息

已通知

（内部）已通知扫描请求已完成或已中止

no\_cck

用于在2GHz频段中以非CCK速率发送探测请求

channels

要扫描的频道。

#### void cfg80211\_scan\_done（struct cfg80211\_scan\_request \* request，struct cfg80211\_scan\_info \* info）

通知扫描完成

**参数**

struct cfg80211\_scan\_request \* request

相应的扫描请求

struct cfg80211\_scan\_info \* info

已完成扫描的信息

#### struct cfg80211\_bss

BSS 说明

**定义**

struct cfg80211\_bss {

struct ieee80211\_channel \*channel;

enum nl80211\_bss\_scan\_width scan\_width;

const struct cfg80211\_bss\_ies \_\_rcu \*ies;

const struct cfg80211\_bss\_ies \_\_rcu \*beacon\_ies;

const struct cfg80211\_bss\_ies \_\_rcu \*proberesp\_ies;

struct cfg80211\_bss \*hidden\_beacon\_bss;

s32 signal;

u16 beacon\_interval;

u16 capability;

u8 bssid[ETH\_ALEN];

u8 chains;

s8 chain\_signal[IEEE80211\_MAX\_CHAINS];

u8 bssid\_index;

u8 priv[0] ;

};

**成员**

channel

该BSS所在的频道

scan\_width

控制频道的宽度

ies

信息元素（请注意，不能保证这些元素格式正确！）；这是一个指针，指向beacon\_ies或proberesp\_ies，具体取决于是否接收到了探查响应帧。它始终为非空。

beacon\_ies

来自上一个Beacon帧的信息元素（实现注意事项：如果hidden\_beacon\_bss设置了此结构体不拥有beacon\_ies，而只是指向从hidden\_beacon\_bss结构体中获得的指针）

proberesp\_ies

来自上一个探测响应帧的信息元素

hidden\_beacon\_bss

如果此BSS结构体表示隐藏其Beacon中的SSID的BSS的探测响应，则指向保存Beacon数据的BSS结构体。当然，beacon\_ies仍然有效，并在这种情况下指向与hidden\_beacon\_bss-> beacon\_ies相同的数据。

信号

信号强度值（类型取决于wiphy的信号类型）

beacon\_interval

来自帧的Beacon间隔

capability

主机字节顺序中的capability字段

bssid

BSS的BSSID

chains

chain\_signal已填写值的位掩码。

chain\_signal

dBm中上一个接收到的BSS的每个信道的信号强度。

priv

驱动程序使用的私有区域，至少有wiphy-> bss\_priv\_size字节

**说明**

此结构体用于说明扫描结果和类似情况下的BSS（也可能是网格网络）。

#### struct cfg80211\_inform\_bss

BSS 通知数据

**定义**

struct cfg80211\_inform\_bss {

struct ieee80211\_channel \*chan;

enum nl80211\_bss\_scan\_width scan\_width;

s32 signal;

u64 boottime\_ns;

u64 parent\_tsf;

u8 parent\_bssid[ETH\_ALEN] ;

u8 chains;

s8 chain\_signal[IEEE80211\_MAX\_CHAINS];

};

**成员**

chan

收到帧的通道

scan\_width

使用的扫描宽度

signal

信号强度值，根据wiphy的信号类型

boottime\_ns

收到信息时的时间戳（CLOCK\_BOOTTIME）；应该与设备实际接收帧的时间相匹配（不仅仅是主机，以防它被缓冲在设备上）并且精确到大约 10 毫秒。如果帧未被缓冲，则只传递 的返回 ktime\_get\_boot\_ns()可能是合适的。

parent\_tsf

帧的时间戳字段的第一个八位字节开始接收的时间。时间为 指定的 BSS 的 TSF parent\_bssid。

parent\_bssid

根据其parent\_tsf设置的 BSS。这被设置为请求接收到信标/探测的扫描的 BSS。

chains

chain\_signal中填充值的位掩码。

chain\_signal

最后接收到的 BSS 的每链信号强度，以 dBm 为单位。

#### struct [cfg80211\_bss](https://www.kernel.org/doc/html/v4.19/driver-api/80211/cfg80211.html" \l "c.cfg80211_bss" \o "cfg80211_bss) \* cfg80211\_inform\_bss\_frame\_data(struct [wiphy](https://www.kernel.org/doc/html/v4.19/driver-api/80211/cfg80211.html" \l "c.wiphy" \o "wiphy) \* wiphy, struct [cfg80211\_inform\_bss](https://www.kernel.org/doc/html/v4.19/driver-api/80211/cfg80211.html" \l "c.cfg80211_inform_bss" \o "cfg80211_inform_bss) \* data, struct ieee80211\_mgmt \* mgmt, size\_t len, gfp\_t gfp)

通知 cfg80211 收到 BSS 帧

**参数**

struct wiphy \* wiphy

报告 BSS 的 Wiphy

struct cfg80211\_inform\_bss \* data

BSS元数据

struct ieee80211\_mgmt \* mgmt

管理框架（探测响应或信标）

size\_t len

管理帧长度

gfp\_t gfp

上下文标志

**说明**

这会通知 cfg80211 已找到 BSS 信息，应该更新/添加 BSS。

**返回**

引用的结构，必须使用cfg80211\_put\_bss()! 或者NULL出错。

#### struct [cfg80211\_bss](https://www.kernel.org/doc/html/v4.19/driver-api/80211/cfg80211.html" \l "c.cfg80211_bss" \o "cfg80211_bss) \* cfg80211\_inform\_bss\_data(struct [wiphy](https://www.kernel.org/doc/html/v4.19/driver-api/80211/cfg80211.html" \l "c.wiphy" \o "wiphy) \* wiphy, struct [cfg80211\_inform\_bss](https://www.kernel.org/doc/html/v4.19/driver-api/80211/cfg80211.html" \l "c.cfg80211_inform_bss" \o "cfg80211_inform_bss) \* data, enum cfg80211\_bss\_frame\_type ftype, const u8 \* bssid, u64 tsf, u16 capability, u16 beacon\_interval, const u8 \* ie, size\_t ielen, gfp\_t gfp)

通知cfg80211有新的BSS

**参数**

struct wiphy \* wiphy

报告 BSS 的 Wiphy

struct cfg80211\_inform\_bss \* data

BSS元数据

enum cfg80211\_bss\_frame\_type ftype

帧类型（如果已知）

const u8 \* bssid

BSS的BSSID

u64 tsf

对端在信标/探测响应中发送的 TSF（或 0）

u16 capability

对方发送的能力字段

u16 beacon\_interval

对端公布的信标间隔

const u8 \* ie

对端发送的额外 IE

size\_t ielen

附加 IE 的长度

gfp\_t gfp

上下文标志

**描述**

这会通知 cfg80211 已找到 BSS 信息，应该更新/添加 BSS。

**返回**

引用的结构，必须使用cfg80211\_put\_bss()! 或者NULL出错。

#### void cfg80211\_unlink\_bss(struct [wiphy](https://www.kernel.org/doc/html/latest/driver-api/80211/cfg80211.html" \l "c.cfg80211_unlink_bss" \o "wiphy) \*wiphy, struct [cfg80211\_bss](https://www.kernel.org/doc/html/latest/driver-api/80211/cfg80211.html" \l "c.cfg80211_bss" \o "cfg80211_bss) \*bss)

取消 BSS 与内部数据结构的链接

**参数**

struct wiphy \*wiphy

wiphy

struct cfg80211\_bss \*bss

要删除的bss

**说明**

此函数从内部数据结构中删除给定的BSS，从而不再在扫描结果等中显示。在检测到BSS消失时使用此功能。通常BSS也将超时，因此根本不需要使用此函数。

#### constu8 \* cfg80211\_find\_ie（u8 eid，constu8 \* ies，int len）

在数据中查找信息元素

**参数**

u8 eid

元素ID

constu8 \* ies

由IE组成的数据

int len

数据长度

**返回**

如果找不到元素ID或元素无效（声称比给定数据长），则为NULL，或指向请求元素的第一个字节的指针，即包含元素ID的字节。

**注**

除必须适合给定数据之外，没有对元素长度的检查。

#### const u8 \* ieee80211\_bss\_get\_ie(struct [cfg80211\_bss](https://www.kernel.org/doc/html/v4.19/driver-api/80211/cfg80211.html" \l "c.cfg80211_bss" \o "cfg80211_bss) \* bss, u8 ie)

查找给定ID的IE

**参数**

struct cfg80211\_bss \*bss

要搜索的bss

u8 ie

IE ID

**说明**

注意返回是受RCU保护的指针，因此在调用此函数时必须持有rcu\_read\_lock()。

**返回**

如果未找到，则返回NULL。

### 实用函数

cfg80211提供了一些实用函数，可以很有用。

#### int ieee80211\_channel\_to\_frequency(int chan，enum nl80211\_band band)

将信道号转换为频率

**参数**

int chan

信道号

enum nl80211\_band band

由于信道号重叠而必要的频带

**返回**

相应的频率（以MHz为单位），如果转换失败则为0。

#### int ieee80211\_frequency\_to\_channel(int freq)

将频率转换为信道号

**参数**

int freq

中心频率（以MHz为单位）

**返回**

相应的信道号，如果转换失败则为0。

#### struct ieee80211\_channel \*ieee80211\_get\_channel(struct wiphy \*wiphy，int freq)

从wiphy获取指定频率的信道结构

**参数**

struct wiphy \*wiphy

要获取通道的struct wiphy

int freq

通道的中心频率（以MHz为单位）

**返回**

来自wiphy的频率。

#### const struct ieee80211\_rate \*ieee80211\_get\_response\_rate(struct ieee80211\_supported\_band \*sband，u32 basic\_rates，int bitrate)

获取给定速率的基本速率

**参数**

struct ieee80211\_supported\_band \* sbabd

查找速率的波段

u32 basic\_rates

基本速度的位图

int bitrate

要查找基本速率的比特率

**返回**

对于给定比特率的基本速率，即基本速率映射中包含的下一个较低比特率，在此函数中给出为带宽表中的速率索引位图。

#### unsigned int \_\_attribute\_const\_\_ ieee80211\_hdrlen(\_\_le16 fc)

从帧控制获取头长度（以字节为单位）

**参数**

\_\_le16 fc

以小端方式表示的帧控制字段

**返回**

标题长度（以字节为单位）。

#### unsigned int ieee80211\_get\_hdrlen\_from\_skb(const struct sk\_buff \*skb)

从数据中获取头长度

**参数**

const struct sk\_buff \*skb

框架

**说明**

给定一个skb，在data指针处有一个原始的802.11标头，此函数返回802.11标头长度。

**返回**

802.11标头长度（不包括加密标头）以字节为单位。如果sk\_buff中的数据太短而不能包含有效的802.11标头，则为0。

#### struct ieee80211\_radiotap\_iterator

跟踪遍历存在的radiotap参数

**定义**

struct ieee80211\_radiotap\_iterator {

struct ieee80211\_radiotap\_header \*\_rtheader;

const struct ieee80211\_radiotap\_vendor\_namespaces \*\_vns;

const struct ieee80211\_radiotap\_namespace \*current\_namespace;

unsigned char \*\_arg, \*\_next\_ns\_data;

\_\_le32 \*\_next\_bitmap;

unsigned char \*this\_arg;

int this\_arg\_index;

int this\_arg\_size;

int is\_radiotap\_ns;

int \_max\_length;

int \_arg\_index;

uint32\_t \_bitmap\_shifter;

int \_reset\_on\_ext;

};

**成员**

\_rtheader

正在遍历的radiotap头指针

\_vns

供应商命名空间定义

current\_namespace

指向当前命名空间定义的指针（或在内部为NULL，如果当前命名空间未知）

\_arg

下一个参数指针

\_next\_ns\_data

下一个命名空间的数据的开头

\_next\_bitmap

下一个存在的u32的内部指针

this\_arg

指向当前radiotap arg的指针；每次调用ieee80211\_radiotap\_iterator\_next()之后它都是有效的，但在ieee80211\_radiotap\_iterator\_init()之后它也是有效的，在那里它将指向实际数据部分的开头

this\_arg\_index

当前参数的索引，在每个成功调用ieee80211\_radiotap\_iterator\_next()之后有效

this\_arg\_size

当前arg的长度，用于方便起见

is\_radiotap\_ns

指示当前命名空间是否为默认radiotap命名空间

\_max\_length

以CPU字节顺序表示的radiotap标头长度

\_arg\_index

下一个参数索引

\_bitmap\_shifter

当前的u32位图的内部移位器，b0 set == arg present

\_reset\_on\_ext

内部；在转到下一个位图字时将参数索引重置为0

**说明**

说明radiotap解析器状态。用下划线前缀的字段不能被解析器的用户使用，只能被解析器内部使用。

### 数据路径助手

除了通用实用程序外，cfg80211还提供了帮助实现不在设备上进行802.11/802.3转换的设备的数据路径的函数。

#### int ieee80211\_data\_to\_8023(struct sk\_buff \*skb，const u8 \*addr，enum nl80211\_iftype iftype)

将802.11数据帧转换为802.3

**参数**

struct sk\_buff \*skb

802.11数据帧

const u8 \*addr

设备MAC地址

enum nl80211\_iftype iftype

虚拟接口类型

**返回**

成功返回0。错误返回非零值。

#### void ieee80211\_amsdu\_to\_8023s(struct sk\_buff \*skb，struct sk\_buff\_head \*list，const u8 \*addr，enum nl80211\_iftype iftype，const unsigned int extra\_headroom，const u8 \*check\_da，const u8 \*check\_sa)

解码IEEE 802.11n A-MSDU帧

**参数**

struct sk\_buff \*skb

没有任何头的输入A-MSDU帧。

struct sk\_buff\_head \*list

802.3帧的输出列表。必须由调用程序分配和初始化。

const u8 \*addr

设备MAC地址。

enum nl80211\_iftype iftype

设备接口类型。

const unsigned int extra\_headroom

列表中SKB的额外硬件头部。

const u8 \*check\_da

DA在内部以太网标头中要检查，或为NULL

const u8 \*check\_sa

SA在内部以太网标头中要检查，或为NULL

**说明**

解码一个IEEE 802.11 A-MSDU并将其转换为802.3帧的列表。如果解码失败则列表为空。在传递给此函数之前，skb必须是完全无头的; 它在此函数中被释放。

#### unsigned int cfg80211\_classify8021d(struct sk\_buff \*skb, struct cfg80211\_qos\_map \*qos\_map)

确定数据帧的802.1p / 1d标记

**参数**

struct sk\_buff \*skb

数据帧

struct cfg80211\_qos\_map \*qos\_map

互操作QoS映射或未使用时为NULL

**返回**

802.1p / 1d标记。

### 规定执行基础设施

待办事项

#### int regulatory\_hint(struct wiphy \*wiphy, const char \*alpha2)

向无线核心提供规制区域提示的驱动程序提示

**参数**

struct wiphy \*wiphy

发出提示的无线设备（仅用于报告冲突）

const char \*alpha2

驱动程序所在的ISO / IEC 3166 alpha2的规制区域。如果设置了rd，则应将其设置为NULL。注意，如果将其设置为NULL，则仍应将rd->alpha2设置为某个已接受的alpha2。

**说明**

无线驱动程序可以使用此函数向无线核心提示其认为应当前规制区域为其所在的ISO / IEC 3166 alpha2国家/地区代码，或通过提供完全构建的规制区域来完成此操作。如果驱动程序提供了ISO / IEC 3166 alpha2，则将查询用户空间以获取相应国家/地区的规制区域结构。

wiphy必须在此调用之前已注册到cfg80211。对于cfg80211驱动程序，这意味着您必须首先使用wiphy\_register（），对于mac80211驱动程序，您必须首先使用ieee80211\_register\_hw（）。

驱动程序应检查返回，有可能会收到-ENOMEM。

**返回**

成功时为0。-ENOMEM。

#### void wiphy\_apply\_custom\_regulatory(struct wiphy \*wiphy, const struct ieee80211\_regdomain \*regd)

应用自定义驱动程序规制区域

**参数**

struct wiphy \*wiphy

我们要在其中处理规制区域的无线设备

const struct ieee80211\_regdomain \*regd

要用于此wiphy的自定义规制区域

**说明**

驱动程序有时会具有不适用于特定国家/地区的自定义规制区域。驱动程序可以使用此功能应用此类自定义规制区域。必须在wiphy注册之前调用此程序。将完全信任自定义规制区域，因此将忽略先前的默认频道设置。如果在规定区域中未找到某个频道的规则，则禁用该频道。使用此项辅助功能的驱动程序应还将wiphy标志REGULATORY\_CUSTOM\_REG设置为cfg80211将为调用此辅助程序的wiphy设置该标志。

#### const struct ieee80211\_reg\_rule \*freq\_reg\_info(struct wiphy \*wiphy, u32 center\_freq)

获取给定频率的规制信息

**参数**

struct wiphy \*wiphy

我们要为其处理此规则的无线设备

u32 center\_freq

我们想要为其获取规制信息的频率（以KHz为单位）

**说明**

使用此函数获取给定无线设备中特定频率的规制规则。如果设备有特定的规制区域要遵循，则在接收和处理国家IE之前，我们将尊重该区域。

**返回**

有效指针，或者发生错误时，例如找不到规则，则返回使用ERR\_PTR（）进行编码。使用IS\_ERR（）来检查，并使用PTR\_ERR（）来获取数字返回。如果我们确定给定的center\_freq甚至没有中心频率带的频率范围的规制规则，则数值返回将为-ERANGE。请参见freq\_in\_rule\_band（）以了解我们当前对带的定义 - 这纯粹是主观的，现在它仅适用于802.11。

### RFkill集成

在cfg80211中的RFkill集成对驱动程序几乎是不可见的，因为cfg80211自动为每个它知道的无线设备注册了rfkill实例。软关断也会被转换为断开连接并关闭所有接口，当所有接口关闭时，驱动程序应该关闭设备。

但是，设备可能具有硬件RFkill线路，在这种情况下，它们还需要通过cfg80211与rfkill子系统进行交互。他们可以在这里使用一些辅助函数进行文档记录。

#### void wiphy\_rfkill\_set\_hw\_state(struct wiphy \*wiphy, bool blocked)

通知cfg80211有关硬件块状态

**参数**

struct wiphy \*wiphy

wiphy

bool blocked

块状态

#### void wiphy\_rfkill\_start\_polling(struct wiphy \*wiphy)

开始轮询RFkill

**参数**

struct wiphy \*wiphy

wiphy

void wiphy\_rfkill\_stop\_polling(struct wiphy \*wiphy)

停止轮询RFkill

**参数**

struct wiphy \*wiphy

wiphy

### 测试模式

测试模式是一组实用函数，允许驱动程序与驱动程序特定的工具进行交互，以帮助例如工厂编程。

本章介绍驱动程序如何与之交互，有关更多信息，请参见nl80211书籍中的章节。

#### struct sk\_buff \*cfg80211\_testmode\_alloc\_reply\_skb(struct wiphy \*wiphy, int approxlen)

分配测试模式回复

**参数**

struct wiphy \*wiphy

wiphy

int approxlen

将放入skb中的数据长度的上限

**说明**

此函数为测试模式命令的回复分配并预先填充skb。由于它旨在用于回复，因此在testmode\_cmd操作之外调用它是无效的。

返回的skb已预先填充了wiphy索引，并设置为以任何放入skb中的数据（使用skb\_put()、nla\_put()或类似方法）将最终位于NL80211\_ATTR\_TESTDATA属性中，因此只需向skb中添加相应的用户空间工具数据，然后该工具即可从testdata属性中读取该数据。您不应以任何其他方式修改skb。

完成后，使用skb调用cfg80211\_testmode\_reply()，并将其错误代码作为testmode\_cmd操作的结果返回。

**返回**

已分配和预先填充的skb。如果发生任何错误，则为NULL。

#### int cfg80211\_testmode\_reply(struct sk\_buff \*skb)

发送回复skb

**参数**

struct sk\_buff \*skb

skb，必须已使用cfg80211\_testmode\_alloc\_reply\_skb()分配

**说明**

由于调用此函数通常是从testmode\_cmd返回之前的最后一件事情，因此您应返回错误代码。请注意，该函数无论返回如何都会消耗skb。

**返回**

错误代码或0表示成功。

#### struct sk\_buff \*cfg80211\_testmode\_alloc\_event\_skb(struct wiphy \*wiphy, int approxlen, gfp\_t gfp)

分配测试模式事件

**参数**

struct wiphy \*wiphy

wiphy

int approxlen

将放入skb中的数据长度的上限

gfp\_t gfp

分配标志

**说明**

此函数为测试模式多播组中的事件分配并预先填充skb。

返回的skb与cfg80211\_testmode\_alloc\_reply\_skb()的方式相同，但为事件准备。与之相同，您只需向其中添加数据，然后该数据将最终位于NL80211\_ATTR\_TESTDATA属性中。同样，您不能以任何其他方式修改skb。

填充skb时，请使用skb调用cfg80211\_testmode\_event()以发送事件。

**返回**

已分配和预先填充的skb。如果发生任何错误，则为NULL。

#### void cfg80211\_testmode\_event(struct sk\_buff \*skb, gfp\_t gfp)

发送事件

**参数**

struct sk\_buff \*skb

skb，必须已使用cfg80211\_testmode\_alloc\_event\_skb()分配

gfp\_t gfp

分配标志

**说明**

此函数发送给定的skb，该skb必须由cfg80211\_testmode\_alloc\_event\_skb()分配作为一个事件。它总是会消耗它。

## mac80211子系统（基础知识）

在实现mac80211驱动程序时，您应该阅读并理解本书的这部分中包含的信息。在某些章节中，指出了高级用法，如果不需要，则可以跳过这些章节。

本书的此部分仅涵盖站点和监视器模式功能，实现其他模式所需的其他信息在本书的第二部分中涵盖。

### 基本硬件处理

TBD

此章节应该包含有关如何分配hw结构并将其注册到mac80211的信息。

由于需要在注册hw结构之前分配速率/模式，因此此章节还应包含有关设置速率/模式结构的信息。

此外，应在此处讨论有关回调和一般编程模型的问题，包括定义ieee80211\_ops，这将经常被引用。

最后，应该进行有关硬件功能的讨论，并涉及本书的其他部分的参考资料。

#### struct ieee80211\_hw

硬件信息和状态

**定义**

struct ieee80211\_hw {

struct ieee80211\_conf conf;

struct wiphy \*wiphy;

const char \*rate\_control\_algorithm;

void \*priv;

unsigned long flags[BITS\_TO\_LONGS(NUM\_IEEE80211\_HW\_FLAGS)];

unsigned int extra\_tx\_headroom;

unsigned int extra\_beacon\_tailroom;

int vif\_data\_size;

int sta\_data\_size;

int chanctx\_data\_size;

int txq\_data\_size;

u16 queues;

u16 max\_listen\_interval;

s8 max\_signal;

u8 max\_rates;

u8 max\_report\_rates;

u8 max\_rate\_tries;

u16 max\_rx\_aggregation\_subframes;

u16 max\_tx\_aggregation\_subframes;

u8 max\_tx\_fragments;

u8 offchannel\_tx\_hw\_queue;

u8 radiotap\_mcs\_details;

u16 radiotap\_vht\_details;

struct {

int units\_pos;

s16 accuracy;

} radiotap\_timestamp;

netdev\_features\_t netdev\_features;

u8 uapsd\_queues;

u8 uapsd\_max\_sp\_len;

u8 n\_cipher\_schemes;

const struct ieee80211\_cipher\_scheme \*cipher\_schemes;

u8 max\_nan\_de\_entries;

};

**成员**

conf

struct ieee80211\_conf，设备配置，请勿使用。

wiphy

此指向为此802.11 PHY分配的结构 wiphy。您必须使用 SET\_IEEE80211\_DEV() 和 SET\_IEEE80211\_PERM\_ADDR()  填写此结构的 perm\_addr 和 dev 成员。此外，所有支持的带（带通道、数据速率）也在此处注册。

rate\_control\_algorithm

此硬件的速率控制算法。如果未设置（为NULL），则将使用默认算法。必须在调用 ieee80211\_register\_hw() 之前设置。

priv

指向为驱动程序使用与此结构一起分配的私有区域的指针。

flags

硬件标志，请参见enum ieee80211\_hw\_flags。

extra\_tx\_headroom

每个传输skb中保留供驱动程序使用（例如用于传输标头）的头。

extra\_beacon\_tailroom

在每个beacon tx skb中保留的tailroom。驱动程序可以将其用于添加其他信息元素。

vif\_data\_size

在struct ieee80211\_vif中，drv\_priv数据区域的大小（以字节为单位）。

sta\_data\_size

在struct ieee80211\_sta中，drv\_priv数据区域的大小（以字节为单位）。

chanctx\_data\_size

在struct ieee80211\_chanctx\_conf中，drv\_priv数据区域的大小（以字节为单位）。

txq\_data\_size

在struct ieee80211\_txq中，drv\_priv数据区域的大小（以字节为单位）。

queues

发送数据包的可用硬件传输队列数量。 WMM/QoS需要至少四个，这些队列需要具有可配置的访问参数。

max\_listen\_interval

硬件支持的最大侦听间隔，单位为信标间隔。

max\_signal

接收信息中信号（RSSI）的最大值，仅在IEEE80211\_HW\_SIGNAL\_UNSPEC或IEEE80211\_HW\_SIGNAL\_DB时使用。

max\_rates

硬件可以处理的备用速率重试阶段的最大数量。

max\_report\_rates

硬件可以报告的备用速率重试阶段的最大数量。

max\_rate\_tries

每个阶段的最大尝试次数。

max\_rx\_aggregation\_subframes

用于A-MPDU块ACK接收器聚合的最大缓冲区大小（子帧数）。只有设备对子帧数有限制，如果它依赖于mac80211进行重新排序，则不应设置它。

max\_tx\_aggregation\_subframes

HT/HE设备在聚合中传输的最大子帧数。在HT AddBA中，我们将广告一个常量值64，因为一些较旧的AP在窗口大小较小时会崩溃（例如LinkSys WRT120N，固件版本为v1.0.07构建002，2012年6月18日）。对于AddBA到HE能力的对等点，将使用此值。

max\_tx\_fragments

(A)-MSDU的最大tx缓冲区数，对于frag\_list中的每个skb，1 + skb\_shinfo(skb)->nr\_frags的总和。

offchannel\_tx\_hw\_queue

用于离频TX的HW队列ID（如果设置了IEEE80211\_HW\_QUEUE\_CONTROL）。

radiotap\_mcs\_details

列出HW可以报告的哪些MCS信息，默认设置为\_MCS、\_GI和\_BW，但不包括\_FMT。使用IEEE80211\_RADIOTAP\_MCS\_HAVE \_\*值，今天只支持添加\_BW。

radiotap\_vht\_details

列出HW报告的哪些VHT MCS信息，默认为\_GI|\_BANDWIDTH。使用IEEE80211\_RADIOTAP\_VHT\_KNOWN \_\*值。

radiotap\_timestamp

radiotap时间戳字段的信息；如果“units\_pos”成员设置为非负值，则必须将其设置为IEEE80211\_RADIOTAP\_TIMESTAMP\_UNIT\_\*和IEEE80211\_RADIOTAP\_TIMESTAMP\_SPOS\_\*值的组合，然后从struct ieee80211\_rx\_status的device\_timestamp添加并填充时间戳字段。如果“accuracy”成员是非负的，则将其放入精度radiotap字段中，并设置精度已知标志。

netdev\_features

在从此HW创建的每个netdev中设置的netdev功能。请注意，并非所有功能都可与mac80211一起使用，其他功能将在HW注册期间被拒绝。

uapsd\_queues

此位图包含在（重新）关联帧中，以指示每个访问类别是否启用了uAPSD触发器并启用了传递。使用IEEE80211\_WMM\_IE\_STA\_QOSINFO\_AC\_\*设置此位图。每位对应于不同的AC。特定位中的值“1”表示相应的AC既启用了触发器又启用了传递。‘0’表示都未启用。

uapsd\_max\_sp\_len

WMM AP在由WMM STA触发的任何服务期间内可能向WMM STA传递的缓冲帧的最大数量。使用IEEE80211\_WMM\_IE\_STA\_QOSINFO\_SP\_\*设置正确的值。

n\_cipher\_schemes

密码方案定义数组的大小。

cipher\_schemes

指向HW支持的密码方案定义数组的指针。

max\_nan\_de\_entries

设备支持的NAN DE函数的最大数量。

**说明**

此结构包含802.11 PHY的配置和硬件信息。

#### enum ieee80211\_hw\_flags

硬件标志

**常量**

IEEE80211\_HW\_HAS\_RATE\_CONTROL

硬件或固件包括速率控制，并且无法由堆栈控制。因此，不应实例化速率控制算法，并且将从TX状态而非速率控制算法中获取报告给用户空间的TX速率。请注意，这需要驱动程序实施许多回调，以便它具有正确的信息，它需要具有set\_rts\_threshold回调，并且必须查看BSS配置use\_cts\_prot用于G/N保护，use\_short\_slot用于2.4 GHz的时隙定时和use\_short\_preamble用于CCK帧的前导码。

IEEE80211\_HW\_RX\_INCLUDES\_FCS

表示传递到堆栈的接收帧包括结尾处的FCS。

IEEE80211\_HW\_HOST\_BROADCAST\_PS\_BUFFERING

某些无线LAN芯片组在硬件/固件中缓冲广播/多播帧以用于节能站点，而其他芯片组则依赖于主机系统来进行此类缓冲。此选项用于配置IEEE 802.11上层以在有省电站点时缓冲广播和多播帧，以便驱动程序可以使用ieee80211\_get\_buffered\_bc()获取它们。

IEEE80211\_HW\_SIGNAL\_UNSPEC

硬件可以提供信号值，但我们不知道其单位。我们期望值在0到max\_signal之间。如果可能，请提供dB或dBm。

IEEE80211\_HW\_SIGNAL\_DBM

硬件以dBm为单位提供信号值，即与一毫瓦之差的分贝。这是首选方法，因为它在不同设备之间是标准化的。不需要设置max\_signal。

IEEE80211\_HW\_NEED\_DTIM\_BEFORE\_ASSOC

此设备需要在关联前（即dtim\_period）从信标获取数据。

IEEE80211\_HW\_SPECTRUM\_MGMT

硬件支持802.11h测量、频道切换、安静和TPC中定义的频谱管理。

IEEE80211\_HW\_AMPDU\_AGGREGATION

硬件支持11n A-MPDU聚合。

IEEE80211\_HW\_SUPPORTS\_PS

硬件支持电源保存（即可进入睡眠状态）。

IEEE80211\_HW\_PS\_NULLFUNC\_STACK

硬件要求堆栈中使用nullfunc帧处理，意味着支持动态PS的堆栈。

IEEE80211\_HW\_SUPPORTS\_DYNAMIC\_PS

硬件支持动态PS。

IEEE80211\_HW\_MFP\_CAPABLE

硬件支持管理帧保护（MFP，IEEE 802.11w）。

IEEE80211\_HW\_WANT\_MONITOR\_VIF

驱动程序希望在监视器接口是唯一活动接口时得到虚拟监视器接口的通知。

IEEE80211\_HW\_NO\_AUTO\_VIF

驱动程序希望不创建wlanX。预计用户空间将根据需要创建虚拟接口（因此具有所需的名称）。

IEEE80211\_HW\_SW\_CRYPTO\_CONTROL

驱动程序希望控制可以在软件中完成的加密算法 - 因此不要在硬件加密失败时自动尝试回退，但仅在驱动程序返回1时这样做。这也强制驱动程序宣传其支持的密码套件。

IEEE80211\_HW\_SUPPORT\_FAST\_XMIT

驱动程序/硬件支持快速传输，目前仅需要能够计算帧的持续时间。

IEEE80211\_HW\_REPORTS\_TX\_ACK\_STATUS

硬件可以向堆栈提供发送帧的ACK状态报告。

IEEE80211\_HW\_CONNECTION\_MONITOR

硬件执行其自己的连接监视，包括定期保持向AP和在信标丢失时探测AP。

IEEE80211\_HW\_QUEUE\_CONTROL

驱动程序希望控制每个接口的队列映射，以便为不同的虚拟接口使用不同的队列（不仅仅是每个AC一个）。有关更多详细信息，请参见有关HW队列控制的文档部分。

IEEE80211\_HW\_SUPPORTS\_PER\_STA\_GTK

设备的加密引擎支持IBSS RSN或快速转换期间使用的每个站GTK。如果设备不支持每个站GTK，但可以要求不解密组地址帧，则仍然可以使用IBSS RSN支持，但将使用软件加密。仅在那种情况下广告wiphy标志。

IEEE80211\_HW\_AP\_LINK\_PS

在AP模式下操作时，设备自主管理连接的站的PS状态。当设置了此标志时，mac80211不会基于传入帧的PM位触发连接的站的PS模式。使用ieee80211\_start\_ps()/ieee8021\_end\_ps()手动配置连接的站的PS模式。

IEEE80211\_HW\_TX\_AMPDU\_SETUP\_IN\_HW

设备严格通过HW处理TX A-MPDU会话设置。mac80211不应尝试在软件中执行此操作。

IEEE80211\_HW\_SUPPORTS\_RC\_TABLE

驱动程序支持使用速率选择算法提供的速率选择表。

IEEE80211\_HW\_P2P\_DEV\_ADDR\_FOR\_INTF

对于任何P2P界面，请使用P2P设备地址。即使支持多个接口也将受到此服务。

IEEE80211\_HW\_TIMING\_BEACON\_ONLY

仅从信标帧使用同步定时，以允许获取DTIM信标的TBTT。

IEEE80211\_HW\_SUPPORTS\_HT\_CCK\_RATES

硬件支持混合HT/CCK速率并且可以处理聚合会话中的CCK速率（例如通过不为此类帧使用聚合来实现）。

IEEE80211\_HW\_CHANCTX\_STA\_CSA

使用通道上下文支持基于802.11h的通道切换（CSA），为单个活动通道提供。当未启用支持时，默认操作为在获取CSA帧时断开连接。

IEEE80211\_HW\_SUPPORTS\_CLONED\_SKBS

驱动程序永远不会在未复制它们的情况下修改TX skbs的有效负载或尾部空间。

IEEE80211\_HW\_SINGLE\_SCAN\_ON\_ALL\_BANDS

硬件支持一次扫描所有频段，mac80211无需针对每个频段运行单独的扫描。

IEEE80211\_HW\_TDLS\_WIDER\_BW

设备/驱动程序支持TDLS链路上比BSS带宽更宽的带宽。

IEEE80211\_HW\_SUPPORTS\_AMSDU\_IN\_AMPDU

驱动程序支持在A-MPDU中接收A-MSDU。

IEEE80211\_HW\_BEACON\_TX\_STATUS

设备/驱动程序提供已发送信标的TX状态。

IEEE80211\_HW\_NEEDS\_UNIQUE\_STA\_ADDR

硬件（或驱动程序）要求每个站具有唯一地址，即每个站条目可以仅通过其MAC地址进行标识；例如，这可以防止同一站在同时连接两个虚拟AP接口。

IEEE80211\_HW\_SUPPORTS\_REORDERING\_BUFFER

硬件（或驱动程序）在内部管理重排缓冲区，确保mac80211按顺序接收帧，并且不需要管理自己的重排缓冲区或BA会话超时。

IEEE80211\_HW\_USES\_RSS

设备使用RSS，因此需要并行RX，这意味着使用每个CPU站点统计信息。

IEEE80211\_HW\_TX\_AMSDU

硬件（或驱动程序）支持软件聚合A-MSDU帧。需要软件TX队列和快速传输支持。当不使用minstrel/minstrel\_ht速率控制时，驱动程序必须根据当前TX速率通过设置struct ieee80211\_sta中的max\_rc\_amsdu\_len来限制最大A-MSDU大小。

IEEE80211\_HW\_TX\_FRAG\_LIST

硬件（或驱动程序）支持发送frag\_list skbs，用于零拷贝软件A-MSDU。

IEEE80211\_HW\_REPORTS\_LOW\_ACK

驱动程序（或固件）通过基于自己的算法的ieee80211\_report\_low\_ack()报告低ACK事件。对于这样的驱动程序，mac80211数据包丢失机制不会被触发，并且驱动程序完全依赖固件事件用于强制站点退出。

IEEE80211\_HW\_SUPPORTS\_TX\_FRAG

硬件本身进行分片。栈不会进行分片。还应该设置set\_frag\_threshold回调函数。

IEEE80211\_HW\_SUPPORTS\_TDLS\_BUFFER\_STA

硬件支持TDLS链路上的缓冲区STA。

IEEE80211\_HW\_DEAUTH\_NEED\_MGD\_TX\_PREP

驱动程序要求在发出认证完成但未收到信标时，在发送解除身份验证帧之前调用mgd\_prepare\_tx()回调。在多通道场景中需要这样做，其中虚拟接口可能不会获得传输帧的空间时间，因为它尚未与AP/P2P GO同步，因此解除身份验证帧可能不会传输。

IEEE80211\_HW\_DOESNT\_SUPPORT\_QOS\_NDP

驱动程序(或固件)不支持用于AP探测的QoS NDP - 这很可能是驱动程序错误。

NUM\_IEEE80211\_HW\_FLAGS

硬件标志的数量，用于调整数组大小

**说明**

这些标志用于指示硬件的功能。一般来说，这里的标志应该按照最简单的硬件不需要设置任何特定的标志的方式来完成含义。然而，有一些例外，因此建议仔细查看这些标志。

#### **voi**d SET\_IEEE80211\_DEV(struct [ieee80211\_hw](https://www.kernel.org/doc/html/latest/driver-api/80211/mac80211.html" \l "c.ieee80211_hw" \o "ieee80211_hw) \*hw, struct [device](https://www.kernel.org/doc/html/latest/driver-api/infrastructure.html" \l "c.device" \o "device) \*dev)

设置802.11硬件的设备

**参数**

struct ieee80211\_hw \*hw

设置设备的struct ieee80211\_hw

struct device \*dev

这个802.11设备的设备结构

#### **void** SET\_IEEE80211\_PERM\_ADDR(struct ieee80211\_hw \*hw, const u8 \*addr)

设置802.11硬件的永久MAC地址

**参数**

struct ieee80211\_hw \*hw

用于设置MAC地址的struct ieee80211\_hw

const u8 \*addr

要设置的地址

#### struct ieee80211\_ops

来自mac80211到驱动程序的回调

**定义**

struct ieee80211\_ops {

void (\*tx)(struct ieee80211\_hw \*hw,struct ieee80211\_tx\_control \*control, struct sk\_buff \*skb);

int (\*start)(struct ieee80211\_hw \*hw);

void (\*stop)(struct ieee80211\_hw \*hw);

#ifdef CONFIG\_PM;

int (\*suspend)(struct ieee80211\_hw \*hw, struct cfg80211\_wowlan \*wowlan);

int (\*resume)(struct ieee80211\_hw \*hw);

void (\*set\_wakeup)(struct ieee80211\_hw \*hw, bool enabled);

#endif;

int (\*add\_interface)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif);

int (\*change\_interface)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, enum nl80211\_iftype new\_type, bool p2p);

void (\*remove\_interface)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif);

int (\*config)(struct ieee80211\_hw \*hw, u32 changed);

void (\*bss\_info\_changed)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif,struct ieee80211\_bss\_conf \*info, u32 changed);

int (\*start\_ap)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif);

void (\*stop\_ap)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif);

u64 (\*prepare\_multicast)(struct ieee80211\_hw \*hw, struct netdev\_hw\_addr\_list \*mc\_list);

void (\*configure\_filter)(struct ieee80211\_hw \*hw,unsigned int changed\_flags,unsigned int \*total\_flags, u64 multicast);

void (\*config\_iface\_filter)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif,unsigned int filter\_flags, unsigned int changed\_flags);

int (\*set\_tim)(struct ieee80211\_hw \*hw, struct ieee80211\_sta \*sta, bool set);

int (\*set\_key)(struct ieee80211\_hw \*hw, enum set\_key\_cmd cmd,struct ieee80211\_vif \*vif, struct ieee80211\_sta \*sta, struct ieee80211\_key\_conf \*key);

void (\*update\_tkip\_key)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif,struct ieee80211\_key\_conf \*conf,struct ieee80211\_sta \*sta, u32 iv32, u16 \*phase1key);

void (\*set\_rekey\_data)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, struct cfg80211\_gtk\_rekey\_data \*data);

void (\*set\_default\_unicast\_key)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif, int idx);

int (\*hw\_scan)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif, struct ieee80211\_scan\_request \*req);

void (\*cancel\_hw\_scan)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif);

int (\*sched\_scan\_start)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif,struct cfg80211\_sched\_scan\_request \*req, struct ieee80211\_scan\_ies \*ies);

int (\*sched\_scan\_stop)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif);

void (\*sw\_scan\_start)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, const u8 \*mac\_addr);

void (\*sw\_scan\_complete)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif);

int (\*get\_stats)(struct ieee80211\_hw \*hw, struct ieee80211\_low\_level\_stats \*stats);

void (\*get\_key\_seq)(struct ieee80211\_hw \*hw,struct ieee80211\_key\_conf \*key, struct ieee80211\_key\_seq \*seq);

int (\*set\_frag\_threshold)(struct ieee80211\_hw \*hw, u32 value);

int (\*set\_rts\_threshold)(struct ieee80211\_hw \*hw, u32 value);

int (\*sta\_add)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif, struct ieee80211\_sta \*sta);

int (\*sta\_remove)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif, struct ieee80211\_sta \*sta);

#ifdef CONFIG\_MAC80211\_DEBUGFS;

void (\*sta\_add\_debugfs)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif,struct ieee80211\_sta \*sta, struct dentry \*dir);

#endif;

void (\*sta\_notify)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif, enum sta\_notify\_cmd, struct ieee80211\_sta \*sta);

int (\*sta\_state)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif,struct ieee80211\_sta \*sta,enum ieee80211\_sta\_state old\_state, enum ieee80211\_sta\_state new\_state);

void (\*sta\_pre\_rcu\_remove)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, struct ieee80211\_sta \*sta);

void (\*sta\_rc\_update)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif,struct ieee80211\_sta \*sta, u32 changed);

void (\*sta\_rate\_tbl\_update)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, struct ieee80211\_sta \*sta);

void (\*sta\_statistics)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif,struct ieee80211\_sta \*sta, struct station\_info \*sinfo);

int (\*conf\_tx)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, u16 ac, const struct ieee80211\_tx\_queue\_params \*params);

u64 (\*get\_tsf)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif);

void (\*set\_tsf)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif, u64 tsf);

void (\*offset\_tsf)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif, s64 offset);

void (\*reset\_tsf)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif);

int (\*tx\_last\_beacon)(struct ieee80211\_hw \*hw);

int (\*ampdu\_action)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, struct ieee80211\_ampdu\_params \*params);

int (\*get\_survey)(struct ieee80211\_hw \*hw, int idx, struct survey\_info \*survey);

void (\*rfkill\_poll)(struct ieee80211\_hw \*hw);

void (\*set\_coverage\_class)(struct ieee80211\_hw \*hw, s16 coverage\_class);

#ifdef CONFIG\_NL80211\_TESTMODE;

int (\*testmode\_cmd)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif, void \*data, int len);

int (\*testmode\_dump)(struct ieee80211\_hw \*hw, struct sk\_buff \*skb,struct netlink\_callback \*cb, void \*data, int len);

#endif;

void (\*flush)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif, u32 queues, bool drop);

void (\*channel\_switch)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, struct ieee80211\_channel\_switch \*ch\_switch);

int (\*set\_antenna)(struct ieee80211\_hw \*hw, u32 tx\_ant, u32 rx\_ant);

int (\*get\_antenna)(struct ieee80211\_hw \*hw, u32 \*tx\_ant, u32 \*rx\_ant);

int (\*remain\_on\_channel)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif,struct ieee80211\_channel \*chan,int duration, enum ieee80211\_roc\_type type);

int (\*cancel\_remain\_on\_channel)(struct ieee80211\_hw \*hw);

int (\*set\_ringparam)(struct ieee80211\_hw \*hw, u32 tx, u32 rx);

void (\*get\_ringparam)(struct ieee80211\_hw \*hw, u32 \*tx, u32 \*tx\_max, u32 \*rx, u32 \*rx\_max);

bool (\*tx\_frames\_pending)(struct ieee80211\_hw \*hw);

int (\*set\_bitrate\_mask)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif, const struct cfg80211\_bitrate\_mask \*mask);

void (\*event\_callback)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, const struct ieee80211\_event \*event);

void (\*allow\_buffered\_frames)(struct ieee80211\_hw \*hw,struct ieee80211\_sta \*sta,u16 tids, int num\_frames,enum ieee80211\_frame\_release\_type reason, bool more\_data);

void (\*release\_buffered\_frames)(struct ieee80211\_hw \*hw,struct ieee80211\_sta \*sta,u16 tids, int num\_frames,enum ieee80211\_frame\_release\_type reason, bool more\_data);

int (\*get\_et\_sset\_count)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif, int sset);

void (\*get\_et\_stats)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, struct ethtool\_stats \*stats, u64 \*data);

void (\*get\_et\_strings)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, u32 sset, u8 \*data);

void (\*mgd\_prepare\_tx)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, u16 duration);

void (\*mgd\_protect\_tdls\_discover)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif);

int (\*add\_chanctx)(struct ieee80211\_hw \*hw, struct ieee80211\_chanctx\_conf \*ctx);

void (\*remove\_chanctx)(struct ieee80211\_hw \*hw, struct ieee80211\_chanctx\_conf \*ctx);

void (\*change\_chanctx)(struct ieee80211\_hw \*hw,struct ieee80211\_chanctx\_conf \*ctx, u32 changed);

int (\*assign\_vif\_chanctx)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, struct ieee80211\_chanctx\_conf \*ctx);

void (\*unassign\_vif\_chanctx)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, struct ieee80211\_chanctx\_conf \*ctx);

int (\*switch\_vif\_chanctx)(struct ieee80211\_hw \*hw,struct ieee80211\_vif\_chanctx\_switch \*vifs,int n\_vifs, enum ieee80211\_chanctx\_switch\_mode mode);

void (\*reconfig\_complete)(struct ieee80211\_hw \*hw, enum ieee80211\_reconfig\_type reconfig\_type);

#if IS\_ENABLED(CONFIG\_IPV6);

void (\*ipv6\_addr\_change)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, struct inet6\_dev \*idev);

#endif;

void (\*channel\_switch\_beacon)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, struct cfg80211\_chan\_def \*chandef);

int (\*pre\_channel\_switch)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, struct ieee80211\_channel\_switch \*ch\_switch);

int (\*post\_channel\_switch)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif);

int (\*join\_ibss)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif);

void (\*leave\_ibss)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif);

u32 (\*get\_expected\_throughput)(struct ieee80211\_hw \*hw, struct ieee80211\_sta \*sta);

int (\*get\_txpower)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif, int \*dbm);

int (\*tdls\_channel\_switch)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif,struct ieee80211\_sta \*sta, u8 oper\_class,struct cfg80211\_chan\_def \*chandef, struct sk\_buff \*tmpl\_skb, u32 ch\_sw\_tm\_ie);

void (\*tdls\_cancel\_channel\_switch)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, struct ieee80211\_sta \*sta);

void (\*tdls\_recv\_channel\_switch)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, struct ieee80211\_tdls\_ch\_sw\_params \*params);

void (\*wake\_tx\_queue)(struct ieee80211\_hw \*hw, struct ieee80211\_txq \*txq);

void (\*sync\_rx\_queues)(struct ieee80211\_hw \*hw);

int (\*start\_nan)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, struct cfg80211\_nan\_conf \*conf);

int (\*stop\_nan)(struct ieee80211\_hw \*hw, struct ieee80211\_vif \*vif);

int (\*nan\_change\_conf)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, struct cfg80211\_nan\_conf \*conf, u32 changes);

int (\*add\_nan\_func)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, const struct cfg80211\_nan\_func \*nan\_func);

void (\*del\_nan\_func)(struct ieee80211\_hw \*hw,struct ieee80211\_vif \*vif, u8 instance\_id);

};

**成员**

tx

处理器，802.11 模块在每个已传输帧时调用该处理器。skb 包含从 IEEE 802.11 头开始的缓冲区。底层驱动程序应该根据 TX 控制数据中的配置发送帧。此处理器最好永远不会失败并适当停止队列。必须是原子操作。

start

在启用连接到硬件的第一个 netdevice 之前调用。这应该打开硬件并必须打开帧接收（可能已启用监视器界面）。返回负错误代码，这些可能在用户空间中看到，或者为零。当设备启动时，它不应该有 MAC 地址，以避免在添加非监视器设备之前确认帧。必须实现并且可以睡眠。

stop

在停用连接到硬件的最后一个 netdevice 后调用。这应该关闭硬件（至少必须关闭帧接收）。可能在添加接口拒绝接口后立即调用。如果您在 mac80211 工作队列中添加了任何工作，您应该确保在回调上取消它。必须实现并且可以睡眠。

suspend

挂起设备；mac80211 本身将静止前后操作并停止传输和执行任何其他配置，然后要求设备挂起。仅在配置了 WoWLAN 时才调用此函数，否则设备将完全被取消配置，并在恢复时重新配置。驱动程序还可以施加一些特殊条件，以在其想要使用“正常”暂停（取消配置）下使用 WoWLAN，例如如果它仅在设备关联时支持 WoWLAN。在这种情况下，它必须从此函数返回 1。

resume

如果配置了 WoWLAN，则表示 mac80211 现在正在恢复其操作，之后设备必须再次完全正常运行。如果返回错误，则唯一的出路是还注销设备。如果返回 1，则 mac80211 也会在恢复上进行定期的完全重新启动。

set\_wakeup

在修改WoWLAN配置时启用或禁用唤醒。原因是当配置发生变化时，应调用device\_set\_wakeup\_enable()，而不仅仅在挂起时调用。

add\_interface

当附加到硬件的netdevice被启用时调用。因为对于监视模式设备，不会调用它，所以必须实现start和stop。驱动程序在启用设备之前应执行任何必要的初始化。接口的初始配置以conf参数的形式给出。通过返回负错误代码（将在用户空间中看到）拒绝添加接口。必须实现，并且可能会睡眠。

change\_interface

当Netdevice更改类型时调用。此回调函数是可选的，但仅在支持此回调函数时才能在接口UP时切换接口类型。回调可能会睡眠。请注意，在接口被切换时，它将无法被接口迭代回调找到。

remove\_interface

通知驱动程序接口正在关闭。如果是最后一个接口且不存在监视接口，则调用stop回调。当所有接口都被删除时，必须清除硬件中的MAC地址，以便设备不再确认数据包，但是conf结构的mac\_addr成员设置为即将离开的设备的MAC地址。因此，必须实现此回调函数。它可以睡眠。

config

用于配置请求的处理程序。IEEE 802.11代码调用此函数以更改硬件配置，例如通道。此函数不应失败，但如果它失败，则返回负错误代码。回调可能会睡眠。

bss\_info\_changed

处理与BSS参数相关的配置请求，这些参数可能会在BSS的生命周期中变化，并可能影响低级驱动程序（例如，关联/分离状态、erp参数）。除非用于关联通知，否则不应使用此函数，如果没有设置BSS。当进行调用时，changed参数指示已更改bss参数中的哪个，回调函数可能会睡眠。

start\_ap

在设置bss\_conf中的所有信息并可以检索到beacon之后，在AP接口上启动操作。在调用此函数之前绑定频道上下文。请注意，如果驱动程序使用软件扫描或ROC，则在AP仅为扫描/ROC而“暂停”时（通过禁用/启用信标来指示），不会调用此函数（和stop\_ap）。

stop\_ap

停止AP接口上的操作。

prepare\_multicast

为多播过滤器配置做准备。此回调函数是可选的，并且其返回由configure\_filter()传递。此回调必须是原子的。

configure\_filter

配置设备的RX过滤器。有关更多信息，请参见“Frame filtering”部分。此回调必须得到实现，并且可以睡眠。

config\_iface\_filter

配置接口的RX过滤器。此回调是可选的，并用于配置应将哪些帧传递给mac80211。filter\_flags是FIF\_\*标志的组合。changed\_flags是一个位掩码，用于指示哪些标志已更改。此回调可以睡眠。

set\_tim

设置TIM位。当必须为给定STA设置或清除TIM位时，mac80211调用此函数。必须是原子的。

set\_key

请参见“Hardware crypto acceleration”部分。仅在add\_interface和remove\_interface之间调用此回调函数，即在启用给定虚拟接口时。如果无法添加密钥，则返回负错误代码。回调可能会睡眠。

update\_tkip\_key

请参见“Hardware crypto acceleration”部分。此回调将在Rx的上下文中被调用。对于设置IEEE80211\_KEY\_FLAG\_TKIP\_REQ\_RX\_P1\_KEY的驱动程序进行调用。此回调必须是原子的。

set\_rekey\_data

如果设备支持GTK重新加密，例如当主机处于挂起状态时，它可以分配此回调以检索执行GTK重新加密所需的数据，这是KEK、KCK和重播计数器。完成重新加密后，它应在恢复期间（例如）通过ieee80211\_gtk\_rekey\_notify()通知用户空间新的重播计数器。

set\_default\_unicast\_key

设置默认（单播）密钥索引，对WEP很有用，当设备自主发送数据包时，例如用于ARP卸载。索引可以是0-3，或者是-1表示取消设置它。

hw\_scan

请求硬件服务扫描请求，无需在堆栈中启动扫描状态机。扫描必须遵守Wiphy在注册波段中所做的频道配置。硬件（或驱动程序）需要确保已禁用电源保存。mac80211将req ie/ie\_len成员重写为在SSID之后包含整个IE的内容，因此驱动程序无需完全查看这些内容，只需在SSID之后发送它们--mac80211包括（扩展）受支持的速率和HT信息（如果适用）。当扫描完成时，必须调用ieee80211\_scan\_completed()。请注意，除非此回调返回负错误代码，否则必须在任何错误情况下调用它，以便扫描无法完成。此回调还允许返回特殊的返回1，这表示现在不需要进行硬件扫描，而应进行软件扫描。希望使用此能力的驱动程序必须确保其（硬件）扫描功能不被广告为比mac80211的软件扫描更具能力。回调可能会睡眠。

cancel\_hw\_scan

请求低级 tp 取消活跃的硬件扫描。驱动程序应该请求硬件取消扫描（如果可能），但是扫描只有在驱动程序调用 ieee80211\_scan\_completed()后才会完成。这个回调函数是为了 wowlan 的使用，以防止在低级驱动程序已经挂起之后，再次将新的扫描工作排队。这个回调函数可以进行睡眠。

sched\_scan\_start

请求硬件以特定间隔重复开始扫描。每当驱动程序找到结果时，必须调用 ieee80211\_sched\_scan\_results() 函数。这个过程将继续执行，直到调用 sched\_scan\_stop 停止。

sched\_scan\_stop

告诉硬件停止正在进行的定期扫描。在这种情况下，不能调用 ieee80211\_sched\_scan\_stopped()。

sw\_scan\_start

在软件扫描开始之前调用的通知函数。如果驱动程序不需要这个通知，则可以为 NULL。mac\_addr 参数允许支持 NL80211\_SCAN\_FLAG\_RANDOM\_ADDR，如果驱动程序可以使用这个参数，则可以设置 NL80211\_FEATURE\_SCAN\_RANDOM\_MAC\_ADDR 标志。这个回调函数可以进行睡眠。

sw\_scan\_complete

在软件扫描完成后调用的通知函数。如果驱动程序不需要这个通知，则可以为 NULL。这个回调函数可以进行睡眠。

get\_stats

返回低级统计信息。如果有统计信息，则返回零。这个回调函数可以进行睡眠。

get\_key\_seq

如果您的设备在硬件中实现加密并进行 IV/PN 分配，则应提供此回调函数以从硬件中读取给定密钥的 IV/PN。这个回调函数必须是原子的。

set\_frag\_threshold

配置分片阈值。如果设备自己进行分片，则分配它。请注意，为了防止堆栈进行分片，应设置 IEEE80211\_HW\_SUPPORTS\_TX\_FRAG。这个回调函数可以进行睡眠。

set\_rts\_threshold

配置 RTS 阈值（如果设备需要）。这个回调函数可以进行睡眠。

sta\_add

通知低级驱动程序有一个已关联的 station、AP、IBSS/WDS/mesh peer 等的添加。这个回调函数可以进行睡眠。

sta\_remove

通知低级驱动程序有一个已关联的 station、AP、IBSS/WDS/mesh peer 等的移除。请注意，在回调函数返回之后，即使是 RCU 保护的指针也不能安全使用；在返回此处和释放站点之间没有 RCU 宽限期。如果需要，可以查看 sta\_pre\_rcu\_remove。这个回调函数可以进行睡眠。

sta\_add\_debugfs

驱动程序可以使用这个回调函数在将一个站点添加到 mac80211 的站点列表时添加 debugfs 文件。这个回调函数应该在 CONFIG\_MAC80211\_DEBUGFS 条件中。这个回调函数可以进行睡眠。

sta\_notify

通知低级驱动程序一个关联站点、AP、IBSS/WDS/mesh peer 等的电源状态转换。对于以 AP 模式操作的 VIF，当设置 IEEE80211\_HW\_AP\_LINK\_PS 标志时，将不会调用此回调函数。必须是原子的。

sta\_state

通知低级驱动程序一个站点（可以是 AP、客户端、IBSS/WDS/mesh peer 等）的状态转换。这个回调函数与 sta\_add/sta\_remove 互斥。它不能对下降转换失败，但可能对状态列表上升的转换失败。还要注意，在回调函数返回之后，即使是 RCU 保护的指针也不能安全使用 - 在返回此处和释放站点之间没有 RCU 宽限期。如果需要，可以查看 sta\_pre\_rcu\_remove。这个回调函数可以进行睡眠。

sta\_pre\_rcu\_remove

在 RCU 同步之前通知驱动程序有站点移除。如果驱动程序需要使用 RCU 保护站点指针，则可以使用此调用清除指针，而不是等待在 sta\_state 中等待 RCU 宽限期过去。这个回调函数可以进行睡眠。

sta\_rc\_update

通知驱动程序正在使用哪些速率向站点进行传输的变化。变化的广告位来自枚举 ieee80211\_rate\_control\_changed，数值反映在数据站点中。如果驱动程序使用硬件速率控制 (IEEE80211\_HW\_HAS\_RATE\_CONTROL)，则应该仅使用这个回调函数，否则速率控制算法直接通知。必须是原子的。

sta\_rate\_tbl\_update

通知驱动程序速率表已经更改。仅当配置的速率控制算法实际使用新的速率表 API 时才使用此函数，因此是可选的。必须是原子的。

sta\_statistics

获取此站点的统计信息。例如，在信标过滤中，mac80211 保持的统计信息可能不准确，因此让驱动程序填充统计信息。驱动程序可以填充大部分值（通过设置填充位图表示），但并非所有值都是有意义的-请查看源代码中哪些值是可能的。驱动程序不填充的统计信息将由 mac80211 填充。这个回调函数可以进行睡眠。

conf\_tx

为硬件 TX 队列配置 TX 队列参数（EDCF（aifs、cw\_min、cw\_max）、bursting）。失败返回负错误代码。这个回调函数可以进行睡眠。

get\_tsf

从固件/硬件获取当前 TSF 计时器值。目前，这仅用于 IBSS 模式 BSSID 合并和调试。不是必需的函数。这个回调函数可以进行睡眠。

set\_tsf

将TSF计时器设置为固件/硬件中指定的值。目前，这只用于IBSS模式调试。不是一个必需的功能。回调函数可以休眠。

offset\_tsf

将TSF计时器偏移指定值在固件/硬件中。优选set\_tsf，因为它避免了调用set\_tsf()和硬件编程之间的延迟，这将显示为TSF延迟。不是一个必需的功能。回调函数可以休眠。

reset\_tsf

复位TSF计时器，并允许固件/硬件与IBSS中的其他STA同步。这仅用于IBSS模式。如果固件/硬件完全负责TSF同步，则此函数是可选的。回调函数可以休眠。

tx\_last\_beacon

确定上一个IBSS信标是否是我们发送的。这仅针对IBSS模式，并且此函数的结果用于确定是否回复探测请求。如果设备发送了上一次信标，则返回非零值。回调函数可以休眠。

ampdu\_action

执行某个A-MPDU操作。RA / TID组合确定我们希望执行ampdu操作的目的地和TID。操作定义为ieee80211\_ampdu\_mlme\_action。当操作设置为IEEE80211\_AMPDU\_TX\_OPERATIONAL时，驱动程序既不能发送包含超过buf\_size个子帧的聚合，也不能以超过缓冲区大小的方式发送聚合以导致丢失的帧超过缓冲区大小。如果仅限制聚合大小，则可以使用buf\_size为8：

TX：1.....7

RX：2....7（丢失帧＃1）

TX：8..1...

这是无效的，因为＃1现在重新传输，远远超过了8的缓冲区大小。重新传输＃1的正确方法是：

TX：1或

TX：18或

TX：81

即使是189也是错误的，因为1可能再次丢失。

发生故障时返回负错误代码。如果会话可以立即开始，则驱动程序可以返回IEEE80211\_AMPDU\_TX\_START的IEEE80211\_AMPDU\_TX\_START\_IMMEDIATE。

get\_survey

返回每个信道的调查信息

rfkill\_poll

轮询rfkill硬件状态。如果需要此项，您还需要在注册之前将wiphy->rfkill\_poll设置为true，并需要在回调中调用wiphy\_rfkill\_set\_hw\_state()。回调函数可以休眠。

set\_coverage\_class

根据IEEE 802.11-2007第17.3.8.6节设置给定覆盖类别的时隙时间，并相应地修改ACK超时；覆盖类别等于-1以启用ACK超时估计算法（dynack）。要禁用dynack，请为覆盖类设置有效值。此回调不是必需的，可以睡眠。

testmode\_cmd

实现cfg80211测试模式命令。传递的vif可能为NULL。回调函数可以休眠。

testmode\_dump

实现cfg80211测试模式转储。回调函数可以休眠。

flush

从硬件队列中刷新所有挂起的帧，确保硬件队列为空。队列参数是要刷新的队列位图，如果不同的虚拟接口使用不同的硬件队列，则这很有用；它也可能指示所有队列。如果参数drop设置为true，则可以丢弃挂起的帧。注意，vif可以为NULL。回调函数可以休眠。

channel\_switch

需要（或希望）从AP接收到的CSA的频道切换操作的驱动程序可以实现此回调。然后，它们必须调用ieee80211\_chswitch\_done()以指示频道切换的完成。

set\_antenna

在设备上设置天线配置（tx\_ant，rx\_ant）。参数是用于TX / RX的允许天线的位图。驱动程序可能通过返回-EINVAL拒绝它们无法支持的TX / RX掩码组合（也请参见nl80211.h NL80211\_ATTR\_WIPHY\_ANTENNA\_TX）。

get\_antenna

从设备获取当前天线配置（tx\_ant，rx\_ant）。

remain\_on\_channel

在给定信道上启动离线通道周期，必须在该信道上回调到ieee80211\_ready\_on\_channel()。请注意，正常信道流量不会停止，因为这是针对硬件卸载的。在离线信道上传输的帧会像往常一样传输，除了IEEE80211\_TX\_CTL\_TX\_OFFCHAN标志。当持续时间（将始终为非零值）到期时，驱动程序必须调用ieee80211\_remain\_on\_channel\_expired()。请注意，此回调可能在设备处于IDLE状态时调用，并且必须在此情况下接受。此回调可以休眠。

cancel\_remain\_on\_channel

请求在到期之前中止正在进行的离线通道周期。此回调可以休眠。

set\_ringparam

设置tx和rx环大小。

get\_ringparam

获取tx和rx环的当前和最大大小。

tx\_frames\_pending

在进入省电模式之前，检查硬件队列中是否有任何挂起的帧。

set\_bitrate\_mask

设置用于传输帧时进行速率控制选择的速率掩码。目前仅处理传统速率。回调函数可以休眠。

event\_callback

通知驱动程序mac80211中的任何事件。参见enum ieee80211\_event\_type以获取不同类型。回调必须是原子的。

allow\_buffered\_frames

准备设备，使得给定数量的帧可以发送到给定站点。在这个调用之后，mac80211会通过通常的TX路径发送这些帧。释放的帧的TX信息还将设置IEEE80211\_TX\_CTL\_NO\_PS\_BUFFER标志，并且最后一帧还将设置IEEE80211\_TX\_STATUS\_EOSP。如果释放来自多个TID的帧，并且驱动程序可能在TID之间重新排序它们，它必须在最后一帧上设置IEEE80211\_TX\_STATUS\_EOSP标志，清除所有其他帧上的该标志，并正确处理QoS头中的EOSP位。或者，它也可以调用ieee80211\_sta\_eosp()函数。tids参数是一个位图，告诉驱动程序帧将位于哪些TID上；它最多只有两个位设定。此回调必须是原子性的。

release\_buffered\_frames

根据给定的参数释放缓冲帧。在驱动程序为睡眠站点缓冲某些帧的情况下，mac80211将使用此回调通知驱动程序释放某些帧，无论是用于PS-poll还是uAPSD。请注意，如果more\_data参数为false，则驱动程序必须检查给定TID上是否有更多帧，如果有多个帧被释放，则仍必须在帧中设置more-data位。如果more\_data参数为true，则当然必须总是设置more-data位。tids参数告诉驱动程序从哪些TID中释放帧，对于PS-poll，它将始终只有一个位设置。如果这被用于PS-poll启动的释放，则num\_frames参数将始终为1，因此代码可共用。在这种情况下，驱动程序还必须在TX状态上设置IEEE80211\_TX\_STATUS\_EOSP标志（并且必须报告TX状态），以便正确结束PS-poll周期。这用于避免为重试的PS-poll帧发送多个响应。如果这是用于uAPSD，则num\_frames参数可能大于一，但驱动程序可以发送较少的帧（但必须发送至少一个）。在这种情况下，它还负责在帧的QoS头中设置EOSP标志。另外，当服务期结束时，驱动程序必须在SP中的最后一帧上设置IEEE80211\_TX\_STATUS\_EOSP。或者，它可以调用函数ieee80211\_sta\_eosp()通知mac80211 SP的结束。此回调必须是原子性的。

get\_et\_sset\_count

Ethtool API以获取字符串集计数。

get\_et\_stats

Ethtool API以获取一组u64统计信息。

get\_et\_strings

Ethtool API以获取一组字符串说明统计信息以及可能支持的其他类型的ethtool数据集。

mgd\_prepare\_tx

在关联之前为传输关联管理帧做准备。在多通道场景中，虚拟接口在关联之前绑定到频道，但由于它尚未关联，因此不必必要地给予空间，尤其是因为任何向P2P GO的传输都需要针对GO的电源保存状态进行同步。mac80211将在成功关联之前调用此函数，以允许驱动程序为传输给出频道时间、获得响应并能够与GO同步。对于设置IEEE80211\_HW\_DEAUTH\_NEED\_MGD\_TX\_PREP的驱动程序，当没有从AP/P2P GO收到信标时，mac80211也会在传输解关帧之前调用该函数。每次传输之前都会调用回调函数，并在返回后立即传输帧。附加信息传递在struct ieee80211\_prep\_tx\_info数据中。如果持续时间大于零，则mac80211提示驱动程序所请求的操作的持续时间。回调是可选的，可以（应该！）休眠。

mgd\_protect\_tdls\_discover

保护TDLS发现会话。在发送TDLS发现请求后，我们期望一个回复在AP的信道上到达。我们必须留在信道上（无PSM、扫描等），因为TDLS设置响应是一个直接的包，不会被AP缓冲。mac80211将在传输TDLS发现请求之前调用此函数。保护期建议至少为2 \*（DTIM周期）。回调是可选的，可以休眠。

add\_chanctx

通知设备驱动程序有新的频道上下文创建。此回调可以休眠。

remove\_chanctx

通知设备驱动程序有频道上下文被销毁。此回调可以休眠。

change\_chanctx

通知设备驱动程序在将不同的虚拟接口与不同设置组合到同一频道上下文时可能发生的频道上下文更改。此回调可以休眠。

assign\_vif\_chanctx

通知设备驱动程序已将信道上下文绑定到虚拟接口。可能使用的方法是进行硬件队列重映射。此回调函数可能会休眠。

unassign\_vif\_chanctx

通知设备驱动程序已将信道上下文与虚拟接口解除绑定。此回调函数可能会休眠。

switch\_vif\_chanctx

将一定数量的虚拟接口从一个信道上下文切换到另一个信道上下文，如 ieee80211\_vif\_chanctx\_switch 列表所指定的那样，根据 ieee80211\_chanctx\_switch\_mode 中定义的模式。此回调函数可能会休眠。

reconfig\_complete

在调用 ieee80211\_restart\_hw() 并在恢复时调用，当重新配置完成时调用。这可以帮助驱动程序实现重新配置步骤（并指示 mac80211 准备好接收帧）。此回调函数可能会休眠。

ipv6\_addr\_change

给定接口上的 IPv6 地址分配已更改。目前，仅为托管或 P2P 客户端接口调用此函数。此回调函数是可选的，不得睡眠。

channel\_switch\_beacon

开始切换到新信道。在调用此函数之前，修改信标以包括 CSA 或 ECSA IE。这些 IE 中的相应计数字段必须递减，当它们达到 1 时，驱动程序必须调用 ieee80211\_csa\_finish()。使用 ieee80211\_beacon\_get() 的驱动程序会使 mac80211 递减 csa 计数器，但必须在信标传输后检查是否为 1，然后调用 ieee80211\_csa\_finish()。如果 CSA 计数从零或 1 开始，则不会调用此函数，因为在切换之前不会有任何时间来发出信标。

pre\_channel\_switch

这是一个可选的回调函数，在开始信道切换过程之前调用（即当 STA 获取 CSA 或用户空间启动的信道切换时），允许驱动程序准备信道切换。

post\_channel\_switch

这是一个可选的回调函数，在信道切换过程完成后调用，允许驱动程序返回到正常配置。

join\_ibss

加入 IBSS（在 IBSS 接口上）；在设置了 bss\_conf 中的所有信息并可以检索到信标后调用此函数。在调用此函数之前，将绑定一个信道上下文。

leave\_ibss

再次离开 IBSS。

get\_expected\_throughput

提取到指定站点的预期吞吐量。返回以 Kbps 表示。如果 RC 算法没有适当的数据提供，则返回 0。

get\_txpower

获取基于配置和硬件限制的当前最大传输功率（以 dBm 为单位）。

tdls\_channel\_switch

与 TDLS 对等体一起启动信道切换。驱动程序负责不断启动信道切换操作并返回到基础信道以与 AP 通信。在调用中，驱动程序接收到信道切换请求模板和模板中切换定时 IE 的位置。该模板仅在调用内有效，驱动程序可以选择复制 skb 以进行进一步重用。

tdls\_cancel\_channel\_switch

停止与 TDLS 对等体的信道切换。通话完成时，两个对等体必须在基本信道上。

tdls\_recv\_channel\_switch

从远程对等体接收到与 TDLS 信道切换相关的帧（请求或响应）。驱动程序获取从传入帧解析出的参数，并可以使用它们来继续进行正在进行的信道切换操作。此外，还提供了信道切换响应模板，以及模板中切换定时 IE 的位置。skb 只能在函数调用中使用。

wake\_tx\_queue

当新数据包添加到队列时调用。

sync\_rx\_queues

处理 RSS 队列中的所有挂起帧。在这种情况下，需要同步，因为驱动程序在其 RSS 队列中具有等待接收的帧，这些帧在当前接受控制路径操作（例如去关联）之前已经接收，但尚未处理。

start\_nan

加入现有的 NAN 簇或创建一个新簇。

stop\_nan

离开 NAN 簇。

nan\_change\_conf

更改 NAN 配置。cfg80211\_nan\_conf 中的数据包含完整的新配置，并且更改指定哪些参数与上一个 NAN 配置不同。驱动程序既获取完整配置又获取自上次 NAN 配置以来更改的参数，因为某些设备可能需要完整配置，而其他设备仅需要更改的参数。

add\_nan\_func

添加 NAN 功能。成功返回 0。cfg80211\_nan\_func 中的数据不能在此调用的范围之外引用。

del\_nan\_func

删除 NAN 功能。驱动程序必须在移除时使用 NL80211\_NAN\_FUNC\_TERM\_REASON\_USER\_REQUEST 原因代码调用 ieee80211\_nan\_func\_terminated()。

**说明**

此结构包含驱动程序可能处理或必须处理的各种回调，例如将硬件配置到新信道或传输帧。

#### struct [ieee80211\_hw](https://www.kernel.org/doc/html/latest/driver-api/80211/mac80211.html" \l "c.ieee80211_hw" \o "ieee80211_hw) \*ieee80211\_alloc\_hw(size\_t priv\_data\_len, const struct [ieee80211\_ops](https://www.kernel.org/doc/html/latest/driver-api/80211/mac80211.html" \l "c.ieee80211_ops" \o "ieee80211_ops) \*ops)

分配新的硬件设备

**参数**

size\_t priv\_data\_len

私人数据长度

const struct ieee80211\_ops \*ops

此设备的回调

**说明**

每个硬件设备必须调用一次。返回的指针必须用于引用此设备在调用其他函数时。mac80211 为驱动程序分配一个由 struct ieee80211\_hw 中的 priv 指向的私有数据区域，该区域的大小由 priv\_data\_len 给出。

**返回**

新的硬件设备指针，或者在出现错误时返回NULL。

#### int ieee80211\_register\_hw（struct ieee80211\_hw \*hw）

注册硬件设备

**参数**

struct ieee80211\_hw \*hw

由ieee80211\_alloc\_hw（）返回的要注册的设备。

**说明**

在mac80211中调用所有其他函数之前必须调用此函数。请注意，在硬件可以注册之前，您需要填充包含的wiphy信息。

**返回**

成功返回0。否则为错误代码。

#### void ieee80211\_unregister\_hw(struct ieee80211\_hw \*hw)

取消注册硬件设备

**参数**

struct ieee80211\_hw \*hw

要取消注册的硬件

**说明**

此函数指示mac80211释放已分配的资源并取消注册网络子系统中的netdevice。

#### void ieee80211\_free\_hw (struct ieee80211\_hw \*hw)

释放硬件说明符

**参数**

struct ieee80211\_hw \*hw

要释放的硬件

**说明**

此函数释放已分配的所有内容，包括驱动程序的私有数据。在调用此函数之前，必须调用ieee80211\_unregister\_hw（）。

### 物理层配置

待定

此章节应说明PHY处理，包括启动/停止回调和使用的各种结构。

#### struct ieee80211\_conf

设备配置

**定义**

struct ieee80211\_conf {

u32 flags;

int power\_level, dynamic\_ps\_timeout;

u16 listen\_interval;

u8 ps\_dtim\_period;

u8 long\_frame\_max\_tx\_count, short\_frame\_max\_tx\_count;

struct cfg80211\_chan\_def chandef;

bool radar\_enabled;

enum ieee80211\_smps\_mode smps\_mode;

};

**成员**

flags

上述配置标志

power\_level

所请求的发送功率（以dBm为单位），仅向后兼容值，设置为所有接口的最小值

dynamic\_ps\_timeout

动态省电超时（以毫秒为单位），请参见下面的省电文档。只有在设置了CONF\_PS标志时，此变量才有效。

listen\_interval

以信标间隔单位的监听时间间隔

ps\_dtim\_period

我们所连接的AP的DTIM周期，用于省电。只有在收到信标并知道DTIM周期之后才会启用省电。

long\_frame\_max\_tx\_count

“长”帧（未受RTS保护的帧）的最大传输次数（称为“dot11LongRetryLimit”的802.11），但实际上是传输次数而不是重试次数

short\_frame\_max\_tx\_count

“短”帧的最大传输次数，称为802.11中的“dot11ShortRetryLimit”，但实际上是传输次数而不是重试次数

chandef

要调谐的信道定义

radar\_enabled

是否启用雷达检测

smps\_mode

空间多路复用省电模式；请注意，当设备未配置HT通道时，使用IEEE80211\_SMPS\_STATIC。请注意，仅当不使用信道上下文时，此选项才有效，否则每个信道上下文都有列出的链的数量。

**说明**

此结构指示驱动程序应如何配置硬件。

#### enum ieee80211\_conf\_flags

配置标志

**常量**

IEEE80211\_CONF\_MONITOR

存在监视器接口-使用此来确定例如是否为数据包计算时间戳，而不是使用过滤器标志！

IEEE80211\_CONF\_PS

在管理模式下启用802.11省电模式。这是IEEE 802.11-2007第11.2节定义的省电模式，这意味着硬件仍会唤醒以接收信标，能够发送帧并接收可能的确认帧。不要与硬件特定的唤醒/睡眠状态混淆，驱动程序负责该任务。有关详细信息，请参见“Powersave support”部分。

IEEE80211\_CONF\_IDLE

设备正在运行，但处于空闲状态；如果设置了标志，则驱动程序应准备好处理配置请求，但可以尽可能地关闭设备。通常，当接口设置为UP但未关联或扫描时，将设置此标志，但在激活监视器接口时也可以取消设置此标志。

IEEE80211\_CONF\_OFFCHANNEL

设备当前未处于其主操作信道上。

**说明**

用于定义PHY配置选项的标志

### 虚拟接口

待定

此章节应说明驱动程序相关的虚拟接口基础知识（VLAN、MGMT等不涉及）。它应解释如何使用add\_iface / remove\_iface回调以及接口配置回调。

与AP模式相关的事项应在此处进行讨论。

涉及支持多个接口的事项应放在相应章节中，但应在此处放置一个大大的注意事项，并建议首先在STA模式下仅允许单个接口！

#### struct ieee80211\_vif

每个接口的数据

**定义**

struct ieee80211\_vif {

enum nl80211\_iftype type;

struct ieee80211\_bss\_conf bss\_conf;

u8 addr[ETH\_ALEN] ;

bool p2p;

bool csa\_active;

bool mu\_mimo\_owner;

u8 cab\_queue;

u8 hw\_queue[IEEE80211\_NUM\_ACS];

struct ieee80211\_txq \*txq;

struct ieee80211\_chanctx\_conf \_\_rcu \*chanctx\_conf;

u32 driver\_flags;

#ifdef CONFIG\_MAC80211\_DEBUGFS;

struct dentry \*debugfs\_dir;

#endif;

unsigned int probe\_req\_reg;

u8 drv\_priv[0] ;

};

**成员**

type

此虚拟接口的类型

bss\_conf

此接口的BSS配置，我们自己的或我们关联的BSS

addr

此接口的地址

p2p

指示此AP或STA接口是否为点对点接口，即GO或p2p-sta

csa\_active

标记是否正在进行信道切换。它在内部被sdata\_lock和local->mtx写保护，因此持有其中任何一个也可以对其进行读取访问。

mu\_mimo\_owner

表示接口拥有MU-MIMO能力。

cab\_queue

后信标（DTIM信标）队列，仅适用于AP模式。

hw\_queue

每个AC的硬件队列。

txq

多播数据TX队列（如果驱动程序使用TXQ抽象）。

chanctx\_conf

分配给此接口的信道上下文，或者在未分配时为NULL。由于TX路径需要访问它，因此此指针受RCU保护；即使当它为NULL时，netdev carrier始终关闭，仍然可能存在竞争，并且数据包在它切换回NULL后仍然可能被处理。

driver\_flags

驱动程序为此接口拥有的标志/功能，在添加接口时需要设置（或清除），或者在运行时更改接口类型（如果驱动程序支持），mac80211永远不会触摸此字段。

debugfs\_dir

debugfs目录条目，驱动程序可以使用它来创建自己的每个接口调试文件。请注意，对于虚拟监视器接口（如果请求），它将为NULL。

probe\_req\_reg

探测请求应报告给此接口的mac80211。

drv\_priv

驱动程序使用的数据区域，将始终对齐sizeof(void \*)。

**说明**

此结构中的数据在虚拟接口的生命周期中始终存在供驱动程序使用。

### 接收和发送处理

#### 该怎么办

待定

应说明mac80211 /驱动程序中的接收和发送路径以及传输状态处理。

#### 帧格式

通常情况下，在mac80211和驱动程序之间传递帧时，它们以IEEE 802.11头开始，并包括通过空气发送的相同八位字节，除了FCS应由硬件计算。

然而，对于高级功能，有各种例外情况：

第一个例外是硬件加密和解密卸载，其中IV / ICV可能在硬件中生成或不生成。

其次，当硬件处理分段时，从mac80211传递到驱动程序的帧是MSDU而不是MPDU。

#### 数据包对齐

驱动程序始终需要将数据包对齐到两个字节边界。此外，如果可能，应以一种保证所包含的IP头对齐到四个字节边界的方式对齐有效载荷数据。对于常规帧而言，这仅意味着将有效载荷对齐到四个字节边界（因为IP头直接包含其中，或者在其前面包含有长度可被四整除的IV / RFC1042头）。如果有效载荷数据未正确对齐且架构不支持高效的非对齐操作，则mac80211将对齐数据。

然而，在A-MSDU帧中，有效载荷数据地址必须产生二模四，因为A-MSDU帧内有14个字节的802.3头，将IP头进一步后移至四的倍数。值得庆幸的是，这次规格足够明智，要求将每个A-MSDU子帧填充到长度为四的倍数。

不支持像Atheros硬件添加的填充位于802.11头和有效载荷之间，这种情况下驱动程序需要将802.11头移动到有效载荷直接前面。

#### 从中断调用mac80211

只有ieee80211\_tx\_status\_irqsafe（）和ieee80211\_rx\_irqsafe（）可以在硬件中断上下文中调用。低级驱动程序不能在硬件中断上下文中调用任何其他函数。如果确实需要这样的调用，则低级驱动程序应首先ACK中断，然后执行IEEE 802.11代码调用，例如从计划的工作队列甚至是tasklet函数。

注意：如果驱动程序选择使用\_irqsafe（）函数，则可能不使用非IRQ-safe函数！

#### 函数/定义

**struct ieee80211\_rx\_status**

接收状态

1）定义

struct ieee80211\_rx\_status {

u64 mactime;

u64 boottime\_ns;

u32 device\_timestamp;

u32 ampdu\_reference;

u32 flag;

u16 freq;

u8 enc\_flags;

u8 encoding:2, bw:3, he\_ru:3;

u8 he\_gi:2, he\_dcm:1;

u8 rate\_idx;

u8 nss;

u8 rx\_flags;

u8 band;

u8 antenna;

s8 signal;

u8 chains;

s8 chain\_signal[IEEE80211\_MAX\_CHAINS];

u8 ampdu\_delimiter\_crc;

};

2）成员

mactime

第一个数据符号（MPDU）到达硬件时64位时间同步功能（TSF）计时器的微秒值。

boottime\_ns

该帧接收的CLOCK\_BOOTTIME时间戳，这仅适用于更新扫描缓存的信标和探测响应。

device\_timestamp

设备的任意时间戳，mac80211不使用它，但可以存储它并将其传递回驱动程序以进行同步。

ampdu\_reference

A-MPDU参考号，每个A-MPDU必须具有不同的值，但是在一个A-MPDU内的每个子帧必须相同。

flag

RX\_FLAG\_\*（接收标志）

freq

收到此帧时无线电调谐的频率，以MHz为单位。必须为管理帧设置此字段，但对于数据（其他）帧没有严格要求-因为它只影响radiotap报告。

enc\_flags

使用来自枚举mac80211\_rx\_encoding\_flags的位

encoding

枚举mac80211\_rx\_encoding

bw

枚举rate\_info\_bw

he\_ru

HE RU，来自枚举nl80211\_he\_ru\_alloc

he\_gi

HE GI，来自枚举nl80211\_he\_gi

he\_dcm

HE DCM值

rate\_idx

数据速率的索引，以波段支持的速率或如果使用HT或VHT，则为MCS索引 (RX\_FLAG\_HT/RX\_FLAG\_VHT)

nss

流的数量（仅限VHT和HE）

rx\_flags

用于mac80211的内部RX标志

band

接收此帧时的活动波段

antenna

使用的天线

signal

接收此帧时的信号强度，可能为dBm、dB或无法确定，具体取决于硬件功能标志IEEE80211\_HW\_SIGNAL\_\*

chains

用于哪些单独的接收链填充了信号强度值的位掩码。

chain\_signal

每个链路的信号强度，以dBm为单位（与信号不同，不支持dB或未指定的单位）

ampdu\_delimiter\_crc

A-MPDU分隔符CRC

3）说明

低级驱动程序应将此信息（硬件支持的子集）提供给每个接收帧的802.11代码，放置于skb的控制缓冲区（cb）中。

**enum mac80211\_rx\_flags**

接收标志

1）常量

RX\_FLAG\_MMIC\_ERROR

此帧报告了Michael MIC错误。与RX\_FLAG\_MMIC\_STRIPPED一起使用。

RX\_FLAG\_DECRYPTED

该帧已在硬件中解密。

RX\_FLAG\_MACTIME\_PLCP\_START

在RX状态中传递的时间戳（mactime字段）有效，包含接收同步前导的时间。

RX\_FLAG\_MMIC\_STRIPPED

从该帧中剥离了Michael MIC，并由硬件进行了验证。

RX\_FLAG\_IV\_STRIPPED

从该帧中剥离了IV和ICV。如果设置了此标志，则堆栈无法进行任何重播检测，因此驱动程序或硬件将不得不执行该操作。

RX\_FLAG\_FAILED\_FCS\_CRC

如果在帧上的FCS检查失败，请设置此标志。

RX\_FLAG\_FAILED\_PLCP\_CRC

如果在帧上的PCLP检查失败，请设置此标志。

RX\_FLAG\_MACTIME\_START

在RX状态中传递的时间戳（mactime字段）有效，包含接收到MPDU的第一个符号的时间。这对于监控模式和适当的IBSS合并很有用。

RX\_FLAG\_NO\_SIGNAL\_VAL

信号强度值不存在。仅适用于数据帧（主要是A-MPDU）。

RX\_FLAG\_AMPDU\_DETAILS

已知A-MPDU的详细信息，特别是必须填充参考号（ampdu\_reference），并且每个A-MPDU必须具有不同的号码。

RX\_FLAG\_PN\_VALIDATED

当前仅对CCMP / GCMP帧有效，此标志指示已验证PN以进行重放保护。请注意，仅当还解密了帧时（即必须设置RX\_FLAG\_DECRYPTED）才支持此标志。

RX\_FLAG\_DUP\_VALIDATED

如果驱动程序自行进行了去重复，则应设置此标志。

RX\_FLAG\_AMPDU\_LAST\_KNOWN

已知上一个子帧是最后一个，应在单个A-MPDU的所有子帧上设置此标志。

RX\_FLAG\_AMPDU\_IS\_LAST

此子帧是A-MPDU的最后一个子帧。

RX\_FLAG\_AMPDU\_DELIM\_CRC\_ERROR

在此子帧上检测到了分隔符CRC错误。

RX\_FLAG\_AMPDU\_DELIM\_CRC\_KNOWN

已知分隔符CRC字段（CRC存储在ampdu\_delimiter\_crc字段中）。

RX\_FLAG\_MACTIME\_END

在RX状态中传递的时间戳（mactime字段）有效，包含接收到MPDU的最后一个符号（包括FCS）的时间。

RX\_FLAG\_ONLY\_MONITOR

仅将帧报告给监视接口，而不以任何常规方式进行处理。如果驱动程序卸载某些帧但仍想将它们报告给嗅探目的，则此功能很有用。

RX\_FLAG\_SKIP\_MONITOR

将帧处理并报告给除监视接口以外的所有接口。如果驱动程序卸载某些帧但仍要报告它们以进行嗅探，则此功能很有用。

RX\_FLAG\_AMSDU\_MORE

出于性能原因，某些驱动程序可能更喜欢报告单独的A-MSDU子帧而不是一个庞大的帧。除了A-MSDU的最后一个MSDU外，所有MSDU都应该设置此标志。例如，如果A-MSDU有三个帧，则前两个必须设置该标志，而第三个（最后一个）则不能设置此标志。此标志用于正确处理重传/重复恢复，因为A-MSDU子帧共享相同的序列号。报告的子帧可以是常规MSDU或单个A-MSDU。子帧不得与其他帧交错。

RX\_FLAG\_RADIOTAP\_VENDOR\_DATA

此帧包含skb->data中的特定于供应商的radiotap数据（在帧之前），如struct ieee80211\_vendor\_radiotap所说明。

RX\_FLAG\_MIC\_STRIPPED

在该数据包中，MIC已被剥离。解密是由硬件执行的。

RX\_FLAG\_ALLOW\_SAME\_PN

允许与先前相同的数据包具有相同的PN。这用于可以与第一个子帧具有相同PN的AMSDU子帧。

RX\_FLAG\_ICV\_STRIPPED

从此帧中去除了ICV。必须在硬件中进行CRC检查。

RX\_FLAG\_AMPDU\_EOF\_BIT

此帧的A-MPDU分隔符中的EOF位的值

RX\_FLAG\_AMPDU\_EOF\_BIT\_KNOWN

已知EOF值

RX\_FLAG\_RADIOTAP\_HE

存在HE radiotap数据（struct ieee80211\_radiotap\_he，mac80211将填充DATA3\_DATA\_MCS、DATA3\_DATA\_DCM、DATA3\_CODING、DATA5\_GI、DATA5\_DATA\_BW\_RU\_ALLOC、DATA6\_NSTS等RX信息数据，因此在构建此数据时将这些数据清零）

RX\_FLAG\_RADIOTAP\_HE\_MU

存在HE MU radiotap数据（struct ieee80211\_radiotap\_he\_mu）

2）说明

这些标志与struct ieee80211\_rx\_status的flag成员一起使用。

**enum mac80211\_tx\_info\_flags**

用于说明传输信息/状态的标志

1）常量

IEEE80211\_TX\_CTL\_REQ\_TX\_STATUS

要求为该帧回调TX状态。

IEEE80211\_TX\_CTL\_ASSIGN\_SEQ

驱动程序必须为此帧分配一个序列号，注意不要覆盖分片编号并仅在设置IEEE80211\_TX\_CTL\_FIRST\_FRAGMENT标志时增加序列号。mac80211将为QoS数据帧正确分配序列号，但对于非QoS数据和管理帧，mac80211无法正确分配序列号，因为信标也需要从该计数器中获取，并且mac80211无法保证正确排序。如果设置了此标志，则驱动程序应指示硬件为该帧分配一个序列号或自行分配一个序列号。参见IEEE 802.11-2007 7.1.3.4.1第3段。此标志将始终对信标设置，并对没有序列号字段的帧始终保持清晰。

IEEE80211\_TX\_CTL\_NO\_ACK

告诉底层不要等待ack

IEEE80211\_TX\_CTL\_CLEAR\_PS\_FILT

清除目标站点的省电过滤器

IEEE80211\_TX\_CTL\_FIRST\_FRAGMENT

这是帧的第一个片段

IEEE80211\_TX\_CTL\_SEND\_AFTER\_DTIM

在DTIM信标后发送此帧

IEEE80211\_TX\_CTL\_AMPDU

应将此帧作为A-MPDU的一部分发送

IEEE80211\_TX\_CTL\_INJECTED

帧已注入，为mac80211内部使用。

IEEE80211\_TX\_STAT\_TX\_FILTERED

由于目标STA处于省电模式，因此未传输该帧。注意，为避免竞争条件，硬件或固件必须在接收指示站点进入睡眠模式的帧时设置过滤器（必须在设备上对已在队列中的帧过滤），并且只有在mac80211通过设置IEEE80211\_TX\_CTL\_CLEAR\_PS\_FILT（请参见上文）为它提供OK后才能取消设置该标志，因为只有在那时才可以保证没有更多的帧在硬件队列中。

IEEE80211\_TX\_STAT\_ACK

帧已经被确认

IEEE80211\_TX\_STAT\_AMPDU

该帧已汇聚，因此状态适用于整个汇聚。

IEEE80211\_TX\_STAT\_AMPDU\_NO\_BACK

未返回块ack，因此考虑使用块ack请求（BAR）。

IEEE80211\_TX\_CTL\_RATE\_CTRL\_PROBE

内部使用的mac80211，可以由速率控制算法设置为指示探针速率，将在分段的帧中清除（除了在最后一个片段中）

IEEE80211\_TX\_INTFL\_OFFCHAN\_TX\_OK

内部mac80211。用于指示可以在离线操作时传输帧，即使队列已停止，也仍然会传输常规帧并且预计设备会正确处理它们。

IEEE80211\_TX\_CTL\_HW\_80211\_ENCAP

此帧使用硬件封装（头转换）

IEEE80211\_TX\_INTFL\_RETRIED

完全内部使用mac80211，用于指示已经由于PS而重试了一次帧

IEEE80211\_TX\_INTFL\_DONT\_ENCRYPT

完全内部mac80211，用于指示帧不应加密

IEEE80211\_TX\_CTL\_NO\_PS\_BUFFER

此帧是对轮询帧（PS-Poll或uAPSD）或非可缓存MMPDU的响应，即使站点处于省电模式，也必须发送。

IEEE80211\_TX\_CTL\_MORE\_FRAMES

在当前帧之后将传递更多帧，驱动程序可以使用此项来启动DMA队列，仅在未设置或队列已满时使用。

IEEE80211\_TX\_INTFL\_RETRANSMISSION

由于目标正在睡眠，因此此帧正在重新传输TX状态，不得再次修改它（不得分配序列号，加密等）。

IEEE80211\_TX\_INTFL\_MLME\_CONN\_TX

此帧由MLME代码用于连接建立进行传输，这表示其状态应该触发MLME状态机。

IEEE80211\_TX\_INTFL\_NL80211\_FRAME\_TX

通过nl80211 MLME命令请求帧（mac80211内部用于确定是否将TX状态发送到用户空间）

IEEE80211\_TX\_CTL\_LDPC

告诉驱动程序在此帧中使用LDPC

IEEE80211\_TX\_CTL\_STBC

对此帧启用空时块编码（STBC）并选择它可以使用的最大流数。

IEEE80211\_TX\_CTL\_TX\_OFFCHAN

当硬件执行保持在信道上的负荷时，标记此数据包以在空闲信道上进行传输-正常数据包仍在流动，应按预期由设备处理。

IEEE80211\_TX\_INTFL\_TKIP\_MIC\_FAILURE

将此数据包标记为用于TKIP测试。它将使用错误的Michael MIC密钥发送，以允许测试TKIP对策。

IEEE80211\_TX\_CTL\_NO\_CCK\_RATE

此帧将使用非CCK速率发送。实际上，此标志用于管理帧，特别是在2GHz频段中未以CCK速率发送的P2P帧。

IEEE80211\_TX\_STATUS\_EOSP

该数据包标记服务期结束，当其状态报告时服务期结束。对于mac80211传输的SP中的帧，它已经设置；对于驱动程序帧，驱动程序可以设置此标志。它还用于对PS-Poll应答执行相同操作。

IEEE80211\_TX\_CTL\_USE\_MINRATE

该帧将以最低速率发送。此标志用于在监视连接目的时以最小速率发送nullfunc帧。

IEEE80211\_TX\_CTL\_DONTFRAG

不要分片此数据包，即使它会因大小被分片（这是可选项，仅用于监视注入）。

IEEE80211\_TX\_STAT\_NOACK\_TRANSMITTED

已成功传输标记有IEEE80211\_TX\_CTL\_NO\_ACK的帧，没有任何错误（如与驱动程序/硬件特定的问题）。不应为不使用IEEE80211\_TX\_CTL\_NO\_ACK请求no-ack行为的帧设置此标志。

2）说明

这些标志与ieee80211\_tx\_info的标志成员一起使用。

3）注

如果必须向枚举添加新标志，则在必要时不要忘记更新

IEEE80211\_TX\_TEMPORARY\_FLAGS。

**enum mac80211\_tx\_control\_flags**

标志用于说明传输控制

1）常量

IEEE80211\_TX\_CTRL\_PORT\_CTRL\_PROTO

这个帧是端口控制协议帧（例如EAP）。

IEEE80211\_TX\_CTRL\_PS\_RESPONSE

这个帧是对轮询帧（PS-Poll或uAPSD）的响应。

IEEE80211\_TX\_CTRL\_RATE\_INJECT

此帧是注入速率信息的帧

IEEE80211\_TX\_CTRL\_AMSDU

这个帧是A-MSDU帧

IEEE80211\_TX\_CTRL\_FAST\_XMIT

该帧正在通过fast\_xmit路径进行

2）说明

这些标志在tx\_info->control.flags中使用

**enum mac80211\_rate\_control\_flags**

由速率控制算法设置的每个速率的标志。

1）常量

IEEE80211\_TX\_RC\_USE\_RTS\_CTS

使用RTS/CTS交换来传输该速率。

IEEE80211\_TX\_RC\_USE\_CTS\_PROTECT

需要CTS-to-self保护。如果当前BSS需要ERP保护，则设置此选项。

IEEE80211\_TX\_RC\_USE\_SHORT\_PREAMBLE

使用短前缀。

IEEE80211\_TX\_RC\_MCS

HT速率。

IEEE80211\_TX\_RC\_GREEN\_FIELD

指示是否应在Greenfield模式下使用此速率。

IEEE80211\_TX\_RC\_40\_MHZ\_WIDTH

指示频道宽度是否应为40 MHz。

IEEE80211\_TX\_RC\_DUP\_DATA

如果当前频道类型为NL80211\_CHAN\_HT40MINUS或NL80211\_CHAN\_HT40PLUS，则应在相邻的两个20 MHz频道上传输该帧。

IEEE80211\_TX\_RC\_SHORT\_GI

应使用短保护间隔来使用此速率。

IEEE80211\_TX\_RC\_VHT\_MCS

VHT MCS速率，在这种情况下，idx字段分为高4位（Nss）和低4位（MCS编号）

IEEE80211\_TX\_RC\_80\_MHZ\_WIDTH

表示80 MHz传输

IEEE80211\_TX\_RC\_160\_MHZ\_WIDTH

表示160 MHz传输（80 + 80尚不支持）

2）说明

这些标志由速率控制算法在tx期间为每个速率设置，位于struct ieee80211\_tx\_rate的flags成员中。

**struct ieee80211\_tx\_rate**

速率选择/状态

1）定义

struct ieee80211\_tx\_rate {

s8 idx;

u16 count:5, flags:11;

};

2）成员

idx

尝试发送的速率索引

count

在切换到下一个速率之前在此速率上尝试的次数

flags

速率控制标志（enum mac80211\_rate\_control\_flags）

3）说明

idx的值为-1表示无效速率，如果在重试速率数组中使用，则不应再尝试更多速率。

在用于传输状态报告时，驱动程序应始终报告使用的速率及其标志。

[struct ieee80211\_tx\_info](https://www.kernel.org/doc/html/v4.19/driver-api/80211/mac80211.html" \l "c.ieee80211_tx_info" \o "ieee80211_tx_info)在控制信息中包含这些结构的数组，它会根据应该发送的内容由速率控制算法填充。例如，如果此数组包含格式为 { <idx>, <count> } 的信息:

{ 3, 2 }, { 2, 2 }, { 1, 4 }, { -1, 0 }, { -1, 0 }

那么这意味着帧应该以速率 3 最多传输两次，以速率 2 最多传输两次，如果没有得到确认，则以速率 1 最多传输四次。假设它在第五次尝试后得到对等方的确认，则状态信息应包含：

{ 3, 2 }, { 2, 2 }, { 1, 1 }, { -1, 0 } ...

因为它以速率 3 传输了两次，以速率 2 传输了两次，以速率 1 传输了一次，之后我们收到了确认。

**struct ieee80211\_tx\_info**

skb传递信息

1）定义

struct ieee80211\_tx\_info {

u32 flags;

u8 band;

u8 hw\_queue;

u16 ack\_frame\_id;

union {

struct {

union {

struct {

struct ieee80211\_tx\_rate rates[ IEEE80211\_TX\_MAX\_RATES];

s8 rts\_cts\_rate\_idx;

u8 use\_rts:1;

u8 use\_cts\_prot:1;

u8 short\_preamble:1;

u8 skip\_table:1;

};

unsigned long jiffies;

};

struct ieee80211\_vif \*vif;

struct ieee80211\_key\_conf \*hw\_key;

u32 flags;

codel\_time\_t enqueue\_time;

} control;

struct {

u64 cookie;

} ack;

struct {

struct ieee80211\_tx\_rate rates[IEEE80211\_TX\_MAX\_RATES];

s32 ack\_signal;

u8 ampdu\_ack\_len;

u8 ampdu\_len;

u8 antenna;

u16 tx\_time;

bool is\_valid\_ack\_signal;

void \*status\_driver\_data[19 / sizeof(void \*)];

} status;

struct {

struct ieee80211\_tx\_rate driver\_rates[ IEEE80211\_TX\_MAX\_RATES];

u8 pad[4];

void \*rate\_driver\_data[ IEEE80211\_TX\_INFO\_RATE\_DRIVER\_DATA\_SIZE / sizeof(void \*)];

};

void \*driver\_data[ IEEE80211\_TX\_INFO\_DRIVER\_DATA\_SIZE / sizeof(void \*)];

};

};

2）成员

flags

传输信息标志，上面定义

band

要传输的频段（用于检查比赛）

hw\_queue

HW queue to put the frame on,skb\_get\_queue\_mapping()给出 AC

ack\_frame\_id

TX 状态的内部帧 ID，内部使用

{unnamed\_union}

匿名的

control

控制数据联合

{unnamed\_union}

匿名的

{unnamed\_struct}

匿名的

status

状态数据联盟

{unnamed\_struct}

匿名的

driver\_data

driver\_data 指针数组

描述

这个结构体放在skb->cb中有三个用途：

1. mac80211 TX 控制 - mac80211 告诉驱动程序做什么
2. 驱动程序内部使用（如果适用）
3. TX 状态信息 - 驱动程序告诉 mac80211 发生了什么

**void ieee80211\_tx\_info\_clear\_status(struct [ieee80211\_tx\_info](https://www.kernel.org/doc/html/v4.19/driver-api/80211/mac80211.html" \l "c.ieee80211_tx_info" \o "ieee80211_tx_info) \* info)**

清除TX状态

1）参数

struct ieee80211\_tx\_info \* info

要清除的struct ieee80211\_tx\_info。

2）说明

当驱动程序将skb传回mac80211时，必须报告TX状态中的许多内容。该函数清除TX状态中的一切，但保留速率控制信息（它确实清除计数，因为您需要填充该计数）。

3）注意

如果您不使用info->driver\_data，则只能使用此函数！如果您仅需要允许较少空间的info->rate\_driver\_data，请改用该项。

**void ieee80211\_rx（struct ieee80211\_hw \* hw，struct sk\_buff \* skb）**

接收帧

1）参数

struct ieee80211\_hw \* hw

该帧所在的硬件

struct sk\_buff \* skb

接收的缓冲区，在此调用后由mac80211拥有

2）说明

使用此函数将接收的帧传递给mac80211。 skb中的接收缓冲区必须以IEEE 802.11标题开头。如果使用分页skb，则建议驱动程序将帧的ieee80211标题放在skb的线性部分上，以避免栈通过内存分配和/或memcpy进行复制。

此函数不能在IRQ上下文中调用。将此函数的调用同步对单个硬件。不能将此函数的调用，ieee80211\_rx\_ni（）和ieee80211\_rx\_irqsafe（）混合用于单个硬件。不能同时运行ieee80211\_tx\_status（）或ieee80211\_tx\_status\_ni（）。

在进程上下文中使用ieee80211\_rx\_ni（）代替。

**void ieee80211\_rx\_ni（struct ieee80211\_hw \* hw，struct sk\_buff \* skb）**

接收帧（在进程上下文中）

1）参数

struct ieee80211\_hw \* hw

该帧所在的硬件

struct sk\_buff \* skb

接收的缓冲区，在此调用后由mac80211拥有

2）说明

像ieee80211\_rx（）一样，但可以在进程上下文中调用（内部禁用底部半部分）。

对于单个硬件，不能将此函数的调用，ieee80211\_rx（）和ieee80211\_rx\_irqsafe（）混合使用。不能同时运行ieee80211\_tx\_status（）或ieee80211\_tx\_status\_ni（）。

**void ieee80211\_rx\_irqsafe（struct ieee80211\_hw \* hw，struct sk\_buff \* skb）**

接收帧

1）参数

struct ieee80211\_hw \* hw

该帧所在的硬件

struct sk\_buff \* skb

接收的缓冲区，在此调用后由mac80211拥有

2）说明

像ieee80211\_rx（）一样，但可以在IRQ上下文中调用（内部将延迟到任务let）。

不能将此函数的调用，ieee80211\_rx（）或ieee80211\_rx\_ni（）混合用于单个硬件。不能同时运行ieee80211\_tx\_status（）或ieee80211\_tx\_status\_ni（）。

**struct ieee80211\_tx\_status**

用于速率控制的扩展TX状态信息

1）定义

struct ieee80211\_tx\_status {

struct ieee80211\_sta \*sta;

struct ieee80211\_tx\_info \*info;

struct sk\_buff \*skb;

};

2）成员

sta

发送数据包的站点

info

基本 tx 状态信息

skb

数据包 skb（如果驱动程序未提供，则可以为 NULL）

**void ieee80211\_tx\_status(struct [ieee80211\_hw](https://www.kernel.org/doc/html/latest/driver-api/80211/mac80211.html" \l "c.ieee80211_hw" \o "ieee80211_hw) \*hw, struct [sk\_buff](https://www.kernel.org/doc/html/latest/networking/kapi.html" \l "c.sk_buff" \o "sk_buff) \*skb)**

传输状态回调

1）参数

struct ieee80211\_hw \* hw

传输帧的硬件

struct sk\_buff \* skb

已传输的帧，在此调用后由mac80211拥有

2）说明

在所有传输的帧传输后调用此函数。可以不为组播帧调用此函数，但这可能会影响统计信息。

此函数不能在IRQ上下文中调用。将此函数的调用同步对单个硬件。不能将此函数的调用，ieee80211\_tx\_status\_ni（）和ieee80211\_tx\_status\_irqsafe（）混合用于单个硬件。不能同时运行ieee80211\_rx（）或ieee80211\_rx\_ni（）。

void ieee80211\_tx\_status\_ni（struct ieee80211\_hw \* hw，struct sk\_buff \* skb）

传输状态回调（在进程上下文中）

1）参数

struct ieee80211\_hw \* hw

传输帧的硬件

struct sk\_buff \* skb

已传输的帧，在此调用后由mac80211拥有

2）说明

像ieee80211\_tx\_status（）一样，但可以在进程上下文中调用。

不能将此函数的调用，ieee80211\_tx\_status（）和ieee80211\_tx\_status\_irqsafe（）混合使用于单个硬件。

**void ieee80211\_tx\_status\_irqsafe（struct ieee80211\_hw \* hw，struct sk\_buff \* skb）**

中断安全的传输状态回调

1）参数

struct ieee80211\_hw \* hw

传输帧的硬件

struct sk\_buff \* skb

已传输的帧，在此调用后由mac80211拥有

2）说明

像ieee80211\_tx\_status（）一样，但可以在IRQ上下文中调用（内部将延迟到任务let）。

不能将此函数的调用，ieee80211\_tx\_status（）和ieee80211\_tx\_status\_ni（）混合使用于单个硬件。

**void ieee80211\_rts\_get（struct ieee80211\_hw \* hw，struct ieee80211\_vif \* vif，const void \* frame，size\_t frame\_len，const struct ieee80211\_tx\_info \* frame\_txctl，struct ieee80211\_rts \* rts）**

RTS帧生成函数

1）参数

struct ieee80211\_hw \* hw

从ieee80211\_alloc\_hw（）中获取的指针。

struct ieee80211\_vif \* vif

从add\_interface回调中获取的struct ieee80211\_vif指针。

const void \* frame

指向即将受到RTS保护的框架的指针。

size\_t frame\_len

帧长度（以八位为单位）。

const struct ieee80211\_tx\_info \* frame\_txctl

结构体 ieee80211\_tx\_info 的帧。

struct ieee80211\_rts \*rts

存储 RTS 帧的缓冲区。

2）说明

如果 RTS 帧是由主机系统生成的（即不在硬件/固件中），则低级驱动程序使用此函数从 802.11 代码接收下一个 RTS 帧。低级别负责在需要 RTS 帧之前调用此函数。

**\_\_le16 ieee80211\_rts\_duration(struct ieee80211\_hw \*hw，struct ieee80211\_vif \*vif，size\_t frame\_len，const struct ieee80211\_tx\_info \*frame\_txctl)**

获取 RTS 帧的持续时间字段

1）参数

struct ieee80211\_hw \*hw

从 ieee80211\_alloc\_hw() 获取的指针。

struct ieee80211\_vif \*vif

来自 add\_interface 回调的 struct ieee80211\_vif 指针。

size\_t frame\_len

要由 RTS 保护的帧的长度。

const struct ieee80211\_tx\_info \*frame\_txctl

帧的 struct ieee80211\_tx\_info。

2）说明

如果 RTS 在固件中生成，但主机系统必须提供持续时间字段，则低级驱动程序使用此函数以 little-endian 字节顺序接收持续时间字段值。

3）返回

持续时间。

**void ieee80211\_ctstoself\_get(struct ieee80211\_hw \*hw，struct ieee80211\_vif \*vif，const void \*frame，size\_t frame\_len，const struct ieee80211\_tx\_info \*frame\_txctl，struct ieee80211\_cts \*cts)**

CTS-to-self 帧生成函数

1）参数

struct ieee80211\_hw \*hw

从 ieee80211\_alloc\_hw() 获取的指针。

struct ieee80211\_vif \*vif

来自 add\_interface 回调的 struct ieee80211\_vif 指针。

const void \*frame

指向要由 CTS-to-self 保护的帧的指针。

size\_t frame\_len

帧长度（以八位字节为单位）。

const struct ieee80211\_tx\_info \*frame\_txctl

帧的 struct ieee80211\_tx\_info。

struct ieee80211\_cts \*cts

存储 CTS-to-self 帧的缓冲区。

2）说明

如果 CTS-to-self 帧由主机系统生成（即不在硬件/固件中），则低级驱动程序使用此函数从 802.11 代码接收下一个 CTS-to-self 帧。低级别负责在需要 CTS-to-self 帧之前调用此函数。

**\_\_le16 ieee80211\_ctstoself\_duration(struct ieee80211\_hw \*hw，struct ieee80211\_vif \*vif，size\_t frame\_len，const struct ieee80211\_tx\_info \*frame\_txctl)**

获取 CTS-to-self 帧的持续时间字段

1）参数

struct ieee80211\_hw \*hw

从 ieee80211\_alloc\_hw() 获取的指针。

struct ieee80211\_vif \*vif

来自 add\_interface 回调的 struct ieee80211\_vif 指针。

size\_t frame\_len

要由 CTS-to-self 保护的帧的长度。

const struct ieee80211\_tx\_info \*frame\_txctl

帧的 struct ieee80211\_tx\_info。

2）说明

如果 CTS-to-self 在固件中生成，但主机系统必须提供持续时间字段，则低级驱动程序使用此函数以 little-endian 字节顺序接收持续时间字段值。

3）返回

持续时间。

**\_\_le16 ieee80211\_generic\_frame\_duration(struct ieee80211\_hw \*hw，struct ieee80211\_vif \*vif，enum nl80211\_band band，size\_t frame\_len，struct ieee80211\_rate \*rate)**

计算帧的持续时间字段

1）参数

struct ieee80211\_hw \*hw

从 ieee80211\_alloc\_hw() 获取的指针。

struct ieee80211\_vif \*vif

来自 add\_interface 回调的 struct ieee80211\_vif 指针。

enum nl80211\_band band

要在其中计算帧持续时间的带。

size\_t frame\_len

帧的长度。

struct ieee80211\_rate \*rate

帧将被传输的速率。

2）说明

计算某些通用帧的持续时间字段，给出其长度和传输速率（以 100 kbps 为单位）。

3）返回

持续时间。

**void ieee80211\_wake\_queue(struct ieee80211\_hw \*hw，int queue)**

唤醒特定队列

1）参数

struct ieee80211\_hw \*hw

从 ieee80211\_alloc\_hw() 获取的指针。

int queue

队列编号（从零开始计数）。

2）说明

驱动程序应使用此函数而不是 netif\_wake\_queue。

**void ieee80211\_wake\_queues(struct [ieee80211\_hw](https://www.kernel.org/doc/html/v4.19/driver-api/80211/mac80211.html" \l "c.ieee80211_hw" \o "ieee80211_hw) \* hw)**

唤醒所有队列

1）参数

struct ieee80211\_hw \* hw

从获得的指针[ieee80211\_alloc\_hw()](https://www.kernel.org/doc/html/v4.19/driver-api/80211/mac80211.html" \l "c.ieee80211_alloc_hw" \o "ieee80211_alloc_hw)。

2）说明

驱动程序应该使用此函数而不是 netif\_wake\_queue。

**void ieee80211\_stop\_queues(struct ieee80211\_hw \*hw)**

停止所有队列

1）参数

struct ieee80211\_hw \*hw

从 ieee80211\_alloc\_hw() 获取的指针。

2）说明

驱动程序应使用此函数而不是 netif\_stop\_queue。

**void ieee80211\_stop\_queue(struct ieee80211\_hw \*hw，int queue)**

停止特定队列

1）参数

struct ieee80211\_hw \*hw

从 ieee80211\_alloc\_hw() 获取的指针。

int queue

队列编号（从零开始计数）。

2）说明

驱动程序应使用此函数而不是 netif\_stop\_queue。

**int ieee80211\_queue\_stopped(struct ieee80211\_hw \*hw，int queue)**

测试队列的状态

1）参数

struct ieee80211\_hw \*hw

从 ieee80211\_alloc\_hw() 获取的指针。

int queue

队列编号（从零开始计数）。

2）说明

驱动程序应使用此函数而不是 netif\_stop\_queue。

3）返回

如果队列已停止，则为 true。否则为 false。

### 帧过滤

mac80211 需要看到许多管理帧以进行正确的操作，并且在监听模式下，用户可能希望看到更多的帧。但是，为了实现最佳 CPU 使用率和功耗，尽可能少地让帧穿过堆栈是可取的。因此，硬件应尽可能过滤。

为了实现这一点，mac80211 使用过滤标志（参见下文）告诉驱动程序的 configure\_filter() 函数哪些帧应传递到 mac80211，哪些应被过滤掉。

在调用 configure\_filter() 之前，将使用 mc\_count 和 mc\_list 参数调用 prepare\_multicast() 回调，这是所有虚拟接口的组播地址列表的组合。它的使用是可选的，并返回一个传递给 configure\_filter() 的 u64。此外，configure\_filter() 具有更改的标志参数，指示更改了哪些标志，并具有新标志状态的 total\_flags。

如果您的设备没有组播地址过滤器，则驱动程序需要检查FIF\_ALLMULTI标志和mc\_count参数，以确定是否应接受或丢弃组播帧。总共\_flags中的所有不受支持的标志都必须清除。如果硬件无法将帧传输到堆栈，则不支持标志。否则，驱动程序必须忽略标志，但不清除标志。如果无法将数据包类型传递到堆栈（因此硬件始终过滤它），则只能清除标志（向mac80211宣布不支持该标志）。例如，如果您的硬件始终过滤控制帧，则应清除FIF\_CONTROL。如果您的硬件始终将控制帧传递到内核并且无法对其进行过滤，则不清除FIF\_CONTROL标志。此规则也适用于所有其他FIF标志。

#### ieee80211\_filter\_flags枚举

硬件过滤标志

**常量**

FIF\_ALLMULTI

传递所有组播帧，如果用户请求或硬件无法按组播地址进行过滤，则使用此选项。

FIF\_FCSFAIL

传递带有失败FCS的帧（但您需要为它们设置RX\_FLAG\_FAILED\_FCS\_CRC）

FIF\_PLCPFAIL

传递带有失败PLCP CRC的帧（但您需要为它们设置RX\_FLAG\_FAILED\_PLCP\_CRC）

FIF\_BCN\_PRBRESP\_PROMISC

在扫描期间设置此标志，以向硬件指示它不应按BSSID过滤信标或探测响应。过滤它们可以大大减少mac80211所需的处理量和CPU唤醒次数，因此如果可能，您应该尊重此标志。

FIF\_CONTROL

传递发送到该站点的控制帧（除了PS Poll之外）

FIF\_OTHER\_BSS

传递到其他BSS的帧

FIF\_PSPOLL

传递PS Poll帧

FIF\_PROBE\_REQ

传递探测请求帧

FIF\_MCAST\_ACTION

传递多播Action帧

**说明**

这些标志确定硬件中的过滤器应该如何进行编程以允许通过以及不应传递给堆栈的内容。传递比请求的框架更安全，但对功耗有负面影响。

### mac80211工作队列

mac80211为驱动程序和内部mac80211使用提供了自己的工作队列。工作队列是单线程的工作队列，只能由助手进行访问以进行健全性检查。驱动程序必须确保在停止（）回调上取消对mac80211工作队列的所有工作添加。

mac80211将在接口删除和挂起期间刷新工作队列。

在mac80211工作队列中执行的所有工作都不能获取RTNL锁。

#### void ieee80211\_queue\_work（struct ieee80211\_hw \* hw，struct work\_struct \* work）

将工作添加到mac80211工作队列中

**参数**

struct ieee80211\_hw \* hw

我们要为其添加工作的接口的硬件结构

struct work\_struct \* work

我们想要添加到mac80211工作队列中的工作

**说明**

驱动程序和mac80211使用此功能将工作添加到mac80211工作队列中。该助手确保驱动程序在不应该排队工作时不排队工作。

#### void ieee80211\_queue\_delayed\_work（struct ieee80211\_hw \* hw，struct delayed\_work \* dwork，unsigned long delay）

将延迟工作添加到mac80211工作队列中

**参数**

struct ieee80211\_hw \* hw

我们要为其添加工作的接口的硬件结构

struct delayed\_work \* dwork

要排队到mac80211工作队列中的可延迟工作

unsigned long delay

在排队之前等待的jiffies数

**说明**

驱动程序和mac80211使用此来将延迟工作排队到mac80211工作队列中。

## mac80211子系统（高级）

本书的这一部分中包含的信息仅适用于mac80211和驱动程序的高级交互，以利用更多的硬件功能并提高性能。

### LED支持

Mac80211支持各种方式的闪烁LED。在可能的情况下，设备LED应该被公开为LED类设备并连接到适当的触发器，然后由mac80211适当触发触发器。

#### const char \* ieee80211\_get\_tx\_led\_name（struct ieee80211\_hw \* hw）

获取TX LED的名称

**参数**

struct ieee80211\_hw \* hw

要为其获取LED触发器名称的硬件

**说明**

mac80211为每个无线硬件创建一个传输LED触发器，如果您的驱动程序注册了LED设备，则可以用于驱动LED。此函数返回触发器的名称（或如果未配置为LED，则为NULL），以便您可以自动链接LED设备。

**返回**

LED触发器的名称。如果未配置为LED，则为NULL。

#### const char \* ieee80211\_get\_rx\_led\_name（struct ieee80211\_hw \* hw）

获取RX LED的名称

**参数**

struct ieee80211\_hw \* hw

要为其获取LED触发器名称的硬件

**说明**

mac80211为每个无线硬件创建接收LED触发器，如果您的驱动程序注册了LED设备，则可以用于驱动LED。此函数返回触发器的名称（或如果未配置为LED，则为NULL），以便您可以自动链接LED设备。

**返回**

LED触发器的名称。如果未配置为LED，则为NULL。

#### const char \*ieee80211\_get\_assoc\_led\_name（struct ieee80211\_hw \*hw）

获取关联LED的名称

**参数**

struct ieee80211\_hw \* hw

要获取LED触发器名称的硬件。

**说明**

mac80211为每个无线硬件创建一个关联LED触发器，如果您的驱动程序注册了LED设备，则可以用于驱动LED。此函数返回触发器的名称（如果未配置为LED，则返回NULL），以便您可以自动连接LED设备。

**返回**

LED触发器的名称。如果未配备LED，则为空。

#### const char \*ieee80211\_get\_radio\_led\_name（struct ieee80211\_hw \*hw）

获取无线电LED的名称

**参数**

struct ieee80211\_hw \* hw

要获取LED触发器名称的硬件。

**说明**

mac80211为每个无线硬件创建一个无线电更改LED触发器，如果您的驱动程序注册了LED设备，则可以用于驱动LED。此函数返回触发器的名称（如果未配置为LED，则返回NULL），以便您可以自动连接LED设备。

**返回**

LED触发器的名称。如果未配备LED，则为空。

#### struct ieee80211\_tpt\_blink

吞吐量闪烁说明

**定义**

struct ieee80211\_tpt\_blink {

int throughput;

int blink\_time;

};

**成员**

throughput

以千比特/秒为单位的吞吐量

blink\_time

以毫秒为单位的闪烁时间（完整周期，即一次关闭 + 一次开启）

#### enum ieee80211\_tpt\_led\_trigger\_flags

吞吐量触发标志

**常量**

IEEE80211\_TPT\_LEDTRIG\_FL\_RADIO

启用与无线电的闪烁

IEEE80211\_TPT\_LEDTRIG\_FL\_WORK

在工作时启用闪烁

IEEE80211\_TPT\_LEDTRIG\_FL\_CONNECTED

在某种方式下连接至少一个接口（包括成为AP的接口）时启用闪烁

#### const char \*ieee80211\_create\_tpt\_led\_trigger（struct ieee80211\_hw \* hw，unsigned int flags，const struct ieee80211\_tpt\_blink \* blink\_table，unsigned int blink\_table\_len）

创建吞吐量LED触发器

**参数**

struct ieee80211\_hw \* hw

要创建触发器的硬件

unsigned int flags

触发标志，请参见枚举ieee80211\_tpt\_led\_trigger\_flags

const struct ieee80211\_tpt\_blink \* blink\_table

闪烁表-需要按吞吐量排序

unsigned int blink\_table\_len

闪烁表的大小

**返回**

NULL（在出现错误或未配置LED触发器的情况下）或新触发器的名称。

**注意**

必须在ieee80211\_register\_hw（）之前调用此函数。

### 硬件加速加密

mac80211能够利用许多硬件加速设计进行加密和解密操作。

给定设备的结构ieee80211\_ops中的set\_ key（）回调被调用以启用加密和解密的硬件加速。回调采用sta参数，该参数对于默认密钥或仅用于传输的密钥将为NULL，或针对单个密钥的对等站点信息。当为访问点配置VLAN时，可以使用具有相同密钥索引的多个传输密钥。

在传输时，TX控制数据将使用驱动程序选择的hw\_key\_idx，通过修改key参数指向的struct ieee80211\_key\_conf来set\_key（）函数。

SET\_KEY命令的set\_key（）调用应返回0，如果现在正在使用该密钥，则为-EOPNOTSUPP或-ENOSPC；如果您返回0，则必须将hw\_key\_idx分配给硬件密钥索引，即您可以使用完整的u8范围。

请注意，如果设置了IEEE80211\_HW\_SW\_CRYPTO\_CONTROL标志，则mac80211不会在启用硬件加密失败时自动切换到软件加密。 set\_key（）调用也可以返回1，以允许使用软件完成此特定钥匙/算法。

当cmd为DISABLE\_KEY时，它必须成功。

请注意，即使已将密钥上传到硬件，也可以选择不解密帧，堆栈不会基于密钥是否已上传而做出任何决策，而是基于接收标志而做出决策。

由键参数指向的struct ieee80211\_key\_conf结构保证在set\_key（）删除它之前有效，但它只能用作区分密钥的cookie。

在TKIP中，有些HW需要提供相位1密钥，用于RX解密加速（即iwlwifi）。这些驱动程序应提供update\_tkip\_key处理程序。update\_tkip\_key（）调用使用新的相位1密钥更新驱动程序。每当iv16绕回（每65536个数据包）时，就会发生这种情况。 set\_key（）调用将仅发生一次，用于每个密钥（除非AP进行了重新密钥），它不包括有效的相位1密钥。仅由update\_tkip\_key提供有效的第1相位密钥。触发器使mac80211调用此处理程序是具有iv16的软件解密。

set\_default\_unicast\_key（）调用将默认的WEP密钥索引更新为WEP加密类型所配置的硬件。对于支持数据包卸载的设备（例如ARP响应），这是必需的。

#### enum set\_key\_cmd

**常量**

SET\_KEY

设置了一把键

DISABLE\_KEY

必须禁用一把密钥

**说明**

与struct ieee80211\_ops中的set\_key()回调一起使用，指示是否正在添加或删除一把密钥。

#### struct ieee80211\_key\_conf

密钥信息

**定义**

struct ieee80211\_key\_conf {

atomic64\_t tx\_pn;

u32 cipher;

u8 icv\_len;

u8 iv\_len;

u8 hw\_key\_idx;

s8 keyidx;

u16 flags;

u8 keylen;

u8 key[];

};

**成员**

tx\_pn

用于TX键的PN，如果需要自己进行软件PN分配（例如，由于TSO而需要），驱动程序也可以使用它。

cipher

该密钥的加密套件选择器。

icv\_len

此密钥类型的ICV长度

iv\_len

此密钥类型的IV长度

hw\_key\_idx

由驱动程序设置，这是驱动程序希望在传输帧并需要在硬件中进行加密时给出的键索引。

keyidx

密钥索引（0-3）

flags

键标志，请参阅enum ieee80211\_key\_flags。

keylen

密钥材料长度

key

密钥材料。对于ALG\_TKIP，密钥编码为256位（32字节）数据块：-暂时加密密钥（128位）-暂时认证器Tx MIC密钥（64位）-暂时认证器Rx MIC密钥（64位）

**说明**

mac80211通过struct ieee80211\_ops中的set\_key()回调向驱动程序提供此密钥信息。

#### enum ieee80211\_key\_flags

密钥标志

**常量**

IEEE80211\_KEY\_FLAG\_GENERATE\_IV\_MGMT

驱动程序应该为CCMP/GCMP键设置此标志，以指示仅对管理帧（MFP）生成IV。

IEEE80211\_KEY\_FLAG\_GENERATE\_IV

驱动程序应该设置此标志，以指示需要为此特定密钥生成IV。设置此标志并不一定意味着SKB将具有足够的尾部空间用于ICV或MIC。

IEEE80211\_KEY\_FLAG\_GENERATE\_MMIC

如果TKIP密钥需要在软件中进行Michael MIC生成，则驱动程序应该设置此标志。

IEEE80211\_KEY\_FLAG\_PAIRWISE

由mac80211设置，此标志指示密钥是成对的而不是共享的。

IEEE80211\_KEY\_FLAG\_SW\_MGMT\_TX

如果需要在软件中对管理帧（MFP）进行CCMP/GCMP加密，则驱动程序应该为CCMP/GCMP密钥设置此标志。

IEEE80211\_KEY\_FLAG\_PUT\_IV\_SPACE

如果应准备IV的空间，但不应生成IV本身，则驱动程序应将此标志设置为。不要在同一键上同时设置IEEE80211\_KEY\_FLAG\_GENERATE\_IV。设置此标志并不一定意味着SKB将具有足够的尾部空间用于ICV或MIC。

IEEE80211\_KEY\_FLAG\_RX\_MGMT

此密钥将用于解密接收到的管理帧。该标志可以帮助具有硬件加密实现但不能正确处理管理帧的驱动程序，使其可以不将密钥上传到硬件并退回到软件加密。请注意，此标志仅处理RX，如果您的加密引擎无法处理TX，则还可以设置IEEE80211\_KEY\_FLAG\_SW\_MGMT\_TX标志以在SW中加密此类帧。

IEEE80211\_KEY\_FLAG\_RESERVE\_TAILROOM

驱动程序应该为密钥设置此标志，以指示始终必须保留足够的尾部空间用于ICV或MIC，即使启用了HW加密。

IEEE80211\_KEY\_FLAG\_PUT\_MIC\_SPACE

如果TKIP密钥仅需要MIC空间，则驱动程序应为该密钥设置此标志。不要在同一密钥上同时设置IEEE80211\_KEY\_FLAG\_GENERATE\_MMIC。

**说明**

在struct ieee80211\_key\_conf的标志参数中，这些标志用于驱动程序与mac80211之间的关键通信。

#### void ieee80211\_get\_tkip\_p1k(struct ieee80211\_key\_conf \*keyconf, struct sk\_buff \*skb, u16 \*p1k)

获取TKIP阶段1密钥

**参数**

struct ieee80211\_key\_conf \*keyconf

传递设置密钥的参数

struct sk\_buff \*skb

将从中获取IV32值并将加密该P1K的数据包

u16 \*p1k

将写入密钥的缓冲区，为5个u16值

**说明**

此函数返回从给定包获取的IV32的TKIP阶段1密钥。

#### void ieee80211\_get\_tkip\_p1k\_iv(struct ieee80211\_key\_conf \*keyconf, u32 iv32, u16 \*p1k)

获取用于IV32的TKIP阶段1密钥

**参数**

struct ieee80211\_key\_conf \*keyconf

传递设置密钥的参数

u32 iv32

要获取P1K的IV32

u16 \*p1k

将写入密钥的缓冲区，为5个u16值

**说明**

此函数返回给定IV32的TKIP阶段1密钥。

#### void ieee80211\_get\_tkip\_p2k(struct ieee80211\_key\_conf \*keyconf, struct sk\_buff \*skb, u8 \*p2k)

获取TKIP阶段2密钥

**参数**

struct ieee80211\_key\_conf \*keyconf

传递设置密钥的参数

struct sk\_buff \*skb

要从中获取IV32 / IV16值并使用此密钥加密的数据包

u8 \*p2k

将写入密钥的缓冲区，为16个字节

**说明**

此函数计算数据包中IV值的TKIP RC4密钥。

### 省电支持

mac80211 支持各种节电实现。

首先，它可以支持通过硬件自己处理所有电源管理的硬件，这种硬件应该简单地设置 IEEE80211\_HW\_SUPPORTS\_PS 硬件标志。在这种情况下，根据关联状态，它将被告知所需的节能模式与 IEEE80211\_CONF\_PS 标志。硬件必须在必要时发送 nullfunc 帧，即在进入和离开节电模式时。硬件需要查看信标中的 AID，并在发现流量定向到它时向 AP 发送唤醒信号。

启用 IEEE80211\_CONF\_PS 标志意味着启用了 IEEE 802.11-2007 第11.2节中定义的节省电源模式。这不应与硬件唤醒和睡眠状态混淆。驱动程序负责在向硬件发出命令之前唤醒硬件，并在适当的时候让其进入睡眠状态。

启用 PS 后，硬件需要唤醒信标并在信标后接收缓冲的多播/广播帧。此外，它必须能够发送帧并接收确认帧。

其他硬件设计无法自行发送 nullfunc 帧，还需要软件支持以解析 TIM 位图。mac80211 通过组合 IEEE80211\_HW\_SUPPORTS\_PS 和 IEEE80211\_HW\_PS\_NULLFUNC\_STACK 标志也支持此功能。当然，硬件仍然需要传递信标。硬件仍然需要处理唤醒多播流量；如果不能，则驱动程序必须尽可能处理该问题，mac80211 太慢了，无法处理。

动态节电是常规节电的扩展，其中硬件在发送帧后保持唤醒状态一段用户指定的时间，以便不必缓冲并因此延迟到下一次唤醒的答复帧。它的妥协是在有数据流量时获得足够好的延迟，同时在空闲期间节省大量电力。

mac80211 只需根据流量启用和禁用 PS 即可简单地支持动态节电。驱动程序只需要设置 IEEE80211\_HW\_SUPPORTS\_PS 标志，mac80211 将自动处理所有事项。此外，支持动态 PS 特性的硬件可以设置 IEEE80211\_HW\_SUPPORTS\_DYNAMIC\_PS 标志，以指示它可以自己支持动态 PS 模式。驱动程序需要查看 dynamic\_ps\_timeout 硬件配置值，并在设置 IEEE80211\_CONF\_PS 时使用该值。在这种情况下，mac80211 将在堆栈中禁用动态 PS 功能，并在用户启用电源管理时仅保持 IEEE80211\_CONF\_PS 已启用。

驱动程序通过启用 IEEE80211\_VIF\_SUPPORTS\_UAPSD 标志来通知 U-APSD 客户端支持。通过 conf\_tx（）操作中的 UAPSD 参数配置模式。硬件需要发送 QoS Nullfunc 帧并保持唤醒状态直到服务期结束。为了利用 U-APSD，禁用 voip AC 的动态节电，所有来自该 AC 的帧都在启用电源管理的情况下传输。

注意：IEEE80211\_HW\_PS\_NULLFUNC\_STACK 尚不支持 U-APSD 客户端模式。

### 信标过滤器支持

有些硬件具有信标过滤器支持，可以减少主机 CPU 的唤醒，从而减少系统功耗。它通常的工作方式是固件创建信标的校验和，但省略所有不断变化的元素（TSF、TIM 等）。每当校验和更改时，信标就会转发到主机，否则它就会被丢弃。这样，主机只会接收到某些相关信息已经更改的信标（例如 ERP 保护或 WMM 设置）。

信标过滤器支持由 IEEE80211\_VIF\_BEACON\_FILTER 接口能力进行广告宣传。每当启用节能时，即设置了 IEEE80211\_CONF\_PS，驱动程序需要启用信标过滤器支持。当启用了节能时，堆栈不会检查信标丢失，驱动程序需要通过 ieee80211\_beacon\_loss() 通知信标丢失。

将来，mac80211 和漫游算法将控制固件通知驱动程序信标丢失事件（或错过的信标数）。因为可能有不断变化的信息元素，但软件堆栈中没有关心的内容，所以将来，mac80211 将告诉驱动程序哪些信息元素是有趣的，我们想在其中看到它们的更改。这将包括：

信息元素 ID 列表

供应商信息元素的 OUI 列表

理想情况下，硬件将过滤掉没有更改请求元素的信标，但是如果不能支持它，它可以牺牲一些效率，仅过滤出一部分。例如，如果设备不支持检查 OUIs，它应该通过所有供应商信息元素中的所有更改传递。

请注意，为了简化，改变也包括信标中出现或消失的信息元素。

有些硬件支持“忽略列表”，只需确保没有请求的内容在忽略列表中，并将常见的变化的信息元素ID放入忽略列表中，例如11（BSS负载）和各种供应商分配的具有未知内容的IE（128、129、133-136、149、150、155、156、173、176、178、179、219）；为了前向兼容性，也可以包括一些目前未使用的ID。

除了这些功能外，硬件还应支持通知主机信标RSSI的变化。这与在没有流量流动时实现漫游有关（当流量流动时，我们看到接收到的数据包的RSSI）。这可能包括在RSSI显着变化时或在可配置的阈值以下或以上时通知主机。未来，这些阈值还将由mac80211配置（从用户空间获取）以根据漫游算法实现它们。

如果硬件无法实现此功能，则驱动程序应要求其定期将信标帧传递给主机，以便软件可以进行信号强度阈值检查。

#### void ieee80211\_beacon\_loss（struct ieee80211\_vif \* vif）

通知硬件未接收到信标

**参数**

struct ieee80211\_vif \* vif

从add\_interface回调的struct ieee80211\_vif指针。

**说明**

当启用IEEE80211\_VIF\_BEACON\_FILTER和设置IEEE80211\_CONF\_PS的信标过滤时，驱动程序需要使用此函数通知硬件何时未接收信标。

### 多个队列和QoS支持

待定

#### struct ieee80211\_tx\_queue\_params

传输队列配置

**定义**

struct ieee80211\_tx\_queue\_params {

u16 txop;

u16 cw\_min;

u16 cw\_max;

u8 aifs;

bool acm;

bool uapsd;

bool mu\_edca;

struct ieee80211\_he\_mu\_edca\_param\_ac\_rec mu\_edca\_param\_rec;

};

**成员**

txop

最大突发时间，以32微秒为单位，0表示禁用

cw\_min

最小争用窗口[2 ^ n-1形式的值，在1..32767范围内]

cw\_max

最大争用窗口[如cw\_min]

aifs

仲裁间隔时间[0..255]

acm

是否需要访问类别的强制承认控制

uapsd

队列中U-APSD模式是否启用

mu\_edca

MU EDCA是否配置

mu\_edca\_param\_rec

用于HE的MU EDCA参数记录

**说明**

此结构中提供的信息是QoS传输队列配置所需的。参见IEEE 802.11 7.3.2.29。

### 支持接入点模式

待定

应在这里或硬件加密章节中插入有关带有VLAN接口的说明。

#### 支持省电客户端

为了实现AP和P2P GO模式，mac80211支持客户端省电，包括“传统”PS（PS-Poll / null数据）和uAPSD。目前没有支持sAPSD。

有一项mac80211的假设，即客户端不会同时使用PS-Poll和uAPSD进行轮询触发。二者均受支持，同一客户端均可使用，但同一客户端不能同时使用。这简化了驱动程序代码。

要记住的第一件事是，完整的驱动程序实现有一个标志：IEEE80211\_HW\_AP\_LINK\_PS。如果设置了此标志，则mac80211期望驱动程序处理大多数省电客户端的状态机，并将忽略传入帧中的PM位。然后，驱动程序使用ieee80211\_sta\_ps\_transition（）通知mac80211站点的省电转换。在此模式下，mac80211也不处理PS-Poll / uAPSD。

在没有IEEE80211\_HW\_AP\_LINK\_PS的模式下，mac80211将检查传入帧中的PM位以进行客户端省电转换。当一个站台休眠时，我们将停止向其传输。但是，存在一种竞争条件：当硬件队列上有数据缓冲时，可能会有一个站点进入休眠状态。如果设备支持此操作，它将拒绝帧，并使驱动程序将帧返回给带有设置IEEE80211\_TX\_STAT\_TX\_FILTERED标志的mac80211，这将导致mac80211在该站点唤醒时重试帧。驱动程序还通过调用其sta\_notify回调来通知省电转换。

当站点处于休眠状态时，它有三个选择：它可以醒来，可以PS-Poll，也可以可能开始一个uAPSD服务期。唤醒是通过简单地将所有缓冲的（和过滤的）帧传输到站点来实现的。这是最简单的情况。当站点发送PS-Poll或uAPSD触发帧时，mac80211将使用allow\_buffered\_frames回调（此回调是可选的）通知驱动程序。然后，mac80211将像往常一样传输帧，并在每个帧上设置IEEE80211\_TX\_CTL\_NO\_PS\_BUFFER。服务期中的最后一帧（或PS-Poll的唯一响应）还将设置IEEE80211\_TX\_STATUS\_EOSP，以指示它结束服务期。由于此帧必须具有TX状态报告，因此它还设置IEEE80211\_TX\_CTL\_REQ\_TX\_STATUS。当为此帧报告TX状态时，将标记服务期已结束，并且对等方可以启动新的服务期。

此外，当mac80211设置了IEEE80211\_TX\_CTL\_NO\_PS\_BUFFER时，mac80211还可以发送不可缓存MMPDU。

在一些设备（例如iwlwifi）上可能会存在另一种竞争条件，即当某个站点排队的帧在其唤醒或轮询时，已排队的帧可能会首先被传输，导致EOSP的重新排序和/或错误处理。造成这个问题的原因是允许将帧传输到某个站点是设备的带外通信。为了解决这个问题，驱动程序可以在接收到站点进入休眠通知时调用ieee80211\_sta\_block\_awake()，如果有帧被缓冓，则必须调用该函数来指示该站点不再被阻止。当所有这些帧都已被过滤（参见上文），它必须再次调用该函数以指示该站点不再受阻。

如果驱动程序以任何方式在驱动程序中缓冲聚合帧，则必须在收到有关该站点进入睡眠状态的通知时使用ieee80211\_sta\_set\_buffered()调用来通知mac80211缓冲任何TID上有帧缓冲的信息。请注意，当一个站点唤醒时，这些信息会被重置（因此在通知该站点进入睡眠时需要调用该函数）。然后，在任何原因时开始服务期间，则使用被释放的帧数和它们来自哪些TID的调用release\_buffered\_frames。在这种情况下，驱动程序负责设置EOSP（对于uAPSD）和MORE\_DATA位于释放的帧中，以帮助传递more\_data参数告诉驱动程序在其他TID上是否有更多数据 - 忽略要释放帧的TID，因为mac80211不知道这些TID的缓冲区中包含多少帧。

如果驱动程序还实现了GO模式，那么缺席期可能会缩短服务期间（或中止PS-Poll响应），除了驱动程序中缓冲的帧之外，它必须过滤响应帧-那些必须保持缓冲以避免重新排序。因为在这种情况下可能不会释放任何帧，所以驱动程序必须调用ieee80211\_sta\_eosp()以指示mac80211服务期间已结束。

最后，如果来自多个TID的帧已从mac80211释放，但驱动程序可能会对它们进行重新排序，则必须清除并适当设置标志（只有最后一帧可能具有IEEE80211\_TX\_STATUS\_EOSP），并且还要处理帧中的EOSP和MORE\_DATA位。在这种情况下，驱动程序也可以使用ieee80211\_sta\_eosp()。

请注意，如果驱动程序缓冲除QoS数据帧以外的帧，则必须注意永远不要将非QoS数据帧作为服务期间的最后一帧发送，如果需要，请在非QoS数据帧之后添加QoS-无数据帧。

**struct [sk\_buff](https://www.kernel.org/doc/html/v4.19/networking/kapi.html" \l "c.sk_buff" \o "sk_buff) \* ieee80211\_get\_buffered\_bc(struct [ieee80211\_hw](https://www.kernel.org/doc/html/v4.19/driver-api/80211/mac80211.html" \l "c.ieee80211_hw" \o "ieee80211_hw) \* hw, struct [ieee80211\_vif](https://www.kernel.org/doc/html/v4.19/driver-api/80211/mac80211.html" \l "c.ieee80211_vif" \o "ieee80211_vif) \* vif)**

访问缓冲的广播和多播帧

1）参数

struct ieee80211\_hw \* hw

从获得的指针[ieee80211\_alloc\_hw()](https://www.kernel.org/doc/html/v4.19/driver-api/80211/mac80211.html" \l "c.ieee80211_alloc_hw" \o "ieee80211_alloc_hw)。

struct ieee80211\_vif \* vif

[struct ieee80211\_vif](https://www.kernel.org/doc/html/v4.19/driver-api/80211/mac80211.html" \l "c.ieee80211_vif" \o "ieee80211_vif)来自 add\_interface 回调的指针。

2）说明

用于访问缓冲的广播和多播帧的函数。如果在使用节能时硬件/固件没有实现广播/多播帧的缓冲，802.11 代码会将它们缓冲在主机内存中。低级驱动程序使用此函数来获取下一个缓冲帧。在大多数情况下，这在生成信标帧时使用。

3）返回

指向下一个缓冲 skb 的指针，如果没有更多缓冲帧可用，则为 NULL。

4）笔记

缓冲帧仅在生成 DTIM 信标帧后返回[ieee80211\_beacon\_get()](https://www.kernel.org/doc/html/v4.19/driver-api/80211/mac80211-advanced.html" \l "c.ieee80211_beacon_get" \o "ieee80211_beacon_get)，因此低级驱动程序必须[ieee80211\_beacon\_get()](https://www.kernel.org/doc/html/v4.19/driver-api/80211/mac80211-advanced.html" \l "c.ieee80211_beacon_get" \o "ieee80211_beacon_get)首先调用。[ieee80211\_get\_buffered\_bc()](https://www.kernel.org/doc/html/v4.19/driver-api/80211/mac80211-advanced.html" \l "c.ieee80211_get_buffered_bc" \o "ieee80211_get_buffered_bc)如果先前生成的信标不是 DTIM，则返回 NULL，因此低级驱动程序不需要单独检查 DTIM 信标，并且应该能够对所有信标使用通用代码。

**struct [sk\_buff](https://www.kernel.org/doc/html/latest/networking/kapi.html" \l "c.sk_buff" \o "sk_buff) \*ieee80211\_beacon\_get(struct [ieee80211\_hw](https://www.kernel.org/doc/html/latest/driver-api/80211/mac80211.html" \l "c.ieee80211_hw" \o "ieee80211_hw) \*hw, struct [ieee80211\_vif](https://www.kernel.org/doc/html/latest/driver-api/80211/mac80211.html" \l "c.ieee80211_vif" \o "ieee80211_vif) \*vif, unsigned int link\_id)**

信标生成函数

1）参数

struct ieee80211\_hw \* hw

从获得的指针[ieee80211\_alloc\_hw()](https://www.kernel.org/doc/html/v4.19/driver-api/80211/mac80211.html" \l "c.ieee80211_alloc_hw" \o "ieee80211_alloc_hw)。

struct ieee80211\_vif \* vif

[struct ieee80211\_vif](https://www.kernel.org/doc/html/v4.19/driver-api/80211/mac80211.html" \l "c.ieee80211_vif" \o "ieee80211_vif)来自 add\_interface 回调的指针。

2）说明

看ieee80211\_beacon\_get\_tim()。

3）返回

看ieee80211\_beacon\_get\_tim()。

**void ieee80211\_sta\_eosp(struct [ieee80211\_sta](https://www.kernel.org/doc/html/latest/driver-api/80211/mac80211-advanced.html" \l "c.ieee80211_sta" \o "ieee80211_sta) \*pubsta)**

通知 mac80211 SP 结束

1）参数

struct ieee80211\_sta \* pubsta

车站

2）描述

当设备以无法在 TX 状态中告知 mac80211 EOSP 的方式传输帧时，它必须清除该 IEEE80211\_TX\_STATUS\_EOSP位并改为调用此函数。这适用于 PS-Poll 以及 uAPSD。

请注意，就像 with\_tx\_status()和\_rx()drivers 不能混合调用 irqsafe/non-irqsafe 版本一样，这个函数也不能与那些混合。使用所有的irqsafe，或者所有的非irqsafe，不要混用！

3）注意：这个函数的 \_irqsafe 版本不存在，没有

司机现在需要它。如果您需要 \_irqsafe 版本，请不要调用此函数，查看 git 历史并恢复 \_irqsafe 版本！

**enum ieee80211\_frame\_release\_type**

帧释放原因

常量

IEEE80211\_FRAME\_RELEASE\_PSPOLL

为PS-Poll释放帧

IEEE80211\_FRAME\_RELEASE\_UAPSD

由于触发启用的AC上接收到的帧而释放的帧

**int ieee80211\_sta\_ps\_transition（struct ieee80211\_sta \* sta，bool start）**

连接sta的PS转换

1）参数

struct ieee80211\_sta \* sta

当前连接的sta

bool start

开始或停止PS

2）说明

当在设置IEEE80211\_HW\_AP\_LINK\_PS标志的AP模式下操作时，使用此函数通知mac80211连接的站点进入/离开PS模式。

此函数可能不会在IRQ上下文中调用或启用softirq。

针对单个硬件的对此函数的调用必须相互同步。

3）返回

成功时为0。请求的PS模式已设置时为-EINVAL。

**int ieee80211\_sta\_ps\_transition\_ni（struct ieee80211\_sta \* sta，bool start）**

连接的sta的PS转换（在进程上下文中）

1）参数

struct ieee80211\_sta \* sta

当前连接的sta

bool start

开始或停止PS

2）说明

像ieee80211\_sta\_ps\_transition（）一样，但可以在进程上下文中调用（在内部禁用底部半部）。并发调用限制仍然适用。

3）返回

像ieee80211\_sta\_ps\_transition（）。

**void ieee80211\_sta\_set\_buffered（struct ieee80211\_sta \* sta，u8 tid，bool buffered）**

通知mac80211有关驱动程序缓冲的帧

1）参数

struct ieee80211\_sta \* sta

用于睡眠站点的struct ieee80211\_sta指针

u8 tid

具有缓冲帧的TID

bool buffered

表示是否为此TID缓冲帧

2）说明

如果驱动程序为powersave站点缓冲帧而不是将它们传回mac80211以进行重传，则仍可能需要告知站点有缓冲帧通过TIM位。

此函数通知mac80211是否有驱动程序为给定TID缓冲的帧；然后，mac80211可以使用此数据设置TIM位（注意：这可能会回调到驱动程序的set\_tim调用！请注意锁定！）

如果释放给站点的所有帧（由于PS-poll或uAPSD），则驱动程序需要通知mac80211不再有帧缓冲。但是，当站点唤醒mac80211时，mac80211假定所有缓冲的帧都将被传输，并清除此数据，驱动程序需要确保它们在睡眠转换（sta\_notify（）与STA\_NOTIFY\_SLEEP）期间通知mac80211有关所有缓冲的帧。

请注意，从技术上讲，mac80211只需要知道每个AC而不是每个TID的内容，但由于驱动程序缓冲不可避免地会发生在每个TID上（因为它与聚合相关），因此更容易使mac80211将TID映射到所需的AC而不是在使用此API的所有驱动程序中跟踪。

**void ieee80211\_sta\_block\_awake（struct ieee80211\_hw \* hw，struct ieee80211\_sta \* pubsta，bool block）**

阻止站点唤醒

1）参数

struct ieee80211\_hw \* hw

硬件

struct ieee80211\_sta \* pubsta

车站

bool block

是否阻塞或取消阻塞

2）说明

某些设备要求在将某个睡眠状态的站点的队列中的所有帧刷新之前，无法向该站点提供轮询响应或唤醒后的帧。请注意，这些帧必须被驱动程序拒绝为已过滤状态，具有适当的状态标志。

此函数允许以无竞争的方式实现此模式。

为此，驱动程序必须跟踪特定站点仍排队的帧数。如果该数字在站点进入睡眠状态时不为零，则驱动程序必须调用此函数以强制mac80211将该站点视为已睡眠，而不考虑站点的实际状态。一旦未完成的帧数达到零，则驱动程序必须再次调用此函数以解除阻止。这将使mac80211能够发送ps-poll响应，并且如果站点在此期间查询，则帧也将作为结果发送出去。此外，在站点解除阻止后的某个时间，驱动程序将收到有关站点的唤醒通知，而无论该站点实际上是否在阻止时唤醒。

### 支持多个虚拟接口

TBD

注意：具有相同MAC地址的WDS几乎总是可以接受的

此处插入有关具有不同MAC地址的多个虚拟接口的注释，注明mac80211支持哪些配置，添加有关支持hw crypto的注释。

#### void ieee80211\_iterate\_active\_interfaces（struct ieee80211\_hw \* hw，u32 iter\_flags，void (\* iterator)（void \* data，u8 \* mac，struct ieee80211\_vif \* vif），void \* data）

迭代活动接口

**参数**

struct ieee80211\_hw \* hw

应迭代其中接口的硬件struct

u32 iter\_flags

迭代标志，见枚举 ieee80211\_interface\_iteration\_flags

void (\*iterator)(void \*data, u8 \*mac, struct ieee80211\_vif \*vif)

要调用的迭代器函数

void \*data

迭代器函数的第一个参数

**说明**

该函数在当前活动的与给定硬件关联的接口上进行迭代，并调用它们的回调函数。当迭代器函数是 atomic 时，可以使用 ieee80211\_iterate\_active\_interfaces\_atomic。在 add\_interface()期间不迭代新接口。

#### void ieee80211\_iterate\_active\_interfaces\_atomic(struct ieee80211\_hw \*hw, u32 iter\_flags，void (\*iterator)(void \*data, u8 \*mac, struct ieee80211\_vif \*vif), void \*data)

迭代活动接口

**参数**

struct ieee80211\_hw \*hw

应迭代其接口的硬件结构

u32 iter\_flags

迭代标志，见枚举 ieee80211\_interface\_iteration\_flags

void (\*iterator)(void \*data, u8 \*mac, struct ieee80211\_vif \*vif)

要调用的迭代器函数，不能 sleep

void \*data

迭代器函数的第一个参数

**说明**

该函数在当前活动的与给定硬件相关联的接口上进行迭代，并为它们调用回调。如果不希望使用原子迭代器回调函数，则可以使用 ieee80211\_iterate\_active\_interfaces。在 add\_interface()期间不迭代新接口。

### 站点处理

待办事项

#### struct ieee80211\_sta

站点表条目

**定义**

struct ieee80211\_sta {

u32 supp\_rates[NUM\_NL80211\_BANDS];

u8 addr[ETH\_ALEN];

u16 aid;

struct ieee80211\_sta\_ht\_cap ht\_cap;

struct ieee80211\_sta\_vht\_cap vht\_cap;

struct ieee80211\_sta\_he\_cap he\_cap;

u16 max\_rx\_aggregation\_subframes;

bool wme;

u8 uapsd\_queues;

u8 max\_sp;

u8 rx\_nss;

enum ieee80211\_sta\_rx\_bandwidth bandwidth;

enum ieee80211\_smps\_mode smps\_mode;

struct ieee80211\_sta\_rates \_\_rcu \*rates;

bool tdls;

bool tdls\_initiator;

bool mfp;

u8 max\_amsdu\_subframes;

u16 max\_amsdu\_len;

bool support\_p2p\_ps;

u16 max\_rc\_amsdu\_len;

struct ieee80211\_txq \*txq[IEEE80211\_NUM\_TIDS];

u8 drv\_priv[0] ;

};

**成员**

supp\_rates

支持的速率的位图（每个频段）

addr

MAC地址

aid

如果我们是 AP，则分配给该站点的 AID

ht\_cap

此站点的 HT 能力；仅限于我们自己的能力

vht\_cap

此站点的 VHT 能力；仅限于我们自己的能力

he\_cap

此站点的 HE 能力

max\_rx\_aggregation\_subframes

此站点允许向我们传输的单个 AMPDU 中的最大帧数。可以由驱动程序进行修改。

wme

指示 STA 是否支持 QoS/WME（如果本地设备支持，否则始终为 false）

uapsd\_queues

配置了 uapsd 的队列位图。仅在支持 wme 时有效。位排序如 IEEE80211\_WMM\_IE\_STA\_QOSINFO\_AC\_\*。

max\_sp

最大服务周期。仅在支持 wme 时有效。

rx\_nss

在 HT/VHT 中，站点目前可以接收的最大空间流数，通过操作模式通知和能力改变。该值仅在站点进入关联状态后才有效。

bandwidth

当前站点可接收的带宽

smps\_mode

当前的 SMPS 模式（关闭、静态或动态）

rates

速率控制选择表

tdls

指示 STA 是否为 TDLS 对等方

tdls\_initiator

指示 STA 是否是 TDLS 链接的发起方。仅在首先STA 是 TDLS 对等体时才有效。

mfp

指示 STA 是否使用管理帧保护。

max\_amsdu\_subframes

指示单个 A-MSDU 中的最大 MSDU 数。来自 Extended capabilities 元素。0 表示无限制。

max\_amsdu\_len

指示 A-MSDU 的最大长度（以字节为单位）。对于带有 VHT 前缀的数据包，此字段始终有效。对于带有 HT 前缀的数据包，还适用其他限制：

1. 如果 skb 作为 BA 协议的一部分传输，则 A-MSDU 最大大小为 min(max\_amsdu\_len, 4065) 字节。
2. 如果 skb 不是 BA 协议的一部分，则 A-MSDU 的最大大小为 min(max\_amsdu\_len, 7935) 字节。

这两个附加的 HT 限制必须由低级驱动程序强制执行。这由规范定义（IEEE 802.11-2012 第8.3.2.2 注意 2）。

support\_p2p\_ps

指示 STA 是否支持 P2P PS 机制。

max\_rc\_amsdu\_len

速率控制建议的最大 A-MSDU 大小（以字节为单位）。

txq

每个 TID 数据 TX 队列（如果驱动程序使用 TXQ 抽象）

drv\_priv

驱动程序使用的数据区域，始终对齐为 sizeof(void \*)，大小在 hw 信息中确定。

**说明**

站点表条目表示我们可能正在通信的站点。由于站点在 mac80211 中是 RCU 管理的，因此您得到的任何 ieee80211\_sta 指针必须显式或隐式地受到 rcu\_read\_lock() 的保护，或者您必须小心使用这样一个指针，在调用已将其删除的 sta\_remove 回调后不使用此类指针。

#### enum sta\_notify\_cmd

通知站点命令

**常量**

STA\_NOTIFY\_SLEEP

一个站点正在休眠

STA\_NOTIFY\_AWAKE

睡眠中的站点醒来了

**说明**

与 struct ieee80211\_ops 中的 sta\_notify() 回调一起使用，这表示关联站点是否进行了电源状态转换。

#### struct ieee80211\_sta \*ieee80211\_find\_sta(struct ieee80211\_vif \*vif, const u8 \*addr)

查找站点

**参数**

struct ieee80211\_vif \*vif

虚拟接口寻找站点

const u8 \*addr

站点地址

**返回**

如果找到，则返回站点。否则返回NULL。

**注意**

必须在RCU锁下调用此函数，并且在RCU锁下生成的指针也仅有效。

#### struct [ieee80211\_sta](https://www.kernel.org/doc/html/latest/driver-api/80211/mac80211-advanced.html" \l "c.ieee80211_sta" \o "ieee80211_sta) \*ieee80211\_find\_sta\_by\_ifaddr(struct [ieee80211\_hw](https://www.kernel.org/doc/html/latest/driver-api/80211/mac80211.html" \l "c.ieee80211_hw" \o "ieee80211_hw) \*hw, const u8 \*addr, const u8 \*localaddr)

在硬件上查找站点

**参数**

struct ieee80211\_hw \*hw

从ieee80211\_alloc\_hw()获得的指针

const u8 \*addr

远程站点地址

const u8 \*localaddr

本地地址（vif->sdata->vif.addr）。使用“任何”为NULL。

**返回**

如果找到，则返回站点。否则返回NULL。

**注意**

必须在RCU锁下调用此函数，并且在RCU锁下生成的指针也仅有效。

**注意**：

您可能会将localaddr传递为NULL，但那么您只会得到与远程地址“addr”匹配的第一个STA。我们可以有多个STA与多个逻辑站点相关联（例如，考虑一个连接到同一AP硬件上的另一个BSSID的站点，而不必首先断开连接）。在这种情况下，使用localaddr NULL的此方法的结果不可靠。如有可能，请勿使用此函数的localaddr NULL。

### 硬件扫描卸载

待定

#### void ieee80211\_scan\_completed(struct ieee80211\_hw \*hw, struct cfg80211\_scan\_info \*info)

完成硬件扫描

**参数**

struct ieee80211\_hw \*hw

完成扫描的硬件

struct cfg80211\_scan\_info \*info

有关已完成扫描的信息

**说明**

在使用硬件扫描抵消（即分配hw\_scan（）回调）时，驱动程序需要调用此函数来通知mac80211扫描已完成。该函数可以从任何上下文中调用，包括hardirq上下文。

### 聚合

#### TX A-MPDU聚合

TX侧的聚合需要设置硬件标志IEEE80211\_HW\_AMPDU\_AGGREGATION。然后，将带有A-MPDU聚合标志的数据包传递给驱动程序。驱动程序或设备负责实际聚合帧以及决定聚合多少帧以及聚合哪些帧。

当某个子系统启动TX聚合时（通常适合于速率控制算法），通过调用ieee80211\_start\_tx\_ba\_session（）函数，驱动程序将通过其ampdu\_action函数得到通知，其中带有IEEE80211\_AMPDU\_TX\_START操作。

作为响应，驱动程序稍后需要调用ieee80211\_start\_tx\_ba\_cb\_irqsafe（）函数，此函数表示在peer回应后真正开始聚合会话。如果peer回复为负，会话将立即停止。请注意，即使驱动程序还没有表明已完成设置，聚合会话也可以在之前停止，在这种情况下，它不得表明设置完成。

还要注意的是，由于我们还需要等待peer的回复，因此驱动程序通过ampdu\_action回调通知握手完成。

类似地，当peer或某个调用ieee80211\_stop\_tx\_ba\_session（）的东西停止聚合会话时，将使用驱动程序的ampdu\_action函数将操作IEEE80211\_AMPDU\_TX\_STOP调用。在这种情况下，调用不得失败，并且驱动程序稍后必须调用ieee80211\_stop\_tx\_ba\_cb\_irqsafe（）函数。请注意，STA可能会在BA撕裂完成之前被销毁。

#### RX A-MPDU聚合

RX侧的聚合仅需要实现会话的ampdu\_action回调，该回调被调用以启动/停止任何块确认会话以进行RX聚合。

当由peer开始RX聚合时，驱动程序通过ampdu\_action函数得到通知，其中包含IEEE80211\_AMPDU\_RX\_START操作，并可以拒绝请求，在这种情况下，将向peer发送负响应，如果它接受它，则将发送正响应。

在会话处于活动状态时，设备/驱动程序需要将帧解聚并从一个传递到mac80211，后者将处理重新排序缓冲器。

当由peer或我们自己停止聚合会话时，将使用驱动程序的ampdu\_action函数将操作IEEE80211\_AMPDU\_RX\_STOP调用。在这种情况下，调用不得失败。

**enum ieee80211\_ampdu\_mlme\_action**

A-MPDU操作

1）常量

IEEE80211\_AMPDU\_RX\_START

开始RX聚合

IEEE80211\_AMPDU\_RX\_STOP

停止RX聚合

IEEE80211\_AMPDU\_TX\_START

开始TX聚合，驱动程序必须调用ieee80211\_start\_tx\_ba\_cb\_irqsafe（）或使用状态IEEE80211\_AMPDU\_TX\_START\_DELAY\_ADDBA调用ieee80211\_start\_tx\_ba\_cb\_irqsafe（）以延迟addba，或者只需返回特殊状态IEEE80211\_AMPDU\_TX\_START\_IMMEDIATE。

IEEE80211\_AMPDU\_TX\_STOP\_CONT

停止TX聚合，但继续传输排队的未聚合的数据包。传输所有数据包后，驱动程序必须调用ieee80211\_stop\_tx\_ba\_cb\_irqsafe（）。

IEEE80211\_AMPDU\_TX\_STOP\_FLUSH

停止TX聚合并刷新所有数据包，在删除站点时调用。在这种情况下，没有必要或原因调用ieee80211\_stop\_tx\_ba\_cb\_irqsafe（），因为mac80211假定会话已经消失并删除了站点。

IEEE80211\_AMPDU\_TX\_STOP\_FLUSH\_CONT

当停止TX聚合但驱动程序尚未调用ieee80211\_stop\_tx\_ba\_cb\_irqsafe（）并且现在连接已断开且将删除该站点时调用。驱动程序应在调用此函数时清除和丢弃剩余数据包。

IEEE80211\_AMPDU\_TX\_OPERATIONAL

TX聚合已开始运行

2）说明

这些标志与struct ieee80211\_ops中的ampdu\_action（）回调一起使用，以指示需要执行的操作。

请注意，即使在调用ieee80211\_start\_tx\_ba\_cb\_irqsafe来开始TX聚合会话之前，驱动程序必须能够处理停止TX聚合会话的情况，因为对等方可能会收到addBA帧并立即发送delBA！

### 空间复用省电（SMPS）

SMPS（空间复用省电）是一种在802.11n实现中保护电源的机制。有关机制和原理的详细信息，请参见802.11（经802.11n-2009修改）“11.2.3 SM power save”。

mac80211实现能够将操作帧发送到AP以更新站点的SMPS模式，并指示驱动程序进入特定模式。它还在关联握手期间宣布所请求的SMPS模式。这个特征需要硬件支持，可以通过硬件标志表示。

默认模式将是“自动”，它由nl80211 / cfg80211定义为（常规）电源管理中的动态SMPS，否则关闭SMPS。

为了支持这个特性，驱动程序必须设置适当的硬件支持标志，并处理config（）操作中的SMPS标志。然后通过这个机制被指示在连接到HT AP时进入请求的SMPS模式。

#### void ieee80211\_request\_smps（struct ieee80211\_vif \* vif，unsigned int link\_id，enum ieee80211\_smps\_mode smps\_mode）

请求SM PS转换

**参数**

struct ieee80211\_vif \* vif

从add\_interface回调的struct ieee80211\_vif指针。

unsigned int link\_id

MLO的链接ID，或0

enum ieee80211\_smps\_mode smps\_mode

新的SM PS模式

**说明**

这使驱动程序可以在托管模式下请求SM PS转换。当驱动程序具有有关可能的干扰的更多信息时，例如通过蓝牙，这将非常有用。

待定

本书的此部分说明了速率控制算法接口及其与mac80211和驱动程序的关系。

#### enum ieee80211\_smps\_mode

空间复用省电模式

**常量**

IEEE80211\_SMPS\_AUTOMATIC

自动

IEEE80211\_SMPS\_OFF

关闭

IEEE80211\_SMPS\_STATIC

静态

IEEE80211\_SMPS\_DYNAMIC

动态

IEEE80211\_SMPS\_NUM\_MODES

内部，不使用

### 速率控制API

待定

#### int ieee80211\_start\_tx\_ba\_session（struct ieee80211\_sta \* sta，u16 tid，u16 timeout）

启动tx块确认会话。

**参数**

struct ieee80211\_sta \* sta

要启动BA会话的站点

u16 tid

TID上的BA。

u16超时

会话超时值（以TU表示）

**返回**

如果发送addBA请求，则成功，否则失败

**说明**

尽管mac80211 /低级驱动程序/用户空间应用程序可以估计在某个RA / TID上启动聚合的需要，但会话级别将由mac80211管理。

#### void ieee80211\_start\_tx\_ba\_cb\_irqsafe（struct ieee80211\_vif \* vif，const u8 \* ra，u16 tid）

低级驱动程序准备好聚合。

**参数**

struct ieee80211\_vif \* vif

来自add\_interface回调的struct ieee80211\_vif指针

const u8 \* ra

BA会话收件人的接收器地址。

u16 tid

TID上的BA。

**说明**

低级驱动程序必须在准备好BA会话后调用此函数。它可以从任何上下文调用。

#### int ieee80211\_stop\_tx\_ba\_session（struct ieee80211\_sta \* sta，u16 tid）

停止块确认会话。

**参数**

struct ieee80211\_sta \* sta

停止BA会话的站点

u16 tid

停止BA的TID。

**返回**

当TID无效或未激活聚合时，负错误

**说明**

尽管mac80211 /低级驱动程序/用户空间应用程序可以估计在某个RA / TID上停止聚合的需要，但会话级别将由mac80211管理。

#### enum ieee80211\_rate\_control\_changed

表示发生了什么变化的标志

**常数**

IEEE80211\_RC\_BW\_CHANGED

可以用于向此站点发送传输的带宽发生了变化。实际带宽在站点信息中 - 对于HT20 / 40，在IEEE80211\_HT\_CAP\_SUP\_WIDTH\_20\_40标志发生变化，对于HT和VHT，带宽字段发生变化。

IEEE80211\_RC\_SMPS\_CHANGED

该站点的SMPS状态已更改。

IEEE80211\_RC\_SUPP\_RATES\_CHANGED

由于发现有关对等方的更多信息，该对等方的支持速率集合（在IBSS模式下）已更改。

IEEE80211\_RC\_NSS\_CHANGED

对等方更改了N\_SS（空间流的数量）

#### struct ieee80211\_tx\_rate\_control

用于/从RC算法控制率的速率控制信息

**定义**

struct ieee80211\_tx\_rate\_control {

struct ieee80211\_hw \*hw;

struct ieee80211\_supported\_band \*sband;

struct ieee80211\_bss\_conf \*bss\_conf;

struct sk\_buff \*skb;

struct ieee80211\_tx\_rate reported\_rate;

bool rts, short\_preamble;

u32 rate\_idx\_mask;

u8 \*rate\_idx\_mcs\_mask;

bool bss;

};

**成员**

hw

调用算法的硬件。

sband

在其上传输此帧的频段。

bss\_conf

当前的BSS配置

skb

将要传输的skb，其控制信息需要填写

reported\_rate

速率控制算法可以填充这个属性，以指示应向用户空间报告哪个速率并用于网格网络中的速率计算。

rts

是否使用RTS发送此帧，因为它的长度超过了RTS阈值

短前言

如果所选速率支持，则mac80211是否会请求短前言传输

rate\_idx\_mask

用户请求（传统）速率屏蔽

rate\_idx\_mcs\_mask

用户请求的MCS速率屏蔽（如果未使用，则为NULL）

bss

此帧是在AP模式还是IBSS模式下发送的

#### bool rate\_control\_send\_low(struct [ieee80211\_sta](https://www.kernel.org/doc/html/v4.19/driver-api/80211/mac80211-advanced.html" \l "c.ieee80211_sta" \o "ieee80211_sta) \* sta, void \* priv\_sta, struct [ieee80211\_tx\_rate\_control](https://www.kernel.org/doc/html/v4.19/driver-api/80211/mac80211-advanced.html" \l "c.ieee80211_tx_rate_control" \o "ieee80211_tx_rate_control) \* txrc)

用于管理/无ACK帧的驱动程序的帮助程序

**参数**

struct ieee80211\_sta \* sta

struct ieee80211\_sta指向目标目标的指针。请注意，这可以为空。

void \* priv\_sta

私有速率控制结构。这可能为null。

struct ieee80211\_tx\_rate\_control \* txrc

我们应该为mac80211填充的速率控制信息。

**说明**

同意使用最低速率发送管理帧和NO\_ACK数据，并且具有相应hw重试次数的速率控制算法应在其mac80211 get\_rate回调的开头使用此选项。如果返回true，则速率控制可以简单地返回。如果返回false，则我们保证sta和sta和priv\_sta不为null。

希望对多播/广播帧进行更智能的速率选择的速率控制算法可以选择不使用此选项。

待定

本书的这一部分说明了mac80211的内部。

### 密钥处理

#### 密钥处理基础知识

在mac80211中，基于每个接口（sub\_if\_data）密钥和每个站点密钥进行处理。由于每个站点都属于一个接口，因此每个站点密钥也属于该接口。

对于在软件中实现的算法，硬件加速是在尽力的基础上完成的，对于每个键，硬件被要求为其卸载启用该键，但如果它无法执行该操作，则该键仅用于软件加密（除非它是为未在软件中实现的算法）。目前还没有办法知道密钥是在SW还是HW中处理的，除非在debugfs中查看。

所有密钥管理都受互斥体的内部保护。在mac80211的所有其他部分中，密钥引用与STA结构引用一样，受RCU保护。但是请注意，某些东西没有受到保护，即硬件加速功能中的key->sta解引用。这意味着sta\_info\_destroy（）必须删除等待RCU宽限期的密钥。

#### 更多··待定

待定

### 接收处理

待定

### 传输处理

待定

### 站点信息处理

#### 编程信息

**struct sta\_info**

STA 信息

1）定义

struct sta\_info {

struct list\_head list, free\_list;

struct rcu\_head rcu\_head;

struct rhlist\_head hash\_node;

u8 addr[ETH\_ALEN];

struct ieee80211\_local \*local;

struct ieee80211\_sub\_if\_data \*sdata;

struct ieee80211\_key \_\_rcu \*gtk[NUM\_DEFAULT\_KEYS + NUM\_DEFAULT\_MGMT\_KEYS];

struct ieee80211\_key \_\_rcu \*ptk[NUM\_DEFAULT\_KEYS];

u8 ptk\_idx;

struct rate\_control\_ref \*rate\_ctrl;

void \*rate\_ctrl\_priv;

spinlock\_t rate\_ctrl\_lock;

spinlock\_t lock;

struct ieee80211\_fast\_tx \_\_rcu \*fast\_tx;

struct ieee80211\_fast\_rx \_\_rcu \*fast\_rx;

struct ieee80211\_sta\_rx\_stats \_\_percpu \*pcpu\_rx\_stats;

#ifdef CONFIG\_MAC80211\_MESH;

struct mesh\_sta \*mesh;

#endif;

struct work\_struct drv\_deliver\_wk;

u16 listen\_interval;

bool dead;

bool removed;

bool uploaded;

enum ieee80211\_sta\_state sta\_state;

unsigned long \_flags;

spinlock\_t ps\_lock;

struct sk\_buff\_head ps\_tx\_buf[IEEE80211\_NUM\_ACS];

struct sk\_buff\_head tx\_filtered[IEEE80211\_NUM\_ACS];

unsigned long driver\_buffered\_tids;

unsigned long txq\_buffered\_tids;

long last\_connected;

struct ieee80211\_sta\_rx\_stats rx\_stats;

struct {

struct ewma\_signal signal;

struct ewma\_signal chain\_signal[IEEE80211\_MAX\_CHAINS];

} rx\_stats\_avg;

\_\_le16 last\_seq\_ctrl[IEEE80211\_NUM\_TIDS + 1];

struct {

unsigned long filtered;

unsigned long retry\_failed, retry\_count;

unsigned int lost\_packets;

unsigned long last\_tdls\_pkt\_time;

u64 msdu\_retries[IEEE80211\_NUM\_TIDS + 1];

u64 msdu\_failed[IEEE80211\_NUM\_TIDS + 1];

unsigned long last\_ack;

s8 last\_ack\_signal;

bool ack\_signal\_filled;

struct ewma\_avg\_signal avg\_ack\_signal;

} status\_stats;

struct {

u64 packets[IEEE80211\_NUM\_ACS];

u64 bytes[IEEE80211\_NUM\_ACS];

struct ieee80211\_tx\_rate last\_rate;

u64 msdu[IEEE80211\_NUM\_TIDS + 1];

} tx\_stats;

u16 tid\_seq[IEEE80211\_QOS\_CTL\_TID\_MASK + 1];

struct sta\_ampdu\_mlme ampdu\_mlme;

#ifdef CONFIG\_MAC80211\_DEBUGFS;

struct dentry \*debugfs\_dir;

#endif;

enum ieee80211\_sta\_rx\_bandwidth cur\_max\_bandwidth;

enum ieee80211\_smps\_mode known\_smps\_mode;

const struct ieee80211\_cipher\_scheme \*cipher\_scheme;

struct codel\_params cparams;

u8 reserved\_tid;

struct cfg80211\_chan\_def tdls\_chandef;

struct ieee80211\_sta sta;

};

2）成员

list

全局链表条目

free\_list

用于跟踪免费电台的列表条目

rcu\_head

用于释放此站结构的 RCU 头

hash\_node

rhashtable 的哈希节点

addr

站的 MAC 地址 - 从公共部分复制，让哈希表只使用一个缓存行

local

指向全局信息的指针

sdata

该站所属的虚拟接口

gtk

与此站协商的组密钥（如果有）

ptk

与此站协商的对等密钥（如果有）

ptk\_idx

最后安装的对等密钥索引

rate\_ctrl

码率控制算法参考

rate\_ctrl\_priv

速率控制专用每 STA 指针

rate\_ctrl\_lock

自旋锁用于保护速率控制数据（算法内部的数据，因此在那里序列化调用）

lock

用于锁定所有需要锁定的字段，参见头文件中的注释。

fast\_tx

TX 快速路径信息

fast\_rx

RX 快速路径信息

pcpu\_rx\_stats

每个 CPU 的 RX 统计信息，仅在驱动程序需要时分配（通过公布 USES\_RSS hw 标志）

mesh

网状STA信息

drv\_deliver\_wk

用于驱动PS解锁后的帧传递

listen\_interval

当我们作为 AP 时，本站的收听间隔

dead

当 sta 未链接时设置为 true

removed

当 sta 从 sta\_list 中删除时设置为 true

uploaded

将 sta 上传到驱动程序时设置为 true

sta\_state

复制有关站状态的信息（用于调试）

\_flags

STA flags，见，不要直接使用[enum ieee80211\_sta\_info\_flags](https://www.kernel.org/doc/html/v4.19/driver-api/80211/mac80211-advanced.html" \l "c.ieee80211_sta_info_flags" \o "ieee80211_sta_info_flags)

ps\_lock

用于省电（当 mac80211 是 AP 时）相关锁定

ps\_tx\_buf

当它离开省电状态或轮询时要传输到该站的帧缓冲区（每个 AC）

tx\_filtered

我们已经尝试传输但由于 STA 进入省电状态而被硬件过滤的帧的缓冲区（每个 AC），当它离开 powersave 或轮询帧时，这些也会被传送到站

driver\_buffered\_tids

驱动程序在其上缓冲数据的 TID 的位图

txq\_buffered\_tids

mac80211 缓冲 txq 数据的 TID 位图

last\_connected

站点连接的时间（以秒为单位）

rx\_stats

接收统计

last\_seq\_ctrl

最后从该 STA 接收到的 seq/frag 编号（每个 TID 加一个用于非 QoS 帧）

status\_stats

TX状态统计

tx\_stats

交易统计

tid\_seq

用于发送到此 STA 的每个 TID 序列号

ampdu\_mlme

A-MPDU状态机状态

debugfs\_dir

调试文件系统目录 dentry

cur\_max\_bandwidth

用于 TX 到站的最大带宽，取自 HT/VHT 功能或 VHT 操作模式通知

known\_smps\_mode

客户端认为我们所处的 smps\_mode。仅与 AP 相关。

cipher\_scheme

此站的可选密码方案

cparams

该站的 CoDel 参数。

reserved\_tid

保留 TID（如果有的话，否则 IEEE80211\_TID\_UNRESERVED）

tdls\_chandef

TDLS 对等体可以具有与 BSS 兼容的更宽的 chandef。

sta

我们与司机分享的车站信息

3）说明

此结构收集有关 mac80211 正在与之通信的站点的信息

**enum ieee80211\_sta\_info\_flags**

站标志

1. 常量

WLAN\_STA\_AUTH

站已通过身份验证。

WLAN\_STA\_ASSOC

站关联。

WLAN\_STA\_PS\_STA

工作站处于省电模式

WLAN\_STA\_AUTHORIZED

站被授权发送/接收流量。始终检查此位，因此在未使用虚拟端口控制时需要为所有站启用。

WLAN\_STA\_SHORT\_PREAMBLE

站点能够接收短前导帧。

WLAN\_STA\_WDS

Station 是我们的 WDS 同行之一。

WLAN\_STA\_CLEAR\_PS\_FILT

当下一帧传输到该站时，在硬件中清除 PS 过滤器（使用 IEEE80211\_TX\_CTL\_CLEAR\_PS\_FILT 控制标志）。

WLAN\_STA\_MFP

管理帧保护与此 STA 一起使用。

WLAN\_STA\_BLOCK\_BA

用于在挂起/恢复和站点移除期间拒绝 ADDBA 请求（TX 和 RX）。

WLAN\_STA\_PS\_DRIVER

驱动程序需要在逻辑上将此站点保持在省电模式，以刷新可能仍在队列中的帧

WLAN\_STA\_PSPOLL

当司机保持 Station 处于省电模式时，Station 发送 PS-poll，当司机解锁时回复。

WLAN\_STA\_TDLS\_PEER

站是 TDLS 对等体。

WLAN\_STA\_TDLS\_PEER\_AUTH

此 TDLS 对等体被授权发送直接数据包。这意味着链接已启用。

WLAN\_STA\_TDLS\_INITIATOR

我们是与这个站的 TDLS 链接的发起者。

WLAN\_STA\_TDLS\_CHAN\_SWITCH

此 TDLS 对等点支持 TDLS 信道切换

WLAN\_STA\_TDLS\_OFF\_CHANNEL

本地 STA 当前与此 TDLS 对等体处于信道外

WLAN\_STA\_TDLS\_WIDER\_BW

此 TDLS 对等点支持在 BSS 基本信道上使用更宽的带宽。

WLAN\_STA\_UAPSD

当司机保持车站处于省电模式时，车站请求计划外的 SP，当司机解锁车站时回复。

WLAN\_STA\_SP

Station 处于服务期，因此不要尝试回复其他 uAPSD 触发帧或 PS-Poll。

WLAN\_STA\_4ADDR\_EVENT

已为此帧发送 4-addr 事件。

WLAN\_STA\_INSERTED

这个站被插入到哈希表中。

WLAN\_STA\_RATE\_CONTROL

已为此站初始化速率控制。

WLAN\_STA\_TOFFSET\_KNOWN

为该站计算的 toffset 有效。

WLAN\_STA\_MPSP\_OWNER

本地 STA 是网格对等服务周期的所有者。

WLAN\_STA\_MPSP\_RECIPIENT

本地 STA 是 MPSP 的接收者。

WLAN\_STA\_PS\_DELIVER

站醒了，但我们仍在阻塞 TX，直到传递未决帧

NUM\_WLAN\_STA\_FLAGS

定义标志的数量

1. 说明

这些标志与的标志成员一起使用，但只是间接地与朋友一起使用。[struct sta\_info](https://www.kernel.org/doc/html/v4.19/driver-api/80211/mac80211-advanced.html" \l "c.sta_info" \o "sta_info)set\_sta\_flag()

#### STA信息生命周期规则

STA信息结构（struct sta\_info）在哈希表中进行更快的查找并在列表中进行迭代。它们使用RCU进行管理，即访问列表和哈希表受RCU保护。

使用sta\_info\_alloc()分配STA信息结构后，调用者拥有该结构。然后它必须使用sta\_info\_insert()或sta\_info\_insert\_rcu()将其插入哈希表中; 仅在后者（获得rcu读取节的rcu\_read\_locked（）除外）的情况下，指针仍然有效。请注意，在插入之前，调用者可能无法对STA信息做太多事情，特别是不能开始任何网格对等链路管理或添加加密密钥。

当插入失败（sta\_info\_insert（）返回非零）时，该结构将已由sta\_info\_insert（）释放！

当建立与对等体的连接时，mac80211会添加站点条目。这对于我们支持的不同类型的接口意味着不同的事情。对于常规站点，这意味着当我们从AP收到关联响应时，我们添加AP站点。对于IBSS模式，当我们了解到同一IBSS中的对等体时，会发生这种情况。对于WDS模式，我们立即在设备打开时添加对等体的sta。使用AP模式时，我们通过nl80211从用户空间请求每个相应站点的站点。

为了删除STA info结构，可以使用各种sta\_info\_destroy\_（\*）调用。

STA条目没有所有权的概念，每个结构都由全局哈希表/列表拥有，直到被删除。所有使用该结构的用户都需要受到RCU保护，以便在使用完该结构之前它不会被释放。

### 聚合

#### struct sta\_ampdu\_mlme

STA聚合信息

**定义**

struct sta\_ampdu\_mlme {

struct mutex mtx;

struct tid\_ampdu\_rx \_\_rcu \*tid\_rx[IEEE80211\_NUM\_TIDS];

u8 tid\_rx\_token[IEEE80211\_NUM\_TIDS];

unsigned long tid\_rx\_timer\_expired[BITS\_TO\_LONGS(IEEE80211\_NUM\_TIDS)];

unsigned long tid\_rx\_stop\_requested[BITS\_TO\_LONGS(IEEE80211\_NUM\_TIDS)];

unsigned long tid\_rx\_manage\_offl[BITS\_TO\_LONGS(2 \* IEEE80211\_NUM\_TIDS)];

unsigned long agg\_session\_valid[BITS\_TO\_LONGS(IEEE80211\_NUM\_TIDS)];

unsigned long unexpected\_agg[BITS\_TO\_LONGS(IEEE80211\_NUM\_TIDS)];

struct work\_struct work;

struct tid\_ampdu\_tx \_\_rcu \*tid\_tx[IEEE80211\_NUM\_TIDS];

struct tid\_ampdu\_tx \*tid\_start\_tx[IEEE80211\_NUM\_TIDS];

unsigned long last\_addba\_req\_time[IEEE80211\_NUM\_TIDS];

u8 addba\_req\_num[IEEE80211\_NUM\_TIDS];

u8 dialog\_token\_allocator;

};

**成员**

mtx

保护所有 TX 数据的互斥锁（除了对 tid\_tx[idx] 的非 NULL 赋值，它受 sta 自旋锁保护） tid\_start\_tx 也受 sta->lock 保护。

tid\_rx

每个 TID 的 Rx 聚合信息 – RCU 保护

tid\_rx\_token

有效聚合会话的对话令牌

tid\_rx\_timer\_expired

指示 RX 计时器在其工作运行之前在哪些 TID 上过期的位图

tid\_rx\_stop\_requested

位图指示驱动程序请求关闭每个 TID 的 BA 会话，直到它的工作运行

tid\_rx\_manage\_offl

指示哪些 BA 会话由于卸载而被请求视为已启动/已停止的位图

agg\_session\_valid

指示哪个 TID 具有打开的 rx BA 会话的位图

unexpected\_agg

位图指示哪个 TID 由于会话外的意外聚合相关帧而已发送 delBA

work

用于开始/停止聚合的工作结构

tid\_tx

每个 TID 的 Tx 聚合信息

tid\_start\_tx

请求开始的会话

last\_addba\_req\_time

最后一个 addBA 请求的时间戳。

addba\_req\_num

发送 addBA 请求的次数。

dialog\_token\_allocator

每个新会话的对话令牌枚举器；

#### struct tid\_ampdu\_tx

TID聚合信息(Tx).

**定义**

struct tid\_ampdu\_tx {

struct rcu\_head rcu\_head;

struct timer\_list session\_timer;

struct timer\_list addba\_resp\_timer;

struct sk\_buff\_head pending;

struct sta\_info \*sta;

unsigned long state;

unsigned long last\_tx;

u16 timeout;

u8 dialog\_token;

u8 stop\_initiator;

bool tx\_stop;

u16 buf\_size;

u16 ssn;

u16 failed\_bar\_ssn;

bool bar\_pending;

bool amsdu;

u8 tid;

};

**成员**

rcu\_head

用于释放结构的 rcu 头

session\_timer

检查我们是否在 TID 上保持 Tx-ing（通过超时值）

addba\_resp\_timer

对等方对 addba 请求的响应计时器

pending

pending frames queue——使用sta的自旋锁来保护

sta

我们依附的车站

state

会话状态（见上文）

last\_tx

最后 tx 活动的 jiffies

timeout

ADDBA 请求中要填写的会话超时值

dialog\_token

聚合会话的对话令牌

stop\_initiator

会话停止的发起者

tx\_stop

停止时发送 DelBA 帧

buf\_size

在接收方重新排序缓冲区大小

failed\_bar\_ssn

上次失败的 BAR tx 尝试的 ssn

bar\_pending

BAR需要重新发送

amsdu

支持带有 A-MDPU 的 A-MSDU

tid

工号

**说明**

该结构的生命周期由RCU管理，持有它的数组分配必须保持聚合互斥。

并且仅当状态具有标志HT\_AGG\_STATE\_OPERATIONAL设置时，TX路径可以在RCU无锁情况下访问它。否则，TX路径还必须获取自旋锁并重新检查状态，请参见触及tx代码中的注释。

#### struct tid\_ampdu\_rx

TID 聚合信息(Rx).

**定义**

struct tid\_ampdu\_rx {

struct rcu\_head rcu\_head;

spinlock\_t reorder\_lock;

u64 reorder\_buf\_filtered;

struct sk\_buff\_head \*reorder\_buf;

unsigned long \*reorder\_time;

struct sta\_info \*sta;

struct timer\_list session\_timer;

struct timer\_list reorder\_timer;

unsigned long last\_rx;

u16 head\_seq\_num;

u16 stored\_mpdu\_num;

u16 ssn;

u16 buf\_size;

u16 timeout;

u8 tid;

u8 auto\_seq:1,removed:1, started:1;

};

**成员**

rcu\_head

用于释放此结构的 RCU 头

reorder\_lock

序列化对重新排序缓冲区的访问，请参见下文。

reorder\_buf\_filtered

位图指示重新排序缓冲区中过滤帧的位置，释放帧时应忽略这些帧

reorder\_buf

缓冲区以重新排序传入的聚合 MPDU。MPDU可以是具有单独报告的子帧的A-MSDU。

reorder\_time

添加 skb 时的 jiffies

sta

我们依附的车站

session\_timer

检查对等方是否在 TID 上保持 Tx-ing（通过超时值）

reorder\_timer

从重新排序缓冲区中释放过期的帧。

last\_rx

上次 RX 活动的 jiffies

head\_seq\_num

重新排序缓冲区中的头部序列号。

stored\_mpdu\_num

重新排序缓冲区中的 MPDU 数量

ssn

预期聚合的起始序列号。

buf\_size

传入 A-MPDU 的缓冲区大小

timeout

重置计时器值（以 TU 为单位）。

tid

工号

auto\_seq

用于卸载 BA 会话以自动选择 head\_seq\_and 和 ssn。

removed

此会话已删除（但可能由于 RCU 而被发现）

started

此会话已开始（已收到主管 ssn 或更高级别）

**说明**

此结构的生命周期由 RCU 管理，对包含它的数组的分配必须包含聚合互斥锁。

reorder\_lock用于保护此结构的成员，timeout、buf\_size和dialog\_token除外，它们在结构的整个生命周期内保持不变（对话令牌仅用于调试）。

### 同步化

待定

锁定，大量 RCU

# 用户空间 I/O HOWTO

## 关于本文档

### 翻译

如果您知道此文档的任何翻译，或者您有兴趣翻译它，请发送电子邮件到hjk@hansjkoch.de。

### 前言

对于许多类型的设备，创建Linux内核驱动程序是过度的。实际上，需要的只是某种处理中断并提供对设备内存空间的访问的方式。控制设备的逻辑不一定非得在内核中实现，因为设备不需要利用内核提供的任何其他资源。这种常见的设备类别之一是工业I/O卡。

为了解决这种情况，设计了用户空间I/O系统（UIO）。对于典型的工业I/O卡，只需要非常小的内核模块。驱动程序的主要部分将在用户空间中运行。这简化了开发并降低了内核模块内的严重错误风险。

请注意，UIO不是通用的驱动程序接口。那些已经由其他内核子系统（如网络、串口或USB）很好地处理的设备不适合使用UIO驱动程序。最适合UIO驱动程序的硬件应满足以下所有条件：

设备具有可映射的内存。可以通过写入此内存来完全控制设备。

设备通常会生成中断。

该设备不适合于标准内核子系统之一。

### 致谢

我想感谢Linutronix的Thomas Gleixner和Benedikt Spranger，他们不仅编写了大部分UIO代码，还通过提供各种背景信息极大地帮助了我编写这份HOWTO。

### 反馈

关于本文档有错误？（或者可能是正确的东西？）我很乐意听取您的意见。请给我发送电子邮件hjk@hansjkoch.de。

## 关于UIO

如果您使用UIO为您的卡驱动程序，您将获得以下内容：

1. 只需编写和维护一个小内核模块。
2. 在用户空间中开发驱动程序的主要部分，具有您习惯的所有工具和库。
3. 您的驱动程序中的错误不会导致内核崩溃。
4. 可以在不重新编译内核的情况下更新驱动程序。

### UIO的工作原理

每个UIO设备通过设备文件和多个sysfs属性文件进行访问。第一个设备的设备文件将被称为/dev/uio0，后续设备的设备文件为/dev/uio1、/dev/uio2等。

/dev/uioX用于访问卡的地址空间。只需使用mmap()即可访问您的卡的寄存器或RAM位置。

中断通过从/dev/uioX读取来处理。从/dev/uioX进行阻塞读取()将在中断发生时立即返回。您还可以在/dev/uioX上使用select()等待中断。从/dev/uioX读取的整数值表示总中断计数。您可以使用此数字来确定是否错过了某些中断。

对于一些内部具有多个中断源但没有单独的IRQ屏蔽和状态寄存器的硬件，如果内核处理程序通过写入芯片的IRQ寄存器来禁用它们，那么用户空间无法确定中断源。这种情况下，内核必须完全禁用IRQ以保持芯片的寄存器不变。现在，用户空间部分可以确定中断的原因，但无法重新启用中断。另一个角落是芯片，其中重新启用中断是对组合的IRQ状态/确认寄存器进行读取-修改-写操作。如果同时发生新的中断，这将导致竞态条件。

为了解决这些问题，UIO还实现了一个写()函数。它通常不会被使用，对于只具有单个中断源或具有单独的IRQ屏蔽和状态寄存器的硬件，可以忽略它。但是，如果需要，对/dev/uioX的写操作将调用驱动程序实现的irqcontrol()函数。您必须写入通常为0或1的32位值，以禁用或启用中断。如果驱动程序未实现irqcontrol()，则write()将返回-ENOSYS。

为了正确处理中断，您的自定义内核模块可以提供自己的中断处理程序，它将自动由内置处理程序调用。

对于不生成中断但需要轮询的卡，可以设置定时器以可配置的时间间隔触发中断处理程序。通过从定时器事件处理程序调用uio\_event\_notify()来完成此中断模拟。

每个驱动程序提供用于读取或写入变量的属性。这些属性可通过sysfs文件访问。自定义内核驱动程序模块可以向uio驱动程序拥有的设备添加自己的属性，但目前尚未添加到UIO设备本身。如果发现这很有用，则将来可能会改变这种情况。

以下是UIO框架提供的标准属性：

1. 名称：您的设备的名称。建议使用内核模块的名称。
2. 版本：由驱动程序定义的版本字符串。这允许用户空间部分处理不同版本的内核模块。
3. 事件：驱动程序自上次读取设备节点以来处理的总中断次数。

这些属性显示在/sys/class/uio/uioX目录下。请注意，该目录可能是符号链接而不是真实目录。访问它的任何用户空间代码必须能够处理这一点。

每个UIO设备可以为内存映射提供一个或多个内存区域。这是必要的，因为某些工业输入/输出卡需要在驱动程序中访问多个PCI内存区域。

每个映射在sysfs中都有自己的目录，第一个映射显示为/sys/class/uio/uioX/maps/map0/。后续映射创建目录map1/，map2/等等。仅当映射的大小不为0时，这些目录才会出现。

每个mapX/目录包含四个只读文件，用于显示内存的属性：

1. 名称：此映射的字符串标识符。这是可选的，字符串可以为空。驱动程序可以设置此项以使用户空间更容易找到正确的映射。
2. 地址：可以映射的内存地址。
3. 大小：addr指向的内存的大小（以字节为单位）。
4. 偏移量：必须添加到通过mmap（）返回的指针中以到达实际设备内存的偏移量（以字节为单位）。如果设备的内存不是页面对齐的，则这一点很重要。请记住，由mmap（）返回的指针始终是页面对齐的，因此始终添加此偏移量是良好的风格。

从用户空间，不同的映射通过调整mmap（）调用的偏移参数来区分。要映射映射N的内存，您必须使用N个页面大小作为偏移量：

offset= N \* getpagesize();

有时，存在类似内存的区域的硬件无法使用此处说明的技术进行映射，但仍然可以从用户空间访问它们。最常见的例子是x86 ioport。在x86系统上，用户空间可以使用ioperm（），iopl（），inb（），outb（）等函数访问这些ioport。

由于这些ioport区域无法映射，它们不会像上述普通内存那样出现在/sys/class/uio/uioX/maps/下。如果没有关于端口区域的信息，硬件部分的驱动程序将很难找出哪些端口属于哪个UIO设备。

为了解决这种情况，添加了新目录/sys/class/uio/uioX/portio/。它只存在于驱动程序想要将有关一个或多个端口区域的信息传递给用户空间的情况下。如果是这种情况，则名为port0，port1等的子目录将出现在/sys/class/uio/uioX/portio/下面。

每个portX/目录包含四个只读文件，用于显示端口区域的名称、起始地址、大小和类型：

1. 名称：此端口区域的字符串标识符。该字符串是可选的且可以为空。驱动程序可以设置它以使用户空间更容易找到某个端口区域。
2. 起始地址：此区域的第一个端口。
3. 大小：此区域中的端口数。
4. 端口类型：说明端口类型的字符串。

## 编写您自己的内核模块

请看uio\_cif.c作为示例。以下段落解释了此文件的不同部分。

### struct uio\_info

该结构告诉框架驱动程序的详细信息。其中一些成员是必需的，其他成员是可选的。

char \*name: 必需。驱动程序在sysfs中出现的名称。建议使用模块的名称。

const char \*version：必须。该字符串出现在/sys/class/uio/uioX/version。

struct uio\_mem mem[MAX\_UIO\_MAPS]：如果您有可以通过mmap()映射的内存，则必须填写uio\_mem结构中的一个以处理每个映射。请参阅以下说明了解详细信息。

struct uio\_port port[MAX\_UIO\_PORTS\_REGIONS]：如果要将有关ioports的信息传递给用户空间，则必填每个端口区域之一的uio\_port结构。请参阅以下说明了解详细信息。

long irq：必需。如果硬件生成中断，则在初始化期间确定irq编号是您模块的任务。如果您没有硬件生成的中断，但想以某种其他方式触发中断处理程序，则将irq设置为UIO\_IRQ\_CUSTOM。如果您根本没有中断，则可以将irq设置为UIO\_IRQ\_NONE，但这很少有意义。

unsigned long irq\_flags：如果已将irq设置为硬件中断号，则需要。这里给出的标志将在调用request\_irq()中使用。

int (\*mmap)(struct uio\_info \* info，struct vm\_area\_struct \* vma)：可选。如果您需要特殊的mmap()函数，则可以在此处设置它。如果此指针不为NULL，则您的mmap()将被调用而不是内置的mmap()。

int(\*open)(struct uio\_info \* info，struct inode \* inode)：可选。您可能希望拥有自己的打开函数，例如仅在实际使用设备时启用中断。

int(\*release)(struct uio\_info \* info，struct inode \* inode)：可选。如果定义自己的open()，则可能还需要自定义release()函数。

int(\*irqcontrol)(struct uio\_info \* info，s32 irq\_on)：可选。如果需要能够通过写入/dev/uioX从用户空间启用或禁用中断，则可以实现该函数。参数irq\_on将为0以禁用中断，为1以启用中断。

通常，您的设备将具有一个或多个可以映射到用户空间的内存区域。对于每个区域，您必须在mem[]数组中设置一个uio\_mem结构。以下是uio\_mem结构的字段说明：

const char \*name：可选。设置此项以帮助识别内存区域，它将出现在相应的sysfs节点中。

int memtype：如果使用映射则必需。如果您在卡上有物理内存可以映射，则设置为UIO\_MEM\_PHYS。对于逻辑内存（例如使用\_\_get\_free\_pages()分配但未使用kmalloc()），请使用UIO\_MEM\_LOGICAL。还有UIO\_MEM\_VIRTUAL用于虚拟内存。

phys\_addr\_t addr：如果使用映射，则必需。填写您的内存块的地址。此地址是出现在sysfs中的地址。

resource\_size\_t size：填写addr指向的内存块的大小。如果大小为零，则该映射将被视为未使用。请注意，对于所有未使用的映射，必须将size初始化为零。

void \*internal\_addr：如果您必须从内核模块内部访问此内存区域，则需要使用类似于ioremap()的方法进行内部映射。此函数返回的地址无法映射到用户空间，因此您不能将其存储在addr中。而是使用internal\_addr来记住这样的地址。

请不要触摸uio\_mem结构的map元素!它由UIO框架用于为此映射设置sysfs文件。只需将其保持原样即可。

有时，您的设备可以有一个或多个端口区域，无法映射到用户空间。但是，如果有其他用户空间访问这些端口的可能性，则将有关端口的信息提供给sysfs是有意义的。对于每个区域，您必须在port[]数组中设置一个uio\_port结构。以下是uio\_port结构的字段说明：

char \*porttype：必需。将其设置为预定义的常量之一。在x86体系结构中找到ioport时，请使用UIO\_PORT\_X86。

unsigned long start：如果使用端口区域，则必需。填写此区域的第一个端口号。

unsigned long size：填写此区域中的端口数。如果size为零，则该区域将被视为未使用。请注意，对于所有未使用的区域，必须将size初始化为零。

请不要触摸uio\_port结构的portio元素!这是UIO框架内部用于为此区域设置sysfs文件的。只需将其保持原样即可。

### 添加中断处理程序

您在中断处理程序中需要执行的操作取决于您的硬件以及您希望如何处理它。您应该尽量减少内核中断处理程序中的代码量。如果您的硬件不需要您在每次中断后执行的任何操作，则您的处理程序可以为空。

另一方面，如果每次中断后需要执行某些操作，您必须在内核模块中执行。请注意，您不能依赖于驱动程序的用户空间部分。您的用户空间程序可能随时终止，可能会使您的硬件处于仍需要适当中断处理的状态。

可能还有应用程序需要在每个中断时从您的硬件中读取数据并将其缓冲在应为此目的分配的内核内存块中。使用此技术，如果您的用户空间程序错过中断，您就可以避免数据丢失。

关于共享中断的说明：只有当驱动程序能够检测到硬件是否已触发中断时，您的驱动程序才应支持中断共享。这通常是通过查看中断状态寄存器来完成的。如果驱动程序看到IRQ位实际上被设置了，它将执行其操作，处理程序返回IRQ\_HANDLED。如果驱动程序检测到不是您的硬件引起的中断，则它将不执行任何操作并返回IRQ\_NONE，允许内核调用下一个可能的中断处理程序。

如果您决定不支持共享中断，则您的卡将无法在没有空闲中断的计算机上工作。由于这在PC平台上经常发生，通过支持中断共享，您可以避免很多麻烦。

### 使用uiopdrv平台设备

在许多情况下，平台设备的UIO驱动程序可以以通用方式处理。您只需要在定义struct platform\_device的地方实现您的中断处理程序并填写struct uio\_info。此struct uio\_info向您的平台设备提供了一个指针。

您还需要设置一个包含您的内存映射地址和大小的struct resource数组。这些信息通过struct platform\_device的.resource和.num\_resources元素传递给驱动程序。

现在，您必须将struct platform\_device的.name元素设置为“uio\_pdrv”，以使用通用UIO平台设备驱动程序。该驱动程序将根据给定的资源填充mem[]数组，并注册设备。

这种方法的优点是您只需要编辑一个需要编辑的文件。您不需要创建额外的驱动程序。

### 使用uiopdrv\_genirq平台设备

特别是在嵌入式设备中，您经常会发现将中断引脚连接到其自己的专用中断线上的芯片。在这种情况下，您可以非常确定地中断没有被共享，我们可以将uio\_pdrv的概念推进一步并使用通用中断处理程序。这就是uio\_pdrv\_genirq的作用。

此驱动程序的设置与uio\_pdrv说明的相同，不同之处在于您不需要实现中断处理程序。struct uio\_info的.handler元素必须保持为NULL。.irq\_flags元素不能包含IRQF\_SHARED。

您将struct platform\_device的.name元素设置为“uio\_pdrv\_genirq”，以使用此驱动程序。

uio\_pdrv\_genirq的通用中断处理程序将使用disable\_irq\_nosync()禁用中断线。完成工作后，用户空间可以通过向UIO设备文件写入0x00000001来重新启用中断。驱动程序已经实现了irq\_control()，使其成为可能，您不必自己实现。

使用uio\_pdrv\_genirq不仅可以节省几行中断处理程序代码。您还不需要了解芯片的内部寄存器，以创建驱动程序的内核部分。您只需知道芯片连接到的引脚的中断号。

在设备树启用的系统中使用时，需要使用“of\_id”模块参数将驱动程序探测到要处理的节点的“compatible”字符串。默认情况下，节点的名称（不包括单元地址）在用户空间中作为UIO设备的名称公开。要设置自定义名称，可以在DT节点中指定名为“linux，uio-name”的属性。

### 使用uio\_dmem\_genirq平台设备

除了静态分配的内存范围外，还可能希望在用户空间驱动程序中使用动态分配的区域。特别是，能够访问通过dma-mapping API提供的内存可能特别有用。uio\_dmem\_genirq驱动程序提供了一种实现这一点的方法。

此驱动程序的使用方式类似于“uio\_pdrv\_genirq”驱动程序，与中断配置和处理有关。

将struct platform\_device的.name元素设置为“uio\_dmem\_genirq”以使用此驱动程序。

使用此驱动程序时，请填写struct platform\_device的.platform\_data元素，该元素是struct uio\_dmem\_genirq\_pdata类型，并包含以下元素：

struct uio\_info uioinfo：与uio\_pdrv\_genirq平台数据使用相同的结构

unsigned int \*dynamic\_region\_sizes：指向要映射到用户空间的动态内存区域大小列表的指针。

unsigned int num\_dynamic\_regions：dynamic\_region\_sizes数组中的元素数。

在平台数据中定义的动态区域将附加到平台设备资源的“ mem []”数组后面，这意味着静态和动态内存区域的总数不能超过MAX\_UIO\_MAPS。

当打开UIO设备文件“ / dev / uioX”时，将分配动态内存区域。与静态内存资源类似，动态区域的内存区域信息可通过sysfs在“ / sys / class / uio / uioX / maps / mapY / \*”中看到。 UIO设备文件关闭时，动态内存区域将被释放。当没有进程持有设备文件时，返回给用户空间的地址为~0。

## 在用户空间编写**驱动**程序

使用硬件的工作内核模块后，您可以编写驱动程序的用户空间部分。您不需要任何特殊的库，驱动程序可以用任何合理的语言编写，可以使用浮点数等。简而言之，您可以使用编写用户空间应用程序的所有工具和库。

### 获取有关UIO设备的信息

有关所有UIO设备的信息都可以在sysfs中找到。在驱动程序中的第一件事是检查名称和版本，以确保您正在与正确的设备交互，并且其内核驱动程序具有您期望的版本。

您还应确保所需的内存映射存在并且具有您期望的大小。

有一个名为lsuio的工具，它列出了UIO设备及其属性。它在这里可用：

http://www.osadl.org/projects/downloads/UIO/user/

使用lsuio，您可以快速检查您的内核模块是否已加载以及其导出的属性。有关详细信息，请参阅手册页。

lsuio的源代码可以作为获取有关UIO设备信息的示例。文件uio\_helper.c包含您可以在用户空间驱动程序代码中使用的许多函数。

### mmap（）设备内存

在确定您具有所需的内存映射且其大小符合预期后，您所要做的就是调用mmap（）将设备的内存映射到用户空间。

mmap（）调用的偏移量参数对于UIO设备具有特殊含义：它用于选择要映射的设备映射。要映射映射N的内存，您必须使用N倍的页面大小作为偏移量：

offset = N \* getpagesize();

N从零开始，因此，如果您只有一个要映射的内存范围，请将偏移量设置为0。这种技术的缺点是内存始终以其起始地址开头映射。

### 等待中断

在成功映射设备的内存之后，您可以像普通数组一样访问它。通常，您将执行一些初始化。之后，您的硬件开始工作，并会在完成、有可用数据或需要您的关注因为发生错误时生成中断。

/dev/uioX是只读文件。读取（）将始终阻塞，直到发生中断。对于read（）的计数参数，只有一个合法值，即带符号32位整数的大小（4）。计数值为任何其他值都会导致read（）失败。带符号的32位整数读取是您设备的中断计数。如果该值比您上次读取的值多一个，则一切正常。如果差异大于1，则您错过了中断。

您还可以在/dev/uioX上使用select（）。

## 通用PCI UIO驱动程序

通用驱动程序是一个名为uio\_pci\_generic的内核模块。它可以与任何符合PCI 2.3（约2002年）和任何符合PCI Express设备的设备一起工作。使用此功能，您只需要编写用户空间驱动程序，无需编写特定于硬件的内核模块。

### 使驱动程序识别设备

由于驱动程序未声明任何设备ID，因此它不会自动加载，也不会自动绑定任何设备，您必须自己加载并为驱动程序分配ID。例如:

modprobe uio\_pci\_generic

echo "8086 10f5" > /sys/bus/pci/drivers/uio\_pci\_generic/new\_id

如果您的设备已经有了特定于硬件的内核驱动程序，则通用驱动程序仍然不会绑定到它，如果您要使用通用驱动程序（为什么会？）在这种情况下，您必须手动解除绑定硬件特定的驱动程序并绑定通用驱动程序，例如:

echo -n 0000:00:19.0 > /sys/bus/pci/drivers/e1000e/unbind

echo -n 0000:00:19.0 > /sys/bus/pci/drivers/uio\_pci\_generic/bind

您可以在sysfs中查找设备是否已绑定驱动程序，例如以下内容:

ls -l /sys/bus/pci/devices/0000:00:19.0/driver

如果成功，应该打印:

.../0000:00:19.0/driver -> ../../../bus/pci/drivers/uio\_pci\_generic

请注意，通用驱动程序不会绑定到旧PCI 2.2设备。如果设备绑定失败，请运行以下命令:

dmesg

并在输出中查找故障原因。

### 关于uio\_pci\_generic的注意事项

使用PCI命令寄存器中的中断禁用位和PCI状态寄存器中的中断状态位处理中断。符合PCI 2.3（约2002年）的所有设备和符合PCI Express设备应支持这些位。uio\_pci\_generic检测此支持，并不会绑定到不支持命令寄存器中的中断禁用位的设备。

在每次中断时，uio\_pci\_generic设置中断禁用位。这将防止设备生成更多中断，直到该位被清除。用户空间驱动程序应在阻塞和等待更多中断之前清除此位。

### 使用**uio**\_pci\_generic编写用户空间驱动程序

用户空间驱动程序可以使用pci sysfs接口或包装它的libpci库来与设备通信，并通过写入命令寄存器来重新启用中断。

### 使用uio\_pci\_generic的示例代码

以下是使用uio\_pci\_generic的一些示例用户空间驱动程序代码:

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <errno.h>

int main()

{

int uiofd;

int configfd;

int err;

int i;

unsigned icount;

unsigned char command\_high;

uiofd = open("/dev/uio0", O\_RDONLY);

if (uiofd < 0) {

perror("uio open:");

return errno;

}

configfd = open("/sys/class/uio/uio0/device/config", O\_RDWR);

if (configfd < 0) {

perror("config open:");

return errno;

}

/\* Read and cache command value \*/

err = pread(configfd, &command\_high, 1, 5);

if (err != 1) {

perror("command config read:");

return errno;

}

command\_high &= ~0x4;

for(i = 0;; ++i) {

/\* Print out a message, for debugging. \*/

if (i == 0)

fprintf(stderr, "Started uio test driver.\n");

else

fprintf(stderr, "Interrupts: %d\n", icount);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Here we got an interrupt from the

device. Do something to it. \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Re-enable interrupts. \*/

err = pwrite(configfd, &command\_high, 1, 5);

if (err != 1) {

perror("config write:");

break;

}

/\* Wait for next interrupt. \*/

err = read(uiofd, &icount, 4);

if (err != 4) {

perror("uio read:");

break;

}

}

return errno;

}

## 通用Hyper-V UIO驱动程序

通用驱动程序是名为uio\_hv\_generic的内核模块。它支持类似于PCI总线上的uio\_pci\_generic的Hyper-V VMBus上的设备。

### 使驱动程序识别设备

由于驱动程序不声明任何设备GUID，因此它不会自动加载并且不会自动绑定到任何设备，您必须自己加载它并为驱动程序分配ID。例如，要使用网络设备类GUID:

modprobe uio\_hv\_generic

echo "f8615163-df3e-46c5-913f-f2d2f965ed0e" > /sys/bus/vmbus/drivers/uio\_hv\_generic/new\_id

如果设备已经有硬件特定的内核驱动程序，则通用驱动程序仍不会绑定到它。在这种情况下，如果您想为用户空间库使用通用驱动程序，您必须手动取消绑定硬件特定的驱动程序并绑定通用驱动程序，并使用设备特定的GUID，像这样:

echo -n ed963694-e847-4b2a-85af-bc9cfc11d6f3 > /sys/bus/vmbus/drivers/hv\_netvsc/unbind

echo -n ed963694-e847-4b2a-85af-bc9cfc11d6f3 > /sys/bus/vmbus/drivers/uio\_hv\_generic/bind

您可以在sysfs中查找设备是否已绑定驱动程序，例如以下内容:

ls -l /sys/bus/vmbus/devices/ed963694-e847-4b2a-85af-bc9cfc11d6f3/driver

如果成功，应该打印:

.../ed963694-e847-4b2a-85af-bc9cfc11d6f3/driver -> ../../../bus/vmbus/drivers/uio\_hv\_generic

### 有关uio\_hv\_generic的注意事项

在每次中断时，uio\_hv\_generic设置中断禁用位。这将防止设备生成更多中断，直到该位被清除。用户空间驱动程序应在阻塞和等待更多中断之前清除此位。

当主机撤销设备时，中断文件说明符会被标记为下降，并且中断文件说明符的任何读取都将返回-EIO。类似于关闭的套接字或断开连接的串行设备。

#### **vmbus设备区域映射到uio设备资源**：

1. 通道环形缓冲区：来宾到主机和主机到来宾
2. 来宾到主机中断信令页
3. 来宾到主机监视器页面
4. 网络接收缓冲区域
5. 网络发送缓冲区域

如果通过对主机的请求创建子通道，则uio\_hv\_generic设备驱动程序将为每个通道环形缓冲区创建一个sysfs二进制文件。例如:

/sys/bus/vmbus/devices/3811fe4d-0fa0-4b62-981a-74fc1084c757/channels/21/ring

## 更多信息

[OSADL 主页.](http://www.osadl.org/)

[Linutronix 主页.](http://www.linutronix.de/)

# Linux固件 API

## 介绍

固件API使内核代码能够从用户空间请求所需的文件，用途各不相同：

CPU勘误的微代码

需要加载到设备微控制器上的设备驱动程序固件

设备驱动程序信息数据（校准数据，EEPROM覆盖），其中一些可以完全是可选的。

### 固件请求类型

有两种调用类型：

同步

异步

您使用哪种取决于您的要求，但经验法则是，除非您已经使用异步初始化机制而不会停止或延迟引导，否则应努力使用异步API。即使加载固件不需要很长时间，处理固件可能会，这仍然可能会延迟引导或初始化，因此诸如异步探测的机制可以帮助补充驱动程序。

## 固件API核心功能

固件API具有丰富的核心功能可用。本节记录这些功能。

### 固件搜索路径

以下搜索路径用于在root文件系统上查找固件：

fw\_path\_para - 模块参数 - 默认为空，因此被忽略

/lib/firmware/updates/UTS\_RELEASE/

/lib/firmware/updates/

/lib/firmware/UTS\_RELEASE/

/lib/firmware/

可以传递‘path’模块参数以激活第一个可选的自定义fw\_path\_para。自定义路径只能长达256个字符。传递的内核参数将是:

1）‘firmware\_class.path=$CUSTOMIZED\_PATH’

有另一种方法可以在启动后运行时自定义路径，您可以使用文件:

2）/sys/module/firmware\_class/parameters/path

您将在其中回显您的自定义路径，并且首先会在那里搜索请求的固件。

### 内置固件

固件可以内置到内核中，这意味着将固件直接构建到vmlinux中，以避免从文件系统中查找固件。相反，可以直接在内核中查找固件。您可以使用内核配置选项启用内置固件:

CONFIG\_EXTRA\_FIRMWARE

CONFIG\_EXTRA\_FIRMWARE\_DIR

以下是您可能要考虑使用CONFIG\_EXTRA\_FIRMWARE构建固件内核的原因：

1. 速度
2. 需要固件访问引导设备，用户不希望将固件塞入引导initramfs中。

即使您有这些需求，您可能无法利用内置固件的几个原因：

1. 法律条款-固件不兼容GPL
2. 一些固件可能是可选的
3. 固件升级是可能的，因此新固件将涉及完全的内核重建。
4. 某些固件文件可能非常大。 remote-proc子系统是处理这些固件的示例子系统
5. 固件可能需要从某些设备特定位置动态刮出，例如某些WiFi芯片组的校准数据。此校准数据可以针对每个售出的设备都是唯一的。

### 固件缓存

当Linux从挂起状态恢复时，某些设备驱动程序需要进行固件查找以重新初始化设备。在恢复期间，可能有一段时间无法进行固件查找，在此短暂时间内，固件请求将失败。时间至关重要，而让驱动程序等待根文件系统获取固件会延迟用户对设备功能的使用体验。为了支持这些需求，固件基础架构为大多数API调用的设备驱动程序实现了固件缓存，在幕后自动完成。

固件缓存使使用某些固件API调用在设备驱动程序的挂起和恢复回调期间是安全的。使用这些API调用的用户不需要自己为处理系统恢复期间固件丢失而缓存该固件。

固件缓存的一些实现细节：

1. 通过为每个使用除了request\_firmware\_into\_buf()之外的所有同步调用的设备添加devres条目来设置固件缓存。
2. 如果使用异步调用，则仅在请求\_firmware\_nowait()的第二个参数（uevent）为true时才会为设备设置固件缓存。当uevent为true时，它请求对固件请求通过sysfs回退机制发送到用户空间的kobject uevent，如果未找到固件文件，则会发送uevent。
3. 如果根据上述两个标准确定需要固件缓存，则通过为进行固件请求的设备添加devres条目来设置固件缓存。
4. 固件devres条目在设备的整个生命周期中保持不变。这意味着即使您释放固件缓存，从挂起状态恢复时仍将使用该固件缓存
5. 回退机制的超时时间在挂起期间暂时降低到10秒，因为在挂起期间设置固件缓存，缓存设置完成后，超时时间会恢复到您配置的旧值。
6. 在挂起期间，任何未处理的非uevent固件请求都将被中止以避免阻塞内核，这是通过kill\_requests\_without\_uevent()完成的。因此需要实现自己的固件缓存机制的内核调用在挂起时不能使用固件API。

### 直接文件系统查找

直接文件系统查找是内核执行的最常见的固件查找形式。内核直接在根文件系统的路径中寻找固件，这些路径在“固件搜索路径”节中有所记录。文件系统查找是在fw\_get\_filesystem\_firmware()中实现的，它使用通用的核心内核文件加载器设施kernel\_read\_file\_from\_path()。最大路径允许是PATH\_MAX-目前这是4096个字符。

建议您在根文件系统上保留/lib/firmware路径，避免将它们放在单独的分区中，以避免可能的查找竞争，并避免使用下面所述的自定义回退机制。

#### 固件和initramfs

内置于内核中的驱动程序应该将固件集成到用于启动内核的initramfs中，因为否则可能会在加载驱动程序并且真正的根文件系统还不可用时出现竞争。将固件插入到initramfs中可以解决这个竞争问题，但要注意使用initrd不能解决相同的竞争。

有一些情况可以证明不想将固件包含在initramfs中，例如处理远程处理子系统的大型固件文件。对于这种情况，使用用户空间回退机制目前是唯一可行的解决方案，因为只有用户空间才能确定真正的根文件系统何时准备好并挂载。

### 回退机制

支持回退机制以允许克服在根文件系统上进行直接文件系统查找或者由于实际原因无法安装固件的故障。与支持固件回退机制相关的内核配置选项包括：

CONFIG\_FW\_LOADER\_USER\_HELPER：启用构建固件回退机制。大多数发行版今天都启用了这个选项。如果启用但未启用CONFIG\_FW\_LOADER\_USER\_HELPER\_FALLBACK，则仅可用自定义回退机制和request\_firmware\_nowait（）调用。

CONFIG\_FW\_LOADER\_USER\_HELPER\_FALLBACK：强制每个请求启用kobject uevent回退机制，但对于除request\_firmware\_direct（）以外的所有固件API调用。大多数发行版今天都禁用此选项。调用request\_firmware\_nowait（）允许一种备用回退机制：如果启用了此kconfig选项，并且request\_firmware\_nowait（）的第二个参数uevent设置为false，则您正在告知内核您有一个自定义回退机制，它将手动加载固件。阅读以下内容以获取更多详细信息。

请注意，这意味着在具有此配置时：

CONFIG\_FW\_LOADER\_USER\_HELPER=y CONFIG\_FW\_LOADER\_USER\_HELPER\_FALLBACK=n

kobject uevent回退机制甚至不会对request\_firmware\_nowait（）在uevent设置为true时发挥作用。

#### 证明固件回退机制

直接文件系统查找可能因多种原因而失败。值得列举和记录已知原因，因为这证明了需要回退机制的必要性：

与启动时根文件系统的访问竞争。

在从挂起状态恢复时有闪存。这通过固件缓存解决，但只有在使用uevents时才支持固件缓存，而且它不支持request\_firmware\_into\_buf（）。

固件无法通过典型方式进行访问：

1. 它无法安装到根文件系统中。
2. 固件提供了为所收集的本地信息调整的非常独特的设备特定数据。一个例子是用于移动设备的WiFi芯片组的校准数据。这些校准数据并不适用于所有单位，而是按照单位进行定制。此类信息可能安装在与提供根文件系统的不同闪存分区上。

#### 回退机制类型

实际上有两种可用的回退机制，使用一个共享的sysfs界面作为加载设施：

Kobject uevent回退机制

自定义回退机制

首先记录共享的sysfs加载设施。

#### 固件sysfs加载设备

为了帮助设备驱动程序使用回退机制上传固件，固件基础架构创建了一个sysfs接口，以使用户空间可以加载并指示固件何时准备就绪。通过fw\_create\_instance（）创建sysfs目录。该调用创建以所请求的固件命名的新struct device，并将其与用于发出请求的设备关联起来，作为该设备的父设备。sysfs目录的文件属性通过新设备的类别（firmware\_class）和组（fw\_dev\_attr\_groups）定义和控制。这实际上是原始固件类模块名称的来源，因为最初可用的唯一固件加载机制是我们现在使用的回退机制，它注册了一个struct class firmware\_class。因为所公开的属性是模块名称的一部分，所以模块名称firmware\_class不能在未来重命名，以确保与旧的用户空间向后兼容。

为了使用sysfs接口加载固件，我们公开了一个加载指示器和一个上传固件的文件以下是您可能要考虑使用CONFIG\_EXTRA\_FIRMWARE构建固件内核的原因：

/sys/$DEVPATH/loading

/sys/$DEVPATH/data

要上传固件，您需要将1回声到loading文件中，以表示您正在加载固件。然后将固件写入数据文件，并通过将0回声到loading文件中来通知内核固件已准备就绪。

仅当直接固件加载失败并且针对您的固件请求启用了回退机制时，才会使用固件设备来帮助使用sysfs加载固件。这是通过firmware\_fallback\_sysfs()设置的。需要重申的是，如果直接文件系统查找成功，则不会创建任何设备。

使用以下命令:

echo 1 > /sys/$DEVPATH/loading

将立即清除任何以前的部分加载，并使固件API返回错误。加载固件时，firmware\_class会增加一个缓冲区来容纳image的大小。它以PAGE\_SIZE增量增加，以便在image到达时保持其大小。

firmware\_data\_read()和firmware\_loading\_show()仅提供给test\_firmware驱动程序进行测试，正常使用中不会调用它们，也不希望用户空间经常使用它们。

**firmware\_fallback\_sysfs**

1）int firmware\_fallback\_sysfs(struct firmware \*fw, const char \*name, struct [device](https://www.kernel.org/doc/html/latest/driver-api/firmware/fallback-mechanisms.html" \l "c.firmware_fallback_sysfs" \o "device) \*device, u32 opt\_flags, int ret)

使用回退机制查找固件

参数

struct firmware \*fw

指向固件镜像的指针

const char \*name

要查找的固件文件的名称

struct device \*device

正在加载固件的设备

u32 opt\_flags

按枚举fw\_opt定义的选项，控制固件加载行为

int ret

直接查找的返回，触发了回退机制

说明

如果直接查找固件失败，将调用此函数，通过公开sysfs loading接口启用回退机制。用户空间负责通过sysfs loading接口加载固件。可以通过将proc sysctl值ignore\_sysfs\_fallback设置为true来完全禁用此sysfs回退机制。如果是false，则检查内部API调用方是否设置了FW\_OPT\_NOFALLBACK\_SYSFS标志，如果是，则还将禁用回退机制。系统可能希望始终强制使用sysfs回退机制，它可以通过将ignore\_sysfs\_fallback设置为false和force\_sysfs\_fallback设置为true来实现。启用force\_sysfs\_fallback在功能上等同于使用CONFIG\_FW\_LOADER\_USER\_HELPER\_FALLBACK构建内核。

#### 固件kobject uevent回退机制

由于为sysfs接口创建了一个设备以帮助加载固件作为回退机制，因此可以依靠kobject uevent来通知用户空间已添加设备。将设备添加到设备层次结构中意味着已启动固件加载的回退机制。有关实施细节，请参阅fw\_load\_sysfs\_fallback（），特别是关于dev\_set\_uevent\_suppress（）和kobject\_uevent（）的使用。

内核的kobject uevent机制实现在lib / kobject\_uevent.c中，它向用户空间发出uevent。作为kobject uevent的补充，Linux发行版也可以启用CONFIG\_UEVENT\_HELPER\_PATH，这利用了核心内核的用户模式助手（UMH）功能，以调用用户空间助手来处理kobject uevent。但是，实际上没有任何标准发行版使用CONFIG\_UEVENT\_HELPER\_PATH。如果启用了CONFIG\_UEVENT\_HELPER\_PATH，则每次内核中为每个kobject uevent调用kobject\_uevent\_env（）时，都会调用此二进制文件。

不同的实现已经得到了支持，以利用此回退机制。当只能使用sysfs机制加载固件时，用户空间组件“hotlug”提供了监视kobject事件的功能。历史上，它已被systemd的udev取代，但是自2014年8月systemd提交be2ea723b1d0（“udev：删除用户空间固件加载支持”）以来，从udev中删除了固件加载支持。这意味着今天大多数Linux发行版都未使用或利用由kobject uevent提供的固件回退机制。这特别加剧了因为今天大多数发行版都禁用了CONFIG\_FW\_LOADER\_USER\_HELPER\_FALLBACK。

有关kobject事件变量设置的详细信息，请参阅do\_firmware\_uevent（）。当前传递给“kobject add”事件的变量是:

FIRMWARE=固件名称

TIMEOUT=超时值

ASYNC=请求是否异步

默认情况下，DEVPATH由内部内核kobject基础架构设置。以下是一个示例简单的kobject uevent脚本:

# Both $DEVPATH and $FIRMWARE are already provided in the environment.

MY\_FW\_DIR=/lib/firmware/

echo 1 > /sys/$DEVPATH/loading

cat $MY\_FW\_DIR/$FIRMWARE > /sys/$DEVPATH/data

echo 0 > /sys/$DEVPATH/loading

#### 固件自定义回退机制

使用request\_firmware\_nowait（）调用的用户还有另一种选择：依靠sysfs回退机制，但要求不发出kobject uevent到用户空间。最初的逻辑是除udev之外的其他实用程序可能需要在非传统路径中查找固件-超出“直接文件系统查找”部分所记录的列表之外的路径。对于其他API调用，此选项不可用，因为始终强制执行uevents。

由于只有启用内核中的回退机制，uevents才具有意义，因此启用uevents的内核似乎很奇怪，这些内核在其内核中未启用回退机制。不幸的是，我们还依赖uevent标志，该标志可以通过request\_firmware\_nowait()禁用，以设置固件请求的固件缓存。如上所述，仅当启用uevent进行API调用时，才设置固件缓存。虽然这可以禁用request\_firmware\_nowait()调用的固件缓存，但这个API的用户不应该将其用于禁用缓存，因为这不是该标志的原始目的。不设置uevent标志意味着您希望选择固件回退机制，但希望禁止kobject uevents，因为您有一个自定义解决方案，将以某种方式监视将设备添加到设备层次结构中，并通过自定义路径为您加载固件。

#### 固件回退超时

固件回退机制具有超时时间。如果在超时值之前未将固件加载到sysfs接口上，则会向驱动程序发送错误。默认情况下，如果需要uevents，则将超时设置为60秒，否则使用MAX\_JIFFY\_OFFSET（可能的最大超时）。使用MAX\_JIFFY\_OFFSET进行非uevent的逻辑是，自定义解决方案将拥有加载固件所需的时间。

您可以通过将所需的超时回声到以下文件中来自定义固件超时时间:

/sys/class/firmware/timeout

如果将其回声为0，则意味着将使用MAX\_JIFFY\_OFFSET。超时的数据类型是int。

### 固件查找顺序

不同的功能可用于启用找到固件。以下是驱动程序发出固件API调用后查找固件的时间顺序。

首先检查内置固件，如果固件存在，则立即返回

接下来查看“固件缓存”。如果找到固件，则立即返回

然后执行“直接文件系统查找”，如果找到，则立即返回

如果未找到固件并且启用了回退机制，则创建sysfs接口。之后，要么发出一个kobject uevent，要么在超时值之前依赖于自定义固件加载来加载固件。

## request\_firmware API

通常，您会加载固件，然后将其加载到设备中。典型的固件工作流如下所示:

if(request\_firmware(&fw\_entry, $FIRMWARE, device) == 0)

copy\_fw\_to\_device(fw\_entry->data, fw\_entry->size);

release\_firmware(fw\_entry);

### 同步固件请求

同步固件请求将等待固件被找到或返回错误

#### request\_firmware

**int request\_firmware(const struct firmware \*\*firmware\_p, const char \*name, struct [device](https://www.kernel.org/doc/html/latest/driver-api/firmware/request_firmware.html" \l "c.request_firmware" \o "device) \*device)**

发送固件请求并等待它

1）参数

const struct firmware \*\*firmware\_p

指向固件映像的指针

const char \*name

固件文件的名称

struct device \*device

正在加载固件的设备

2）说明

firmware\_p将用于通过名称为名称的固件映像返回设备device。

应从允许休眠的用户上下文中调用。

名称将用作uevent环境中的$FIRMWARE，并且应足够独特，以便不会与此设备或任何其他设备的任何其他固件映像混淆。

调用方必须持有设备的引用计数。

该函数可以安全地调用设备的挂起和恢复回调内。

#### firmware\_request\_nowarn

**int firmware\_request\_nowarn(const struct [firmware](https://www.kernel.org/doc/html/latest/driver-api/firmware/request_firmware.html" \l "c.firmware_request_nowarn" \o "firmware) \*\*firmware, const char \*name, struct [device](https://www.kernel.org/doc/html/latest/driver-api/firmware/request_firmware.html" \l "c.firmware_request_nowarn" \o "device) \*device)**

请求可选的固件模块

1）参数

const struct firmware \*\*firmware

指向固件映像的指针

const char \*name

固件文件的名称

struct device \*device

正在加载固件的设备

2）说明

此函数与request\_firmware()的行为类似，但在找不到文件时不会产生警告消息。如果直接文件系统查找失败，则启用sysfs回退机制。但是，如果仍然找不到固件文件，则失败仍将被抑制。因此，由驱动程序检查此调用的返回，并决定何时通知用户错误。

#### request\_firmware\_direct

**int request\_firmware\_direct(const struct firmware \*\*firmware\_p, const char \*name, struct [device](https://www.kernel.org/doc/html/latest/driver-api/firmware/request_firmware.html" \l "c.request_firmware_direct" \o "device) \*device)**

在没有用户模式帮手的情况下直接加载固件

1）参数

const struct firmware \*\*firmware\_p

指向固件映像的指针

const char \*name

固件文件的名称

struct device \*device

正在加载固件的设备

2）说明

此函数的工作方式与request\_firmware()类似，但即使无法直接从fs加载固件，它也不会回退到用户模式帮助程序。因此，它适用于加载可选固件（并非始终存在）而不会产生额外长的udev超时。

#### request\_firmware\_into\_buf

**int request\_firmware\_into\_buf(const struct firmware \*\*firmware\_p, const char \*name, struct [device](https://www.kernel.org/doc/html/latest/driver-api/firmware/request_firmware.html" \l "c.request_firmware_into_buf" \o "device) \*device, void \*buf, size\_t size)**

将固件加载到预先分配的缓冲区中

1）参数

const struct firmware \*\*firmware\_p

固件镜像指针

const char \*name

固件文件名

struct device \*device

加载固件和DMA区域分配的设备

void \*buf

要将固件加载到的缓冲区的地址

size\_t size

缓冲区大小

2）说明

该函数的功能与request\_firmware()基本相同，但它不分配用于容纳固件数据的缓冲区。相反，固件直接加载到buf指向的缓冲区中，并且firmware\_p数据成员指向buf。

该函数也不缓存固件。

### 异步固件请求

异步固件请求允许驱动程序代码不必等待固件或错误返回。提供函数回调，以便在找到固件或错误时通过回调通知驱动程序。request\_firmware\_nowait()不能在原子上下文中调用。

#### request\_firmware\_nowait

**int request\_firmware\_nowait(struct [module](https://www.kernel.org/doc/html/latest/driver-api/firmware/request_firmware.html" \l "c.request_firmware_nowait" \o "module) \*module, bool uevent, const char \*name, struct [device](https://www.kernel.org/doc/html/latest/driver-api/firmware/request_firmware.html" \l "c.request_firmware_nowait" \o "device) \*device, gfp\_t gfp, void \*context, void (\*cont)(const struct firmware \*fw, void \*context))**

request\_firmware的异步版本

1）参数

struct module \*module

请求固件的模块

bool uevent

如果该标志为非零，则发送uevent以复制固件映像，否则必须手动进行固件复制。

const char \*name

固件文件名

struct device \*device

正在加载固件的设备

gfp\_t gfp

分配标志

void \*context

将传递给cont，并且如果固件请求失败，则fw可能为NULL。

void（\*cont）（const struct firmware \*fw，void \*context）

异步调用该函数时会在固件请求结束时调用。

2）说明

调用者必须保持设备的引用计数。

针对用户上下文的request\_firmware()的异步变体：

如果gfp是GFP\_KERNEL，则尽可能短地休眠，因为它可能会增加内核引导时间，在内置设备驱动程序在其->probe（）方法中请求固件时。

如果gfp是GFP\_ATOMIC，则不能休眠。

### 关于重启的特殊优化

某些设备具有优化功能，可以在系统重新启动期间保留固件。当使用这种优化时，驱动程序作者必须确保在从挂起状态恢复期间仍可用固件，这可以使用firmware\_request\_cache()而不是请求加载固件来实现。

#### firmware\_request\_cache()

**int firmware\_request\_cache(struct [device](https://www.kernel.org/doc/html/latest/driver-api/firmware/request_firmware.html" \l "c.firmware_request_cache" \o "device) \*device, const char \*name)**

缓存挂起的固件，以便恢复可以使用它

1）参数

struct device \*device

应为其缓存固件的设备

const char \*name

固件文件名

2）说明

某些设备具有优化功能，可以使设备在系统重新启动时不需要加载固件。这种优化在从挂起状态恢复时仍可能需要固件。在这些情况下，可以使用此例程确保在从挂起状态恢复期间固件是可用的。此助手不与使用request\_firmware\_into\_buf()或request\_firmware\_nowait()而未设置uevent的驱动程序兼容。

### 请求固件API预期的驱动程序使用

一旦API调用返回，您就可以处理固件，然后释放固件。例如，如果您使用了request\_firmware()并且它返回，则驱动程序可以在fw\_entry-> {data，size}中访问固件映像。如果出现问题，request\_firmware()返回非零值，并将fw\_entry设置为NULL。一旦您的驱动程序完成固件处理，它便可以调用release\_firmware（fw\_entry）以释放固件映像和任何相关资源。

## 其他固件接口

### DMI接口

#### int dmi\_check\_system(const struct dmi\_system\_id \*list)

检查系统DMI数据

1）参数

const struct dmi\_system\_id \*list

要与之匹配的dmi\_system\_id结构数组。列表的所有非空元素都必须匹配其插槽（字段索引）的数据（即，列表字符串必须是指定DMI插槽字符串数据的子字符串），才能被认为是成功匹配。

说明

运行匹配功能以遍历黑名单表，直到有人返回非零或我们到达结尾。每个成功匹配都会调用回调函数。返回匹配数。

在调用此函数之前，必须调用dmi\_scan\_machine。

#### const struct dmi\_system\_id \*dmi\_first\_match(const struct dmi\_system**\_**id \*list)

查找与系统DMI数据匹配的dmi\_system\_id结构

1）参数

const struct dmi\_system\_id \*list

要与之匹配的dmi\_system\_id结构数组。列表的所有非空元素都必须匹配其插槽（字段索引）的数据（即，列表字符串必须是指定DMI插槽字符串数据的子字符串），才能被认为是成功匹配。

2）说明

遍历黑名单表，直到找到第一个匹配项。返回指向匹配条目的指针，如果没有匹配项，则返回NULL。

在调用此函数之前，必须调用dmi\_setup。

#### const char \*dmi\_get\_system\_info(int field)

返回DMI数据值

参数

int field

数据索引（参见枚举dmi\_field）

说明

返回一个DMI数据值，可用于执行复杂的DMI数据检查。

#### int dmi\_name\_in\_vendors(const char \*str)

检查字符串是否在DMI系统或板供应商名称中

参数

const char \*str

区分大小写的名称

#### const struct dmi\_device \*dmi\_find\_device(int type, const char \*name, const struct dmi\_device \*from)

按类型/名称查找板载设备

参数

int type

设备类型或DMI\_DEV\_TYPE\_ANY，以匹配所有设备类型

const char \*name

设备名称字符串或NULL以匹配所有设备

const struct dmi\_device \*from

在搜索中找到的上一个设备，或在新搜索中为NULL。

说明

遍历已知的板载设备列表。如果找到具有匹配类型和名称的设备，则返回指向其设备结构的指针。否则，返回NULL。通过将from参数设置为NULL来启动新搜索。如果from不为NULL，则从下一个设备继续搜索。

#### bool dmi\_get\_date(int field, int \*yearp, int \*monthp, int \*dayp)

解析DMI日期

参数

int field

数据索引（参见枚举dmi\_field）

int \*yearp

年份的可选输出参数

int \*monthp

月份的可选输出参数

int \*dayp

日期的可选输出参数

说明

日期字段假定类似于[mm[/dd]]/yy[yy]的形式，并将结果存储在输出参数中，任何或所有输出参数均可省略。

如果字段不存在，则所有输出参数设置为零并返回false。否则，返回true，并将日期的任何无效部分设置为零。

在返回时，年、月和日保证分别在[0,9999]、[0,12]和[0,31]范围内。

#### int dmi\_get\_bios\_year(void)

从DMI\_BIOS\_DATE字段中得到年份

参数

void

无参数

说明

成功返回年份，如果未选择DMI，则返回-ENXIO，如果DMI字段不存在或无法解析，则返回不同的负错误代码。

#### int dmi\_walk(void (\*decode)(const struct dmi\_header\*, void\*), void \*private\_data)

遍历DMI表，并为每个记录回调

参数

void (\*decode)(const struct dmi\_header \*, void \*)

回调函数

void \*private\_data

要传递给回调函数的私有数据

说明

成功返回0，如果未选择DMI或未找到则返回-ENXIO，如果DMI遍历失败，则返回不同的负错误代码。

#### bool dmi\_match(enum dmi\_field f, const char \*str)

将字符串与DMI字段（如果存在）进行比较

参数

enum dmi\_field f

DMI字段标识符

const char \*str

要将DMI字段与其进行比较的字符串

说明

如果所请求的字段等于str（包括NULL），则返回true。

### EDD接口

#### ssize\_t edd\_show\_raw\_data(struct edd\_device \*edev, char \*buf)

将原始数据复制到缓冲区，以供用户空间解析

参数

struct edd\_device \*edev

目标edd\_device

char \*buf

输出缓冲区

返回

写入的字节数，或失败的-EINVAL

#### void edd\_release(struct kobject \*kobj)

释放edd结构

参数

struct kobject \* kobj

edd结构的kobject

说明

当edd结构的引用计数达到0时调用此函数。这应该在我们取消注册之后立即发生，但以防万一，我们仍然使用释放回调。

#### int edd\_dev\_is\_type(struct edd\_device \*edev, const char \*type)

此EDD设备是否为“类型”设备？

参数

struct edd\_device \*edev

目标edd\_device

const char \*type

每个EDD规范的主机总线或接口标识符字符串

说明

如果是“类型”设备，则返回1（TRUE），否则返回0。

#### struct pci\_dev \*edd\_get\_pci\_dev(struct edd\_device \*edev)

找到与edd匹配的pci\_dev

参数

struct edd\_device \*edev

edd\_device

说明

如果找到，返回pci\_dev，否则返回NULL

#### int edd\_init(void)

创建EDD数据的sysfs树

参数

void

无参

# PINCTRL (引脚控制) 子系统

本文档概述了Linux中的引脚控制子系统

该子系统处理：

枚举和命名可控引脚

引脚、焊盘等的多路复用，有关详细信息请参见下面的详细信息

引脚、焊盘等的配置，例如软件控制的偏置和驱动模式特定引脚，例如上拉/下拉，开漏，负载电容等。

## 顶层接口

引脚控制器定义：

引脚控制器是一种硬件，通常是一组寄存器，可控制引脚。它可能能够多路复用，偏置，设置负载电容，设置驱动强度等，针对单个引脚或一组引脚。

引脚定义：

引脚等同于焊盘、导线、球或任何要控制的封装输入或输出线路，并通过无符号整数0..maxpin表示。该数字空间在每个引脚控制器中都是本地的，因此在系统中可能有多个这样的数字空间。这个引脚空间可能是稀疏的——即在没有引脚存在的数字空间中可能存在间隙。

当一个PIN控制器实例被创建时，它会向PIN控制框架注册一个说明符，这个说明符包含了一组PIN说明符，说明了被这个特定PIN控制器处理的PIN。

以下是从底部看到的PGA（引脚网格阵列）芯片的示例：

A B C D E F G H

8 o o o o o o o o

7 o o o o o o o o

6 o o o o o o o o

5 o o o o o o o o

4 o o o o o o o o

3 o o o o o o o o

2 o o o o o o o o

1 o o o o o o o o

为了注册一个引脚控制器并命名这个封装中的所有引脚，我们可以在驱动程序中这样做:

#include <linux/pinctrl/pinctrl.h>

const struct pinctrl\_pin\_desc foo\_pins[] = {

PINCTRL\_PIN(0, "A8"),

PINCTRL\_PIN(1, "B8"),

PINCTRL\_PIN(2, "C8"),

...

PINCTRL\_PIN(61, "F1"),

PINCTRL\_PIN(62, "G1"),

PINCTRL\_PIN(63, "H1"),

};

static struct pinctrl\_desc foo\_desc = {

.name = "foo",

.pins = foo\_pins,

.npins = ARRAY\_SIZE(foo\_pins),

.owner = THIS\_MODULE,

};

int \_\_init foo\_probe(void)

{

int error;

struct pinctrl\_dev \*pctl;

error = pinctrl\_register\_and\_init(&foo\_desc, <PARENT>,

NULL, &pctl);

if (error)

return error;

return pinctrl\_enable(pctl);

}

要启用pinctrl子系统和PINMUX和PINCONF子组以及选定的驱动程序，您需要从机器的Kconfig条目中选择它们，因为它们与它们所使用的机器紧密集成。例如，参见arch/arm/mach-ux500/Kconfig作为一个例子。

引脚通常有比这个更复杂的名称。您可以在芯片的数据手册中找到这些。请注意，核心pinctrl.h文件提供了一个名为PINCTRL\_PIN()的高级宏，用于创建结构条目。正如您所看到的，我从左上角从0枚举了引脚到右下角的63。这个枚举是任意选择的，在实践中，您需要仔细考虑您的编号系统，以使它与在驱动程序中的寄存器布局和诸如此类的东西匹配，否则代码可能会变得复杂。您还必须考虑匹配偏移量到由PIN控制器处理的GPIO范围。

对于具有467个焊盘的焊盘环，与实际引脚不同，我使用了这样的枚举，顺着芯片的边缘走，这也似乎是行业标准（所有这些焊盘也有名称）:

0 ..... 104

466 105

. .

. .

358 224

357 .... 225

## 引脚组

许多控制器需要处理引脚组，因此引脚控制子系统具有一个枚举引脚组并检索属于某个特定组的实际枚举引脚的机制。

例如，假设我们有一个处理SPI接口的引脚组为{0，8，16，24}，一个处理引脚的I2C接口的引脚组为{24，25}。

这两个组通过实现一些通用的pinctrl\_ops向引脚控制子系统呈现，例如:

#include <linux/pinctrl/pinctrl.h>

struct foo\_group {

const char \*name;

const unsigned int \*pins;

const unsigned num\_pins;

};

static const unsigned int spi0\_pins[] = { 0, 8, 16, 24 };

static const unsigned int i2c0\_pins[] = { 24, 25 };

static const struct foo\_group foo\_groups[] = {

{

.name = "spi0\_grp",

.pins = spi0\_pins,

.num\_pins = ARRAY\_SIZE(spi0\_pins),

},

{

.name = "i2c0\_grp",

.pins = i2c0\_pins,

.num\_pins = ARRAY\_SIZE(i2c0\_pins),

},

};

static int foo\_get\_groups\_count(struct pinctrl\_dev \*pctldev)

{

return ARRAY\_SIZE(foo\_groups);

}

static const char \*foo\_get\_group\_name(struct pinctrl\_dev \*pctldev,

unsigned selector)

{

return foo\_groups[selector].name;

}

static int foo\_get\_group\_pins(struct pinctrl\_dev \*pctldev, unsigned selector,

const unsigned \*\*pins,

unsigned \*num\_pins)

{

\*pins = (unsigned \*) foo\_groups[selector].pins;

\*num\_pins = foo\_groups[selector].num\_pins;

return 0;

}

static struct pinctrl\_ops foo\_pctrl\_ops = {

.get\_groups\_count = foo\_get\_groups\_count,

.get\_group\_name = foo\_get\_group\_name,

.get\_group\_pins = foo\_get\_group\_pins,

};

static struct pinctrl\_desc foo\_desc = {

...

.pctlops = &foo\_pctrl\_ops,

};

PIN控制子系统将调用.get\_groups\_count()函数来确定合法选择器的总数，然后调用其他函数来检索组的名称和PIN。保持组的数据结构由驱动程序负责，这只是一个简单的例子，实际上您可能需要在组结构中有更多的条目，例如与每个组关联的特定寄存器范围等等。

## 引脚配置

引脚有时可以以各种方式进行软件配置，大多与它们作为输入或输出时的电子属性有关。例如，您可以使输出引脚高阻抗或“三态”，意思是它实际上被断开。您可以使用某个电阻值，如上拉和下拉，将输入引脚连接到VDD或GND，以使当没有任何驱动选到其所连接的轨道时，引脚具有稳定的值，或者当它未连接时。

引脚配置可通过将配置条目添加到映射表中来进行编程；请查看下面的“板/机器配置”部分。

配置参数PLATFORM\_X\_PULL\_UP的格式和含义完全由引脚控制器驱动程序定义。

引脚配置驱动程序实现了在引脚控制器操作中更改引脚配置的回调，如下所示:

#include <linux/pinctrl/pinctrl.h>

#include <linux/pinctrl/pinconf.h>

#include "platform\_x\_pindefs.h"

static int foo\_pin\_config\_get(struct pinctrl\_dev \*pctldev,

unsigned offset,

unsigned long \*config)

{

struct my\_conftype conf;

... Find setting for pin @ offset ...

\*config = (unsigned long) conf;

}

static int foo\_pin\_config\_set(struct pinctrl\_dev \*pctldev,

unsigned offset,

unsigned long config)

{

struct my\_conftype \*conf = (struct my\_conftype \*) config;

switch (conf) {

case PLATFORM\_X\_PULL\_UP:

...

}

}

}

static int foo\_pin\_config\_group\_get (struct pinctrl\_dev \*pctldev,

unsigned selector,

unsigned long \*config)

{

...

}

static int foo\_pin\_config\_group\_set (struct pinctrl\_dev \*pctldev,

unsigned selector, unsigned long config)

{

...

}

static struct pinconf\_ops foo\_pconf\_ops = {

.pin\_config\_get = foo\_pin\_config\_get,

.pin\_config\_set = foo\_pin\_config\_set,

.pin\_config\_group\_get = foo\_pin\_config\_group\_get,

.pin\_config\_group\_set = foo\_pin\_config\_group\_set,

};

/\* Pin config operations are handled by some pin controller \*/

static struct pinctrl\_desc foo\_desc = {

...

.confops = &foo\_pconf\_ops,

};

由于某些控制器具有用于处理整个引脚组的特殊逻辑，它们可以利用特殊的整组引脚控制函数。pin\_config\_group\_set（）回调允许返回错误代码 -EAGAIN，以用于它不想处理的组，或者如果它只想进行一些组级处理，然后跌落到迭代所有引脚，则每个单独的引脚都将通过单独的pin\_config\_set（）调用进行处理。

## 与GPIO子系统的交互

GPIO驱动程序可能希望对同时注册为引脚控制器引脚和GPIO的同一物理引脚执行各种类型的操作。

首先，两个子系统可以完全正交使用，请参阅下面的“驱动程序发出引脚控制请求”和“驱动程序需要引脚控制和GPIO”部分了解详细信息。但在某些情况下，需要在引脚和GPIO之间进行交叉子系统映射。

由于引脚控制器子系统具有其引脚空间局限于引脚控制器，因此我们需要一种映射，以便引脚控制子系统可以确定哪个引脚控制器处理某个GPIO引脚的控制。由于单个引脚控制器可以复用多个GPIO范围（通常是SoC，其具有一组引脚，但内部有多个GPIO硅块，每个模拟为struct gpio\_chip），因此可以像这样将任意数量的GPIO范围添加到引脚控制器实例中:

struct gpio\_chip chip\_a;

struct gpio\_chip chip\_b;

static struct pinctrl\_gpio\_range gpio\_range\_a = {

.name = "chip a",

.id = 0,

.base = 32,

.pin\_base = 32,

.npins = 16,

.gc = &chip\_a;

};

static struct pinctrl\_gpio\_range gpio\_range\_b = {

.name = "chip b",

.id = 0,

.base = 48,

.pin\_base = 64,

.npins = 8,

.gc = &chip\_b;

};

{

struct pinctrl\_dev \*pctl;

...

pinctrl\_add\_gpio\_range(pctl, &gpio\_range\_a);

pinctrl\_add\_gpio\_range(pctl, &gpio\_range\_b);

}

因此，此复杂系统有一个引脚控制器处理两个不同的GPIO芯片。“chip a”具有16个引脚，“chip b”具有8个引脚。“chip a”和“chip b”具有不同的.pin\_base，这意味着GPIO范围的起始引脚号。

“chip a”的GPIO范围从GPIO基数32开始，实际引脚范围也从32开始。然而，“chip b”具有不同的GPIO范围和引脚范围的起始偏移量。“chip b”的GPIO范围从GPIO号48开始，而“chip b”的引脚范围从64开始。

我们可以使用此“pin\_base”将gpio编号转换为实际引脚编号。它们在全局GPIO引脚空间中映射到：

芯片a：

GPIO范围：[32..47]

引脚范围：[32..47]

芯片b：

GPIO范围：[48..55]

引脚范围：[64..71]

上述示例假定GPIO和引脚之间的映射是线性的。如果映射是稀疏或杂乱无章的，则可以像这样在范围中编码任意引脚号的数组:

static const unsigned range\_pins[] = { 14, 1, 22, 17, 10, 8, 6, 2 };

static struct pinctrl\_gpio\_range gpio\_range = {

.name = "chip",

.id = 0,

.base = 32,

.pins = &range\_pins,

.npins = ARRAY\_SIZE(range\_pins),

.gc = &chip;

};

在这种情况下，将忽略pin\_base属性。如果已知引脚组的名称，可以使用函数pinctrl\_get\_group\_pins（）初始化上述结构的pins和npins元素，例如对于引脚组“foo”:

pinctrl\_get\_group\_pins(pctl, "foo", &gpio\_range.pins,

&gpio\_range.npins);

当在引脚控制子系统中调用与GPIO相关的功能时，将使用这些范围来通过检查和匹配跨所有控制器的引脚范围来查找适当的引脚控制器。当找到处理匹配范围的引脚控制器时，将在该特定引脚控制器上调用特定的GPIO特定函数。

对于处理引脚偏置、引脚复用等所有功能，引脚控制器子系统将查找传递的GPIO编号上的相应引脚号，并使用范围的内部来检索引脚号。之后，子系统将将其传递给引脚控制驱动程序，因此驱动程序将获得一个处于其处理的数字范围内的引脚号。此外，还传递了范围ID值，以便引脚控制器知道它应该处理哪个范围。

从pinctrl驱动程序调用pinctrl\_add\_gpio\_range是不推荐的。请参阅gpio.txt文档的第2.1节，了解如何绑定pinctrl和gpio驱动程序。

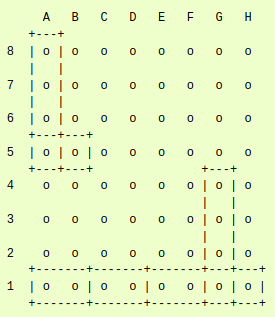
## PINMUX 接口

这些调用使用pinmux\_\*命名前缀。其他调用不应使用该前缀。

## 什么是pinmuxing？

PINMUX，也称为padmux、ballmux、alternate functions或mission modes，是芯片供应商使用某种电气封装的物理引脚（球、垫、指等）实现多种互斥功能的一种方式，具体取决于应用程序。在这个上下文中，“应用程序”通常是指将封装物连接到电子系统的一种方式，尽管该框架也可以在运行时更改功能。

以下是从下面看到的PGA（引脚网格阵列）芯片的示例:



这不是俄罗斯方块。您要考虑的游戏是国际象棋。并非所有PGA / BGA封装都类似于国际象棋棋盘，大型封装在某种排列方式下有“孔”，但我们使用这个作为一个简单的例子。您看到的引脚中，一些会被用于某些VCC和GND等东西，以供电给芯片，而相当多的引脚将被大型端口占用，例如外部存储器接口。其余的引脚通常会用于引脚复用。

上面的8x8 PGA封装将对其物理引脚分配0到63个引脚。它将使用pinctrl\_register\_pins()和适当的数据集命名引脚{A1、A2、A3 ... H6、H7、H8}，如前面所示。

在这个8x8 BGA封装中，引脚{A8，A7，A6，A5}可用作SPI端口（这是四个引脚：CLK、RXD、TXD、FRM）。在这种情况下，引脚B5可以用作一些通用GPIO引脚。但是，在另一种设置中，引脚{A5，B5}可以用作I2C端口（这只是两个引脚：SCL，SDA）。不用说，我们不能同时使用SPI端口和I2C端口。但是，在封装内部，执行SPI逻辑的硅可以在引脚{G4，G3，G2，G1}上路由。

在底行的{A1，B1，C1，D1，E1，F1，G1，H1}处有一些特殊的物品-它是一个外部MMC总线，可以是2、4或8位宽，并且将分别占用2、4或8个引脚，因此要么使用{A1，B1}，要么使用{A1，B1，C1，D1}或全部。如果我们使用了8位，则当然不能同时使用引脚{G4，G3，G2，G1}上的SPI端口。

这样，封装内部的硅块可以通过不同的引脚范围被多路复用“muxed”。通常，当代SoC（片上系统）将包含多个I2C、SPI、SDIO / MMC等硅块，可以通过引脚mux设置路由到不同的引脚。

由于通用输入/输出引脚（GPIO）通常总是短缺的，因此通常可以在某些其他I/O端口未使用时将几乎任何引脚用作GPIO引脚。

## Pinmux约定

引脚控制器子系统中引脚mux功能的目的是为您在机器配置中选择实例化的设备提供抽象和提供pinmux设置。它受到clk、GPIO和稳压器子系统的启发，因此设备将请求其mux设置，但也可以请求单个引脚作为GPIO。

定义：

1. 功能可以通过存储在内核的drivers / pinctrl / \*目录中的引脚控制子系统中的驱动程序切换。引脚控制驱动程序知道可能的功能。在上面的示例中，您可以识别出三个pinmux功能，一个用于spi，一个用于i2c，另一个用于mmc。
2. 功能被假定为可以从一个一维数组中枚举为零。在这种情况下，数组可以是：{spi0, i2c0, mmc0}，用于三个可用功能。
3. 功能具有在通用级别定义的PIN GROUPS - 因此，某个功能始终与某个引脚组集相关联，可能只是一个，但也可能是许多。在上述示例中，i2c函数与引脚{A5，B5}相关联，按{24，25}在控制器引脚空间中枚举。

函数spi与引脚组{A8，A7，A6，A5}和{G4，G3，G2，G1}相关联，分别枚举为{0，8，16，24}和{38，46，54，62}。

每个引脚控制器的组名必须是唯一的，同一控制器上的两个组不能具有相同的名称。

1. 一个功能和一个引脚组合确定了一组引脚的特定功能。函数和引脚组合的知识以及它们的机器特定细节都保存在引脚多路复用驱动程序中。对外只知道枚举器，驱动程序核心可以请求。
2. 使用特定选择器（>=0）的某个选定的功能的名称
3. 与某个功能相关联的组列表
4. 在该列表中启用某个组来执行某个功能

如上所述，引脚组再次自说明，因此核心将从驱动程序中检索某个组中实际引脚范围。

1. 某个引脚控制器上的功能和组通过板文件、设备树或类似的机器设置配置机制映射到一个设备上，类似于稳压器通常通过名称连接到设备。因此，定义引脚控制器、功能和组可以唯一地标识一组特定设备要使用的引脚。（如果仅有一个可用于该功能的引脚组，则不需要提供组名 - 核心将仅选择可用的第一个组。）

在本例中，我们可以定义这台机器将使用spi0设备，其中引脚多路复用功能为fspi0组为gspi0，i2c0对应的引脚多路复用功能为fi2c0组为gi2c0，在主引脚控制器上，我们得到如下映射:

{

{"map-spi0", spi0, pinctrl0, fspi0, gspi0},

{"map-i2c0", i2c0, pinctrl0, fi2c0, gi2c0}

}

每个映射必须指定状态名称、引脚控制器、设备和功能。组不是必需的 - 如果省略，则核心将选择驱动程序呈现为适用于该功能的第一个组，这对简单情况很有用。

可以将几个组映射到相同的设备、引脚控制器和功能组合。这适用于某个引脚控制器上的某个功能在不同配置中使用不同的引脚集的情况。

1. 对于某个引脚控制器上使用某个引脚组的某个功能的引脚，是基于先到先得的原则提供的。因此，如果某个其他设备的多路复用设置或GPIO引脚请求已经使用了您的物理引脚，您将被拒绝使用它。为了获得（激活）新设置，必须先放置（去激活）旧设置。

有时，文档和硬件寄存器将绕着垫子（或“导线”）而不是引脚进行定位 - 这些是包内硅片上的焊接表面，可能与胶囊下方的实际引脚/球的数量匹配或不匹配。选择一些对您有意义的枚举。如果有意义，仅为您可以控制的引脚定义枚举器。

假设：

我们假设可能的功能映射到引脚组的数量受硬件限制。也就是说，我们假设不存在任何函数可以映射到任何引脚的系统，就像电话交换机一样。因此，某个功能的可用引脚组将限制为几个选择（例如最多八个），而不是数百个或任何选择的数量。这是我们通过检查可用的引脚多路复用硬件找到的特征，也是必要的假设，因为我们希望引脚多路复用驱动程序向子系统呈现所有可能的功能与引脚组映射。

## 引脚多路复用驱动程序

引脚多路复用核心负责防止引脚冲突并调用引脚控制器驱动程序执行不同的设置。

引脚多路复用驱动程序的责任是强加进一步限制（例如推断由于负载等电子限制而导致的功能是否实际允许），以确定所请求的功能实际上是否可以允许，并在可以执行请求的多路复用设置时探测硬件。

引脚多路复用驱动程序需要提供几个回调函数，其中一些是可选的。通常实现set\_mux()函数，将值写入某些指定寄存器以激活某个引脚的某个mux设置。

上面示例的一个简单驱动程序将通过将位0、1、2、3或4设置为名称为MUX的某个寄存器来选择某个功能及其引脚组，大致如下所示:

#include <linux/pinctrl/pinctrl.h>

#include <linux/pinctrl/pinmux.h>

struct foo\_group {

const char \*name;

const unsigned int \*pins;

const unsigned num\_pins;

};

static const unsigned spi0\_0\_pins[] = { 0, 8, 16, 24 };

static const unsigned spi0\_1\_pins[] = { 38, 46, 54, 62 };

static const unsigned i2c0\_pins[] = { 24, 25 };

static const unsigned mmc0\_1\_pins[] = { 56, 57 };

static const unsigned mmc0\_2\_pins[] = { 58, 59 };

static const unsigned mmc0\_3\_pins[] = { 60, 61, 62, 63 };

static const struct foo\_group foo\_groups[] = {

{

.name = "spi0\_0\_grp",

.pins = spi0\_0\_pins,

.num\_pins = ARRAY\_SIZE(spi0\_0\_pins),

},

{

.name = "spi0\_1\_grp",

.pins = spi0\_1\_pins,

.num\_pins = ARRAY\_SIZE(spi0\_1\_pins),

},

{

.name = "i2c0\_grp",

.pins = i2c0\_pins,

.num\_pins = ARRAY\_SIZE(i2c0\_pins),

},

{

.name = "mmc0\_1\_grp",

.pins = mmc0\_1\_pins,

.num\_pins = ARRAY\_SIZE(mmc0\_1\_pins),

},

{

.name = "mmc0\_2\_grp",

.pins = mmc0\_2\_pins,

.num\_pins = ARRAY\_SIZE(mmc0\_2\_pins),

},

{

.name = "mmc0\_3\_grp",

.pins = mmc0\_3\_pins,

.num\_pins = ARRAY\_SIZE(mmc0\_3\_pins),

},

};

static int foo\_get\_groups\_count(struct pinctrl\_dev \*pctldev)

{

return ARRAY\_SIZE(foo\_groups);

}

static const char \*foo\_get\_group\_name(struct pinctrl\_dev \*pctldev,

unsigned selector)

{

return foo\_groups[selector].name;

}

static int foo\_get\_group\_pins(struct pinctrl\_dev \*pctldev, unsigned selector,

const unsigned \*\* pins,

unsigned \* num\_pins)

{

\*pins = (unsigned \*) foo\_groups[selector].pins;

\*num\_pins = foo\_groups[selector].num\_pins;

return 0;

}

static struct pinctrl\_ops foo\_pctrl\_ops = {

.get\_groups\_count = foo\_get\_groups\_count,

.get\_group\_name = foo\_get\_group\_name,

.get\_group\_pins = foo\_get\_group\_pins,

};

struct foo\_pmx\_func {

const char \*name;

const char \* const \*groups;

const unsigned num\_groups;

};

static const char \* const spi0\_groups[] = { "spi0\_0\_grp", "spi0\_1\_grp" };

static const char \* const i2c0\_groups[] = { "i2c0\_grp" };

static const char \* const mmc0\_groups[] = { "mmc0\_1\_grp", "mmc0\_2\_grp", "mmc0\_3\_grp" };

static const struct foo\_pmx\_func foo\_functions[] = {

{

.name = "spi0",

.groups = spi0\_groups,

.num\_groups = ARRAY\_SIZE(spi0\_groups),

},

{

.name = "i2c0",

.groups = i2c0\_groups,

.num\_groups = ARRAY\_SIZE(i2c0\_groups),

},

{

.name = "mmc0",

.groups = mmc0\_groups,

.num\_groups = ARRAY\_SIZE(mmc0\_groups),

},

};

static int foo\_get\_functions\_count(struct pinctrl\_dev \*pctldev)

{

return ARRAY\_SIZE(foo\_functions);

}

static const char \*foo\_get\_fname(struct pinctrl\_dev \*pctldev, unsigned selector)

{

return foo\_functions[selector].name;

}

static int foo\_get\_groups(struct pinctrl\_dev \*pctldev, unsigned selector,

const char \* const \*\*groups,

unsigned \* const num\_groups)

{

\*groups = foo\_functions[selector].groups;

\*num\_groups = foo\_functions[selector].num\_groups;

return 0;

}

static int foo\_set\_mux(struct pinctrl\_dev \*pctldev, unsigned selector,

unsigned group)

{

u8 regbit = (1 << selector + group);

writeb((readb(MUX)|regbit), MUX);

return 0;

}

static struct pinmux\_ops foo\_pmxops = {

.get\_functions\_count = foo\_get\_functions\_count,

.get\_function\_name = foo\_get\_fname,

.get\_function\_groups = foo\_get\_groups,

.set\_mux = foo\_set\_mux,

.strict = true,

};

/\* Pinmux operations are handled by some pin controller \*/

static struct pinctrl\_desc foo\_desc = {

...

.pctlops = &foo\_pctrl\_ops,

.pmxops = &foo\_pmxops,

};

在上述示例中，同时激活多路复用0和1，通过设置位0和1，会使用同一引脚，因此它们会发生冲突。

引脚复用子系统的优美之处在于，由于它跟踪所有引脚以及谁在使用它们，它已经拒绝了像这样的不可能请求，因此驱动程序不需要担心这些问题——当它获得传递的选择器时，引脚复用子系统会确保没有其他设备或GPIO分配已经使用了所选引脚。因此，在控制寄存器中设置0和1位将永远不会同时设置。

对于引脚复用驱动程序，所有上述函数都是必须要实现的。

## 引脚控制与GPIO子系统的交互

请注意，以下暗示使用从<linux/gpio.h>中的API在Linux内核中使用某个引脚的情况，使用gpio\_request()和类似的函数。虽然有一些情况可能会使用您的数据表称为“GPIO模式”的东西，但实际上这只是某些设备的电学配置。有关此场景的更多详细信息，请参见下面名为“GPIO模式陷阱”的部分。

公共的引脚复用API包含两个名为pinctrl\_gpio\_request()和pinctrl\_gpio\_free()的函数。这两个函数应该只在基于gpiolib的驱动程序中作为它们的gpio\_request()和gpio\_free()语义的一部分调用。同样，pinctrl\_gpio\_direction\_[input|output]函数只应该从respective gpio\_direction\_[input|output] gpiolib实现内部调用。

注意，平台和单独的驱动程序不应该请求控制GPIO引脚的多路复用。取而代之，实现一个适当的gpiolib驱动程序，并让该驱动程序请求其引脚的适当复用和其他控制。

函数列表可能变得很长，尤其是如果您可以将每个单独的引脚转换为独立于任何其他引脚的GPIO引脚，然后尝试将每个引脚定义为一个函数的方法。

在这种情况下，函数数组将成为每个GPIO设置和设备函数的64个条目。

因此，引脚控制驱动程序可以实现两个函数，以仅在个别引脚上启用GPIO：.gpio\_request\_enable()和.gpio\_disable\_free()。

此函数将通过引脚控制器核心识别出受影响的GPIO范围，因此您知道哪些GPIO引脚受请求操作影响。

如果您的驱动程序需要从框架中获得GPIO引脚的方向指示，您可以实现.gpio\_set\_direction()函数。如上所述，它将从gpiolib驱动程序中调用，并将受影响的GPIO范围、引脚偏移和所需方向传递给该函数。

与使用这些特殊功能相比，完全允许为每个GPIO引脚使用命名功能，pinctrl\_gpio\_request()将尝试获取名为“gpioN”的功能，在其中“N”是全局GPIO引脚编号，如果没有注册特殊的GPIO处理程序，则会尝试获取该功能。

## GPIO模式陷阱

由于硬件工程师使用的命名约定，“GPIO”被认为是不同于内核的东西，因此开发人员可能会因数据表中的引脚可能设置为“GPIO模式”而感到困惑。看起来，硬件工程师所说的“GPIO模式”不一定是内核接口<linux/gpio.h>中所暗示的用例：从内核代码中获取的引脚，然后或听取输入，或将其高低驱动以断言/解除某些外部线路。

硬件工程师认为，“GPIO模式”意味着能够软件控制几个针脚的电学特性，如果该针脚被固定到某个设备，你将无法控制它的特性。

针脚的GPIO部分及其与特定针脚控制器配置和复用逻辑的关系有多种构建方式。以下是两个例子:

(A)

pin config

logic regs

| +- SPI

Physical pins --- pad --- pinmux -+- I2C

| +- mmc

| +- GPIO

pin

multiplex

logic regs

如果针脚用于GPIO或其他用途，都可以配置针脚的电学特性。如果将GPIO复用到针脚，则可以使用“GPIO”寄存器将其拉高/拉低。或者，该针脚可以由某个外设控制，同时应用所需的针脚配置属性。因此，GPIO功能与使用该针脚的任何其他设备是正交的。

在此安排中，针脚控制器的GPIO部分的寄存器或GPIO硬件模块的寄存器很可能位于专门用于GPIO的不同内存范围内，而处理针脚配置和针脚多路复用的寄存器范围则被放置在不同的内存范围和数据表的不同部分中。

结构体引脚复用操作中的“严格”标志可用于检查和拒绝在此类型的硬件上同时从GPIO和针脚复用使用者获取对同一引脚的访问。pinctrl驱动程序应相应地设置此标志。

(B)

pin config

logic regs

| +- SPI

Physical pins --- pad --- pinmux -+- I2C

| | +- mmc

| |

GPIO pin

multiplex

logic regs

在此安排中，可以始终启用GPIO功能，以便例如，可以使用GPIO输入“监听”SPI/I2C/MMC信号。如果在GPIO块上执行错误操作，可能会干扰引脚上的流量，因为它从未真正断开连接。可能将GPIO、针脚配置和针脚复用寄存器放置在相同的内存范围和数据表的同一部分中，但不必如此。

在一些针脚控制器中，即使物理针脚的设计方式与(B)相同，GPIO功能仍然不能与外围设备功能同时启用。因此，“严格”标志应设置，否则GPIO和其他复用设备可能会同时激活。

从内核的角度看，这些都是硬件的不同方面，应该放入不同的子系统中：

1. 控制引脚的电学特性（例如偏置和驱动强度）的寄存器（或寄存器字段）应通过pinctrl子系统公开为“引脚配置”设置
2. 从各种其他硬件块（如I2C、MMC或GPIO）中复用信号到针脚的寄存器（或寄存器字段）应作为复用函数通过pinctrl子系统公开。
3. 控制GPIO功能（例如设置GPIO输出值、读取GPIO输入值或设置GPIO针脚方向）的寄存器（或寄存器字段）应通过GPIO子系统公开，如果它们还支持中断功能，还应通过irqchip抽象公开。

根据确切的硬件寄存器设计，GPIO子系统公开的某些功能可能需要调用pinctrl子系统以协调跨硬件模块之间的寄存器设置。特别是，这可能适用于具有独立的GPIO和针脚控制器硬件模块的硬件，例如GPIO方向由针脚控制器硬件模块中的寄存器而非GPIO硬件模块决定的情况。

引脚的电学特性，如偏置和驱动强度，在所有情况下都可能放置在某个针脚特定寄存器中，或者特别是在(B)中作为GPIO寄存器的一部分。这并不意味着这些特性必然与Linux内核所称的“GPIO”相关。

例如：针脚通常复用为UART TX线路。但是在系统休眠期间，我们需要将此针脚设置为“GPIO模式”并接地。

如果为此针脚建立1：1映射到GPIO子系统，则可能开始考虑需要设计复杂的东西，即针脚必须同时用于UART TX和GPIO，需要抓取针脚控制句柄并将其设置为特定状态，以启用UART TX复用，然后将其切换到GPIO模式并使用gpio\_direction\_output()将其拉低以便在休眠期间使用，然后在唤醒后再将其复用到UART TX，甚至可以包括gpio\_request/gpio\_free作为此周期的一部分。这一切变得非常复杂。

解决方案是不要认为数据表中称为“GPIO模式”的东西必须由<linux/gpio.h>接口处理。相反，将其视为某个引脚配置设置。在例如<linux/pinctrl/pinconf-generic.h>中查找文档，您会发现以下内容:

**PIN\_CONFIG\_OUTPUT:**

这将配置输出引脚，使用参数1表示高电平，使用参数0表示低电平。

因此，可以完全将引脚推入“GPIO模式”，并将该线路驱动到低电平，作为通常的引脚控制映射的一部分。因此，例如您的UART驱动程序可能如下所示:

#include <linux/pinctrl/consumer.h>

struct pinctrl \*pinctrl;

struct pinctrl\_state \*pins\_default;

struct pinctrl\_state \*pins\_sleep;

pins\_default = pinctrl\_lookup\_state(uap->pinctrl, PINCTRL\_STATE\_DEFAULT);

pins\_sleep = pinctrl\_lookup\_state(uap->pinctrl, PINCTRL\_STATE\_SLEEP);

/\* Normal mode \*/

retval = pinctrl\_select\_state(pinctrl, pins\_default);

/\* Sleep mode \*/

retval = pinctrl\_select\_state(pinctrl, pins\_sleep);

### 您的机器配置可能如下所示:

static unsigned long uart\_default\_mode[] = {

PIN\_CONF\_PACKED(PIN\_CONFIG\_DRIVE\_PUSH\_PULL, 0),

};

static unsigned long uart\_sleep\_mode[] = {

PIN\_CONF\_PACKED(PIN\_CONFIG\_OUTPUT, 0),

};

static struct pinctrl\_map pinmap[] \_\_initdata = {

PIN\_MAP\_MUX\_GROUP("uart",PINCTRL\_STATE\_DEFAULT,"pinctrl-foo",

"u0\_group", "u0"),

PIN\_MAP\_CONFIGS\_PIN("uart",PINCTRL\_STATE\_DEFAULT,"pinctrl-foo",

"UART\_TX\_PIN", uart\_default\_mode),

PIN\_MAP\_MUX\_GROUP("uart", PINCTRL\_STATE\_SLEEP, "pinctrl-foo",

"u0\_group", "gpio-mode"),

PIN\_MAP\_CONFIGS\_PIN("uart",PINCTRL\_STATE\_SLEEP, "pinctrl-foo",

"UART\_TX\_PIN", uart\_sleep\_mode),

};

foo\_init(void) {

pinctrl\_register\_mappings(pinmap, ARRAY\_SIZE(pinmap));

}

这里我们要控制的引脚位于“u0\_group”中，有一个名为“u0”的函数可以在此引脚组上启用，然后一切都是正常的UART业务。但是也有一个名为“gpio-mode”的函数，可以映射到相同的引脚上，以将它们移动到GPIO模式。

这将在没有与GPIO子系统的任何虚假交互的情况下产生期望的效果。当该设备进入睡眠状态时，它只是电路配置，这可能意味着将引脚设置为数据表中称为“GPIO模式”的东西，但这并不是重点：它仍由该UART设备用于控制属于该UART驱动程序的引脚，将它们置于UART所需的模式中。在Linux内核意义上的GPIO只是一些1位线路，是不同的用例。

如何操作寄存器以获得推拉和输出低配置，以及“u0”或“gpio-mode”组的多路复用映射到这些引脚的方式是驱动程序的问题。

一些数据表将更有帮助性，并将“GPIO模式”称为“低功耗模式”，而不是与GPIO有关的任何内容。从电气角度来说，这通常意味着相同的事情，但在后一种情况下，软件工程师通常会迅速确定这是一些特定的多路复用或配置，而不是与GPIO API有关的任何内容。

## 板/机器配置

板和机器定义了某个完整运行系统的组合方式，包括如何多路复用GPIO和设备，如何约束调节器以及时钟树的外观。当然，引脚复用设置也是其中的一部分。

机器的引脚控制器配置看起来与简单的调节器配置非常相似，因此对于上面的示例数组，我们要在第二个函数映射上启用i2c和spi:

#include <linux/pinctrl/machine.h>

static const struct pinctrl\_map mapping[] \_\_initconst = {

{

.dev\_name = "foo-spi.0",

.name = PINCTRL\_STATE\_DEFAULT,

.type = PIN\_MAP\_TYPE\_MUX\_GROUP,

.ctrl\_dev\_name = "pinctrl-foo",

.data.mux.function = "spi0",

},

{

.dev\_name = "foo-i2c.0",

.name = PINCTRL\_STATE\_DEFAULT,

.type = PIN\_MAP\_TYPE\_MUX\_GROUP,

.ctrl\_dev\_name = "pinctrl-foo",

.data.mux.function = "i2c0",

},

{

.dev\_name = "foo-mmc.0",

.name = PINCTRL\_STATE\_DEFAULT,

.type = PIN\_MAP\_TYPE\_MUX\_GROUP,

.ctrl\_dev\_name = "pinctrl-foo",

.data.mux.function = "mmc0",

},

};

这里的dev\_name与用于查找设备结构的唯一设备名称匹配（就像使用clockdev或调节器一样）。功能名称必须与处理此引脚范围的引脚复用驱动程序提供的功能匹配。

正如您可以看到的那样，我们可能在系统上有几个引脚控制器，因此需要指定哪一个包含我们希望映射的函数。

您通过以下方式将此引脚映射注册到引脚复用子系统:

ret = pinctrl\_register\_mappings(mapping, ARRAY\_SIZE(mapping));

由于上述结构非常常见，因此还有一个辅助宏，可以使其更加紧凑，该宏假定您要使用pinctrl-foo和位置0进行映射，例如:

static struct pinctrl\_map mapping[] \_\_initdata = {

PIN\_MAP\_MUX\_GROUP("foo-i2c.o", PINCTRL\_STATE\_DEFAULT,

"pinctrl-foo", NULL, "i2c0"),

};

映射表还可以包含引脚配置条目。每个引脚/组通常都有一些影响它的配置条目，因此，对于引用此项的配置表条目，配置表条目通常会引用一个配置参数和值的数组。以下是使用方便宏的示例:

static unsigned long i2c\_grp\_configs[] = {

FOO\_PIN\_DRIVEN,

FOO\_PIN\_PULLUP,

};

static unsigned long i2c\_pin\_configs[] = {

FOO\_OPEN\_COLLECTOR,

FOO\_SLEW\_RATE\_SLOW,

};

static struct pinctrl\_map mapping[] \_\_initdata = {

PIN\_MAP\_MUX\_GROUP("foo-i2c.0", PINCTRL\_STATE\_DEFAULT,

"pinctrl-foo", "i2c0", "i2c0"),

PIN\_MAP\_CONFIGS\_GROUP("foo-i2c.0",PINCTRL\_STATE\_DEFAULT,

"pinctrl-foo", "i2c0", i2c\_grp\_configs),

PIN\_MAP\_CONFIGS\_PIN("foo-i2c.0", PINCTRL\_STATE\_DEFAULT,

"pinctrl-foo", "i2c0scl", i2c\_pin\_configs),

PIN\_MAP\_CONFIGS\_PIN("foo-i2c.0", PINCTRL\_STATE\_DEFAULT,

"pinctrl-foo", "i2c0sda", i2c\_pin\_configs),

};

最后，有些设备希望映射表包含某些特定的命名状态。当在不需要任何引脚控制器配置的硬件上运行时，映射表仍必须包含这些命名状态，以明确指示提供了状态并打算为空。表项宏PIN\_MAP\_DUMMY\_STATE用于定义命名状态而不会导致任何引脚控制器被编程。

static struct pinctrl\_map mapping[] \_\_initdata = {

PIN\_MAP\_DUMMY\_STATE("foo-i2c.0",PINCTRL\_STATE\_DEFAULT),

};

## 复杂映射

由于可以将功能映射到不同的引脚组，所以可以指定可选的.group，如下所示:

...

{

.dev\_name = "foo-spi.0",

.name = "spi0-pos-A",

.type = PIN\_MAP\_TYPE\_MUX\_GROUP,

.ctrl\_dev\_name = "pinctrl-foo",

.function = "spi0",

.group = "spi0\_0\_grp",

},

{

.dev\_name = "foo-spi.0",

.name = "spi0-pos-B",

.type = PIN\_MAP\_TYPE\_MUX\_GROUP,

.ctrl\_dev\_name = "pinctrl-foo",

.function = "spi0",

.group = "spi0\_1\_grp",

},

...

这个示例映射用于在运行时在两个spi0位置之间切换，如下面的“运行时pinmuxing”标题所述。

进一步，可以使一个命名状态影响多个引脚组的复用，例如在上面的mmc0示例中，您可以将mmc0总线从2扩展到4，再扩展到8个引脚。如果要为总共2 + 2 + 4 = 8个引脚（与8位MMC总线一样）使用所有三个组，我们像这样定义映射:

...

{

.dev\_name = "foo-mmc.0",

.name = "2bit"

.type = PIN\_MAP\_TYPE\_MUX\_GROUP,

.ctrl\_dev\_name = "pinctrl-foo",

.function = "mmc0",

.group = "mmc0\_1\_grp",

},

{

.dev\_name = "foo-mmc.0",

.name = "4bit"

.type = PIN\_MAP\_TYPE\_MUX\_GROUP,

.ctrl\_dev\_name = "pinctrl-foo",

.function = "mmc0",

.group = "mmc0\_1\_grp",

},

{

.dev\_name = "foo-mmc.0",

.name = "4bit"

.type = PIN\_MAP\_TYPE\_MUX\_GROUP,

.ctrl\_dev\_name = "pinctrl-foo",

.function = "mmc0",

.group = "mmc0\_2\_grp",

},

{

.dev\_name = "foo-mmc.0",

.name = "8bit"

.type = PIN\_MAP\_TYPE\_MUX\_GROUP,

.ctrl\_dev\_name = "pinctrl-foo",

.function = "mmc0",

.group = "mmc0\_1\_grp",

},

{

.dev\_name = "foo-mmc.0",

.name = "8bit"

.type = PIN\_MAP\_TYPE\_MUX\_GROUP,

.ctrl\_dev\_name = "pinctrl-foo",

.function = "mmc0",

.group = "mmc0\_2\_grp",

},

{

.dev\_name = "foo-mmc.0",

.name = "8bit"

.type = PIN\_MAP\_TYPE\_MUX\_GROUP,

.ctrl\_dev\_name = "pinctrl-foo",

.function = "mmc0",

.group = "mmc0\_3\_grp",

},

...

从设备中获取此映射的结果类似于使用以下命令:

p = devm\_pinctrl\_get(dev);

s = pinctrl\_lookup\_state(p, "8bit");

ret = pinctrl\_select\_state(p, s);

或更简单地:

p = devm\_pinctrl\_get\_select(dev, "8bit");

将同时激活映射中的所有三个底部记录的结果。由于它们共享相同的名称、引脚控制器设备、功能和设备，并且我们允许多个组与单个设备匹配，它们都被选择，并且由pinmux核心同时启用和禁用。

## 驱动程序的引脚控制请求

当设备驱动程序要探测设备时，设备核心会自动尝试在这些设备上发出pinctrl\_get\_select\_default()。这样，驱动程序编写者就不需要添加下面找到的任何样板代码。但是，如果进行精细的状态选择并且不使用“默认”状态，则可能需要对pinctrl句柄和状态进行一些设备驱动程序处理。

因此，如果您只想将某个设备的引脚放入默认状态并完成操作，则除提供正确的映射表外，无需进行任何操作。设备核心将处理其余部分。

通常不建议让各个驱动程序获取和启用引脚控制。因此，如果可能，请在平台代码或其他可以访问所有受影响的struct device \*指针的地方处理引脚控制。在某些情况下，如果驱动程序需要在运行时切换不同的复用映射，这是不可能的。

例如，如果驱动程序需要在正常操作和进入睡眠状态之间切换引脚的偏置，从PINCTRL\_STATE\_DEFAULT移动到PINCTRL\_STATE\_SLEEP，则可能会出现这种情况，甚至重新偏置或重新复用引脚以在睡眠模式下节省电流。

驱动程序可以请求激活特定的控制状态，通常仅是默认状态，如下所示:

#include <linux/pinctrl/consumer.h>

struct foo\_state {

struct pinctrl \*p;

struct pinctrl\_state \*s;

...

};

foo\_probe()

{

/\* Allocate a state holder named "foo" etc \*/

struct foo\_state \*foo = ...;

foo->p = devm\_pinctrl\_get(&device);

if (IS\_ERR(foo->p)) {

/\* FIXME: clean up "foo" here \*/

return PTR\_ERR(foo->p);

}

foo->s = pinctrl\_lookup\_state(foo->p, PINCTRL\_STATE\_DEFAULT);

if (IS\_ERR(foo->s)) {

/\* FIXME: clean up "foo" here \*/

return PTR\_ERR(foo->s);

}

ret = pinctrl\_select\_state(foo->s);

if (ret < 0) {

/\* FIXME: clean up "foo" here \*/

return ret;

}

}

如果您不希望每个驱动程序都处理它，而且您知道总线上的安排，则同样可以由总线驱动程序处理此get / lookup / select / put序列。

pinctrl API的语义如下：

1. 在进程上下文中调用pinctrl\_get()，以获取给定客户端设备的所有pinctrl信息的句柄。它将从内核内存中分配一个结构体来保存pinmux状态。所有映射表解析或类似的慢操作都在此API中进行。
2. devm\_pinctrl\_get()是pinctrl\_get()的变体，它会在关联设备被移除时自动调用pinctrl\_put()释放检索到的指针。建议使用此函数而不是普通的pinctrl\_get()。
3. 在进程上下文中调用pinctrl\_lookup\_state()，以获取客户端设备的特定状态的句柄。该操作可能也很慢。
4. pinctrl\_select\_state()根据映射表中给出的状态定义，对于引脚控制器硬件进行编程。理论上，这是一种快速通道操作，因为它只涉及向硬件中一些寄存器写入设置。但是，注意有些引脚控制器的寄存器可能位于一个慢/基于IRQ的总线上，因此客户端设备不应假设它们可以从非阻塞上下文调用pinctrl\_select\_state()。
5. pinctrl\_put()释放与pinctrl句柄关联的所有信息。
6. devm\_pinctrl\_put()是pinctrl\_put()的变体，可用于显式销毁devm\_pinctrl\_get()返回的pinctrl对象。但是，由于即使不调用此函数也可以自动清理，因此很少使用此函数。

pinctrl\_get()必须与普通的pinctrl\_put()配对使用。pinctrl\_get()不能与devm\_pinctrl\_put()配对使用。devm\_pinctrl\_get()可以选择与devm\_pinctrl\_put()配对使用。devm\_pinctrl\_get()不能与普通的pinctrl\_put()配对使用。

通常，引脚控件核心负责处理get/put对并调用设备驱动程序的簿记操作，例如检查可用功能和相关联的引脚，而select\_state将其传递给引脚控制器驱动程序，后者通过快速 poking 几个寄存器来负责激活/停用mux设置。

当您发出devm\_pinctrl\_get()调用时，会为您的设备分配引脚，之后您应该能够在所有引脚的debugfs列表中看到这一点。

注意：如果pinctrl系统找不到请求的pinctrl句柄，例如pinctrl驱动程序尚未注册，则pinctrl系统将返回-EPROBE\_DEFER。因此，请确保您的驱动程序的错误路径正常清理并准备在启动过程中稍后重试探测。

## 需要pin控制和GPIO的驱动程序

同样，不推荐让驱动程序自行查找和选择pin控制状态，但有时这是不可避免的。

所以假设您的驱动程序是这样获取其资源的:

#include <linux/pinctrl/consumer.h>

#include <linux/gpio.h>

struct pinctrl \*pinctrl;

int gpio;

pinctrl = devm\_pinctrl\_get\_select\_default(&dev);

gpio = devm\_gpio\_request(&dev, 14, "foo");

在这里，我们首先请求某个引脚状态，然后请求使用GPIO 14。如果您使用子系统像这样正交使用，您通常应该在请求GPIO之前获取您的pinctrl句柄并选择所需的pinctrl状态。这是一个语义约定，以避免可能会产生电气问题的情况，您肯定希望在GPIO子系统开始处理它们之前以某种方式对引脚进行mux和偏置。

上述可以隐藏起来：使用设备核心，pinctrl核心可以在设备探测之前正确设置端口配置和muxing，然而与GPIO子系统正交。

但是，在某些情况下，让GPIO子系统直接与pinctrl子系统通信并使用后者作为后端是有意义的。这是当GPIO驱动程序可能调用上面“引脚控制与GPIO子系统交互”部分中说明的函数时。这仅涉及每个引脚的复用，完全隐藏在gpio\_\*()函数命名空间之后。在这种情况下，驱动程序无需与引脚控制子系统交互。

如果引脚控制驱动程序和GPIO驱动程序正在处理相同的引脚且使用案例涉及复用，则必须将引脚控制器实现为GPIO驱动程序的后端，如下所示，除非硬件设计允许GPIO控制器通过硬件覆盖引脚控制器的复用状态，而无需与引脚控制系统交互。

## 系统引脚控制占用

当注册引脚控制器时，核可以占用pin控制映射表条目。这意味着在注册引脚控制设备后，核将尝试立即在其中调用pinctrl\_get()，lookup\_state()和select\_state()。

这发生在客户端设备名称等于引脚控制器设备名称，以及状态名称为PINCTRL\_STATE\_DEFAULT的映射表条目中：

{

.dev\_name = "pinctrl-foo",

.name = PINCTRL\_STATE\_DEFAULT,

.type = PIN\_MAP\_TYPE\_MUX\_GROUP,

.ctrl\_dev\_name = "pinctrl-foo",

.function = "power\_func",

},

由于请求核心在主要引脚控制器上独占一些始终适用的mux设置可能很常见，因此有一个方便的宏来实现:

PIN\_MAP\_MUX\_GROUP\_HOG\_DEFAULT("pinctrl-foo", NULL /\* group \*/,

"power\_func")

这与上面的构造完全相同。

## 运行时 pinmuxing

在运行时，可以将某个函数从一个 set 的引脚移到另一个 set 的引脚中进行 mux，例如将 SPI 端口从一组引脚移动到另一组引脚。例如，在上面的 spi0 示例中，我们为相同的功能公开了两个不同的引脚组，但在上面“高级映射”中不同的命名。因此，对于一个 SPI 设备，我们有两个状态，命名为“pos-A”和“pos-B”。

此片段首先初始化两组的状态对象（在 foo\_probe() 中），然后在由组 A 定义的引脚上进行函数 mux，最后在由组B 定义的引脚上进行 mux:

#include <linux/pinctrl/consumer.h>

struct pinctrl \*p;

struct pinctrl\_state \*s1, \*s2;

foo\_probe()

{

/\* Setup \*/

p = devm\_pinctrl\_get(&device);

if (IS\_ERR(p))

...

s1 = pinctrl\_lookup\_state(foo->p, "pos-A");

if (IS\_ERR(s1))

...

s2 = pinctrl\_lookup\_state(foo->p, "pos-B");

if (IS\_ERR(s2))

...

}

foo\_switch()

{

/\* Enable on position A \*/

ret = pinctrl\_select\_state(s1);

if (ret < 0)

...

...

/\* Enable on position B \*/

ret = pinctrl\_select\_state(s2);

if (ret < 0)

...

...

}

以上必须从进程上下文中进行。当状态被激活时，将保留引脚，因此在运行中的系统上可以使用不同的函数在不同的时间使用一个特定的引脚。

# 通用输入/输出(GPIO)

## 介绍

### GPIO 接口

此目录中的文件提供了有关如何在驱动程序中访问 GPIO 以及如何编写提供 GPIO 的设备的驱动程序的详细说明。

由于内核中 GPIO 接口的历史，有两种不同的方法可以获取和使用 GPIO：

1. 基于描述符的接口是操作 GPIO 的首选方式，并且由该目录中除 gpio-legacy.txt 之外的所有文件描述。
2. gpio-legacy.txt 中记录了被认为已弃用（但出于兼容性原因仍可用）的旧式基于整数的接口。

本文档的其余部分适用于新的基于描述符的接口。gpio-legacy.txt 包含应用于旧式整数接口的相同信息。

### 什么是 GPIO?

“通用输入/输出”(GPIO) 是一种灵活的软件可控的数字信号。它们由许多种芯片提供，对于使用嵌入式和定制硬件的 Linux 开发人员而言非常熟悉。每个 GPIO 表示连接到特定引脚或球阵列上的位。板子电路图显示外部硬件连接到哪些 GPIO。驱动程序可以编写通用，以便板子设置代码向驱动程序传递这样的引脚配置数据。

系统级芯片 (SOC) 处理器在很大程度上依赖 GPIO。在某些情况下，每个非专用引脚都可以配置为 GPIO；大多数芯片至少有几十个 GPIO。可编程逻辑设备 (如 FPGAs) 可以轻松地提供 GPIO；像电源管理器和音频编解码器这样的多功能芯片通常有几个这样的引脚以帮助在 SOC 上处理引脚短缺；还有连接使用 I2C 或 SPI 序列总线的“GPIO 扩展器”芯片。大多数 PC 南桥都有几十个 GPIO 可编程引脚 (只有 BIOS 固件知道它们如何使用)。

GPIO 的确切功能因系统而异。常见选项：

1. 输出值可写入 (高=1，低=0)。有些芯片也有关于如何驱动该值的选项，以便仅驱动一个值，支持“线或”和其他值的类似方案 (尤其是“开漏”信号)。
2. 输入值同样可读取 (1、0)。有些芯片支持配置为“输出”的引脚的读回，这在这种 “线或”情况下非常有用 (以支持双向信号)。GPIO 控制器可能具有输入去抖 / 去斑逻辑，有时支持软件控制。
3. 输入通常可以用作 IRQ 信号，通常是边缘触发的，但有时是电平触发的。这些 IRQ 可以配置为系统唤醒事件，以将系统从低功耗状态唤醒。
4. 通常一个 GPIO 可以根据不同的产品板需要配置为输入或输出；也存在单向 GPIO。
5. 大多数 GPIO 可以在保持自旋锁的情况下访问，但通过串行总线访问的 GPIO 通常无法访问。一些系统支持两种类型。

在给定的板子上，每个 GPIO 都用于一个特定的目的，例如监视 MMC / SD 卡插入/拔出、检测卡片写保护状态、驱动 LED、配置收发器、控制串行总线、触发硬件看门狗、检测开关等等。

### 常见 GPIO 属性

这些属性在 GPIO 接口的其他文件中都存在，了解它们非常有用，特别是如果您需要定义 GPIO 映射。

#### 高电平有效和低电平有效

当 GPIO 的输出信号为 1 ("高电平") 时，人们很自然地会认为它是 "有效" 的，并且当它为 0 ("低电平") 时为非活动状态。然而，在实践中，GPIO 的信号可能会在到达目的地之前被反转，或者设备可能会决定有关 "活动" 的含义有不同的约定。这些决策应该对设备驱动程序透明，因此可以将 GPIO 定义为活动高电平 ("1" 表示 "有效"，默认值) 或活动低电平 ("0" 表示 "有效")，这样驱动程序只需要关注逻辑信号，而不需要关心线路电平上发生了什么。

#### 开漏和开源

有时，共享信号需要使用 "开漏"(只有低电平信号实际被驱动) 或者 "开源"(只有高电平信号被驱动) 信号传输。该术语适用于 CMOS 晶体管；"开集电极" 用于 TTL。拉上或拉下电阻引起高或低信号电平。这有时被称为 "电线与"；或者更实际地，从负逻辑(低 = 真)的角度来看，这是一个 "电线或"。

一个常见的开漏信号例子是共享的活动低 IRQ 线。此外，双向数据总线信号有时使用开漏信号。

一些 GPIO 控制器直接支持开漏和开源输出；许多不支持。当您需要开漏信号传输，但您的硬件不直接支持它时，您可以使用常见的用法来模拟任何可用作输入或输出的 GPIO 引脚：

低：gpiod\_direction\_output(gpio, 0) ……这会驱动信号并覆盖上拉电阻。

高：gpiod\_direction\_input(gpio) ……这将关闭输出，所以上拉电阻(或其他某个设备)控制信号。

相同的逻辑可以应用于通过驱动高电平信号并将 GPIO 配置为低电平输入来模拟开源信号传输。这种开漏/开源模拟可以由 GPIO 框架透明地处理。

如果您正在 "驱动" 信号高，但 gpiod\_get\_value(gpio) 报告低值(经过适当的上升时间后)，则您知道其他组件正在将共享信号驱动到低值。这不一定是错误。一个常见的例子就是 I2C 时钟的拉伸：需要较慢时钟的从设备会延迟 SCK 的上升沿，并且 I2C 主设备会相应地调整其信号传输速率。

## GPIO 描述符驱动接口

本文档可作为 GPIO 芯片驱动编写者的指南。请注意，它描述了新的基于描述符的接口。有关已弃用的基于整数的 GPIO 接口的说明，请参阅 gpio-legacy.txt。

每个 GPIO 控制器驱动程序都需要包含以下头文件，该头文件定义用于定义 GPIO 驱动程序的结构:

#include <linux/gpio/driver.h>

### GPIO 的内部表示

在 GPIO 驱动程序内部，通过其硬件编号识别各个 GPIO，该编号是介于 0 和 n 之间的唯一数字，n 是由芯片管理的 GPIO 数量。该数字是纯粹的内部数字：特定 GPIO 描述符的硬件编号从未在驱动程序之外显示。

在此内部编号的基础上，每个 GPIO 线路还需要在整数 GPIO 命名空间中具有全局编号，以便与旧版 GPIO 接口一起使用。因此，每个芯片必须具有 "基本" 数字(可以自动分配)，每个 GPIO 线路的全局数字将为 (base + 硬件编号)。尽管整数表示已被视为弃用，但仍然拥有许多用户，因此需要维护。

例如，一个平台可以使用全局编号 32-159 来定义 GPIO，控制器在 32 的基础上定义了 128 个 GPIO，而另一个平台使用全局编号 0..63，其中一个 GPIO 控制器使用 64-79，而在一个特定的板上使用 80-95。遗留数字不需要连续；这两个平台都可以使用编号 2000-2063 来标识 I2C GPIO 扩展器的一个银行中的 GPIO 线路。

### 控制器驱动程序: gpio\_chip

在 gpiolib 框架中，每个 GPIO 控制器都打包为 "struct gpio\_chip" (请参见 <linux/gpio/driver.h> 完整定义)，其中的成员对于该类型的每个控制器都是通用的，这些应由驱动程序代码分配：

1. 用于确立 GPIO 线路方向的方法
2. 用于访问 GPIO 线路值的方法
3. 指定给定 GPIO 线路的电气配置的方法
4. 返回与给定 GPIO 线路相关联的 IRQ 编号的方法
5. 标志指示其方法的调用是否可能休眠
6. 可选的线路名称数组以识别线路
7. 可选的 debugfs 转储方法（显示额外的状态信息）
8. 可选的基础编号（如果省略，则将自动分配）
9. 用于诊断和使用平台数据进行 GPIO 芯片映射的标签

实现 gpio\_chip 的代码应支持控制器的多个实例，最好使用驱动程序模型。该代码将配置每个 gpio\_chip 并发出 gpiochip\_add()、gpiochip\_add\_data() 或 devm\_gpiochip\_add\_data()。移除 GPIO 控制器应该很少发生；当不可避免时，请使用 gpiochip\_remove()。

通常，gpio\_chip 是实例特定结构的一部分，具有 GPIO 接口未暴露的状态，例如寻址、电源管理等等。诸如音频编解码器之类的芯片将具有复杂的非 GPIO 状态。

任何 debugfs 转储方法通常应忽略未被请求的行。它们可以使用 gpiochip\_is\_requested()，该函数在请求 GPIO 行时返回 NULL 或与该行关联的标签。

RT\_FULL：如果预计要从 -RT 中的原子上下文调用 GPIO API（在硬中断处理程序和类似情况下内部调用），GPIO 驱动程序不应在其 gpio\_chip 实现（.get/.set 和方向控制回调）中使用 spinlock\_t 或任何可休眠的 API（如 PM runtime）。通常情况下，这不是必需的。

#### GPIO 电气配置

可以通过使用 .set\_config() 回调来将 GPIO 配置为几种电气工作模式。当前此 API 支持设置去抖动和单端模式（开漏/开源）。下面描述了这些设置。

.set\_config() 回调使用与通用 pin 控制驱动程序相同的枚举值和配置语义。这不是巧合：可以将 .set\_config() 分配给函数 gpiochip\_generic\_config()，这将导致调用 pinctrl\_gpio\_set\_config() 并最终结束于 GPIO 控制器“后面”的引脚控制后端，通常更靠近实际的引脚。这样引脚控制器就可以管理下面列出的 GPIO 配置。

如果使用引脚控制器后端，则 GPIO 控制器或硬件描述需要提供“GPIO 范围”，将 GPIO 行偏移量映射到引脚控制器上的引脚号，以便它们可以彼此正确交叉参照。

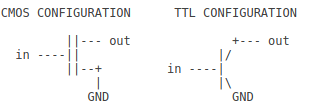
#### 具有去抖动支持的 GPIO

去抖动是设置为指示将其连接到机械开关或按钮或类似物的引脚。弹跳意味着出于机械原因，该线路被快速拉高/低，很短的时间间隔。这可能导致值不稳定或 irqs 反复触发，除非去抖动该线路。

实际上，去抖动涉及在线路上发生某些事情时设置定时器，等待一小段时间，然后再次采样该行，以查看其是否仍具有相同的值（低或高）。这也可以通过聪明的状态机重复，等待一条线变得稳定。无论哪种情况，都会设置一定数量的毫秒进行去抖动，或者仅为“打开/关闭”如果该时间不可配置。

#### 具有开漏/开源支持的 GPIO

开漏（CMOS）或开集合（TTL）意味着该线路没有主动驱动高电平：相反，您提供漏/集合作为输出，因此当晶体管未打开时，它将呈现高阻抗（三态）到外部总线:

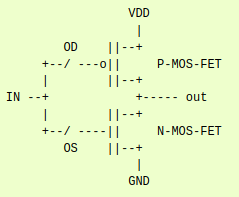


1. 电平转换：为了达到比输出所在的硅更高的逻辑电平。
2. 反向电线“或”在 I/O 线上，例如 GPIO 线，使得可以驱动该线的任何驱动阶段在其他输出同时驱动该线高时仍然将其驱动低。这是 I2C 总线的特例，它符合定义是一条电线“或”总线。

这两种用例都要求该线路配备上拉电阻。该电阻将使该线路倾向于高电平，除非总线上的任何晶体管主动将其下拉。

该线路上的电平将达到上拉电阻器上的 VDD 的高度，这可能高于晶体管支持的电平，从而实现到更高 VDD 的电平转换。

集成电子设备通常具有 CMOS “跷跷板”输出驱动器级别的输出驱动级别，其中一个 N-MOS 和一个 P-MOS 晶体管，其中一个驱动线路高，另一个驱动线路低。这称为推挽输出。如下所示:



期望的输出信号（例如直接来自某些 GPIO 输出寄存器）到达 IN。名为“OD”和“OS”的开关通常关闭，创建一个推挽电路。

请考虑名为“OD”和“OS”的小“开关”，它们可以在输入分裂后启用/禁用P-MOS或N-MOS晶体管。如您所见，如果此开关打开，则任何晶体管都将完全失灵。此时，totem极性被减半且提供高阻抗，而不是主动地将线路带高或低。这通常是软件控制的开漏/开源的工作方式。

一些GPIO硬件以开漏/开源配置形式出现。有些是硬连线，无论如何都只支持开漏或开源：只有一个晶体管。一些是软件配置的：通过在寄存器中翻转一个位，可以将输出配置为开漏或开源，实际上是通过翻转图上标有“OD”和“OS”的开关来实现的。

通过禁用P-MOS晶体管，可以在GND和高阻抗之间（开漏）驱动输出，通过禁用N-MOS晶体管，可以在VDD和高阻抗之间（开源）驱动输出。在第一种情况下，需要在流出的导轨上使用上拉电阻器来完成电路，而在第二种情况下，需要在导轨上使用下拉电阻器。

支持开漏或开源或两者的硬件可以在gpio\_chip中实现特殊回调：.set\_config（），其中包含通用的pinconf打包值，告知是否将行配置为开漏、开源或推挽。这将发生在来自机器文件或其他硬件描述的GPIO\_OPEN\_DRAIN或GPIO\_OPEN\_SOURCE标志设置的响应中。

如果无法在硬件中配置此状态，即如果GPIO硬件不支持硬件开漏/开源，则GPIO库将使用一个技巧：当将线路设置为输出时，如果将该行标记为开漏并且IN输出值为低，则它将像通常一样被驱动为低电平。但是，如果将IN输出值设置为高，则它将不会被驱动高，而是会被切换到输入，因为输入模式是高阻抗，因此实现了某种“开放漏模拟”：电性行为将是相同的，除了在切换线路模式时可能会出现硬件故障。

对于开源配置，使用相同的原理，只是不是主动地将线路驱动到低电平，而是将其设置为输入。

### 提供IRQ的GPIO驱动程序

通常，GPIO驱动程序（GPIO芯片）也提供中断，通常是从父中断控制器级联，有些特殊情况下，GPIO逻辑与SoC的主要中断控制器结合使用。

GPIO块的IRQ部分使用irq\_chip实现，使用头文件<linux/irq.h>。因此，此组合驱动程序同时使用两个子系统：gpio和irq。

RT\_FULL：实时兼容的GPIO驱动程序不应将spinlock\_t或任何可睡眠的API（例如PM运行时）用作其irqchip实现的一部分。

1. spinlock\_t应该替换为raw\_spinlock\_t[1]。
2. 如果必须使用可睡眠的API，则可以从.irq\_bus\_lock（）和.irq\_bus\_unlock（）回调中执行这些操作，因为这些是irqchip上唯一的缓慢路径回调。如果需要，请创建回调[2]。

GPIO irqchips通常分为两类：

1. 链接的GPIO IRQCHIPS：这通常是嵌入在SoC上的类型。这意味着在GPIO的快速IRQ流处理器中有一个被父IRQ处理程序级联调用的IRQ流处理器，最常见的是系统中断控制器。这意味着GPIO irqchip处理程序将立即从父irqchip调用，同时保持IRQ禁用。然后，GPIO irqchip将在其中断处理程序中调用诸如以下序列之类的东西：

static irqreturn\_t foo\_gpio\_irq(int irq, void \*data)

chained\_irq\_enter(...);

generic\_handle\_irq(...);

chained\_irq\_exit(...);

链接的GPIO irqchips通常无法在struct gpio\_chip上设置.can\_sleep标志，因为所有内容都直接在回调中发生：不能使用I2C之类的缓慢总线流量。

RT\_FULL：请注意，链接的IRQ处理程序不会在-RT上被强制线程化。因此，链接的IRQ处理程序中不能使用spinlock\_t或任何可睡眠的API（例如PM运行时）。如果需要（如果无法转换为嵌套线程GPIO irqchip，请参见下文），则链接的IRQ处理程序可以转换为通用irq处理程序，从而在-RT上成为线程化IRQ处理程序，在非RT上成为硬IRQ处理程序（例如，请参见[3])。Know W/A：期望使用IRQ禁用的generic\_handle\_irq（）被调用，因此如果它从被迫线程的IRQ处理程序中调用，IRQ核心将出现问题。可以使用“假？”原始锁来解决这个问题:

raw\_spinlock\_t wa\_lock;

static irqreturn\_t omap\_gpio\_irq\_handler(int irq, void \*gpiobank)

unsigned long wa\_lock\_flags;

raw\_spin\_lock\_irqsave(&bank->wa\_lock, wa\_lock\_flags);

generic\_handle\_irq(irq\_find\_mapping(bank->chip.irq.domain, bit));

raw\_spin\_unlock\_irqrestore(&bank->wa\_lock, wa\_lock\_flags);

1. 通用链接GPIO IRQCHIPS：它们与“链接GPIO irqchips”相同，但不使用链接的IRQ处理程序。代之以使用通用IRQ处理程序进行GPIO IRQ分发，该处理程序使用request\_irq()进行配置。然后，GPIO irqchip将在其中断处理程序中调用类似于以下序列的内容:

static irqreturn\_t gpio\_rcar\_irq\_handler(int irq, void \*dev\_id)

for each detected GPIO IRQ

generic\_handle\_irq(...);

RT\_FULL：这种类型的处理程序将在-RT上强制形成线程，并且结果是IRQ核心会抱怨使用启用IRQ调用generic\_handle\_irq()，与“链接GPIO irqchips”一样，可以应用同样的解决方法。

1. 嵌套线程GPIO IRQCHIPS：这些是芯片外GPIO扩展器和任何其他GPIO irqchip，它们位于睡眠总线的另一侧。当然，这些需要慢速总线流量读取IRQ状态等的驱动程序，这可能会引起其他IRQs发生，因此不能通过禁用IRQ的快速IRQ处理程序处理。相反，它们需要生成一个线程，然后屏蔽父IRQ线路，直到驱动程序处理中断为止。这种驱动程序的标志是在其中断处理程序中调用以下内容:

static irqreturn\_t foo\_gpio\_irq(int irq, void \*data)

...

handle\_nested\_irq(irq);

线程GPIO irqchips的标志是将.can\_sleep标志设置为true on struct gpio\_chip，表示访问GPIO时此芯片可睡眠。

为了帮助处理GPIO irqchips的设置和管理以及相关的irqdomain和资源分配回调，gpiolib具有一些可通过选择GPIOLIB\_IRQCHIP Kconfig符号来启用的助手：

1. gpiochip\_irqchip\_add()：向gpiochip添加链接的irqchip。它将向所有IRQ回调传递struct gpio\_chip\*，因此回调需要将gpio\_chip嵌入其状态容器中，并使用container\_of()获取指向容器的指针。 （请参见Documentation/driver-model/design-patterns.txt
2. gpiochip\_irqchip\_add\_nested()：将嵌套的irqchip添加到gpiochip中。除此之外，它的工作方式完全与链接的irqchip相同。
3. gpiochip\_set\_chained\_irqchip()：为gpio\_chip设置链接的irq处理程序，从父IRQ传递struct gpio\_chip\*作为处理程序数据。（请注意处理程序数据，因为irqchip数据可能由父irqchip使用！）。
4. gpiochip\_set\_nested\_irqchip()：为gpio\_chip设置嵌套的irq处理程序，从父IRQ传递。由于父IRQ通常由驱动程序显式请求，在这种情况下，除了将所有子IRQ标记为具有其他IRQ作为父项之外，它几乎没有做更多的事情。

如果需要排除某些GPIO不包含在IRQ域中，则可以在调用gpiochip\_add\_data()之前设置gpiochip的.irq.need\_valid\_mask。这将分配一个具有与芯片中GPIO相同数量的位设置的.irq.valid\_mask。驱动程序可以通过清除此掩码中的位来排除GPIO。在调用gpiochip\_irqchip\_add()或gpiochip\_irqchip\_add\_nested()之前必须填充该掩码。

要使用助手，请考虑以下几点：

确保分配了struct gpio\_chip的所有相关成员，以便irqchip可以初始化。例如，.dev和.can\_sleep应正确设置。

在设置调用中将所有处理程序名义设置为handle\_bad\_irq()，并在gpiochip\_irqchip\_add()中将handle\_bad\_irq()作为流处理程序参数传递，如果预计GPIO驱动程序需要调用irqchip .set\_type()回调才能使用/启用GPIO IRQ。然后，在irqchip .set\_type()回调中将处理程序设置为handle\_level\_irq()和/或handle\_edge\_irq()，具体取决于控制器支持什么。

任何IRQ使用者都可以请求来自任何irqchip的IRQ，无论是合并的GPIO+IRQ驱动程序还是什么。基本前提是gpio\_chip和irq\_chip是正交的，并且独立于彼此提供其服务。

gpiod\_to\_irq()只是一个方便函数，用于找出某个GPIO线的IRQ，并且不应依赖它在使用IRQ之前已被调用。

因此，请始终在来自GPIO和irqchip API的各个回调中准备好硬件并准备好进行操作。不要依赖于gpiod\_to\_irq()已经被调用。

这种正交性导致了我们需要解决的歧义：如果子系统内部存在资源使用的竞争（例如某个GPIO线路和寄存器），则需要拒绝某些操作并在gpiolib子系统内部跟踪使用情况。这就是下面的API存在的原因。

#### 锁定IRQ使用

输入GPIO可以用作IRQ信号。发生这种情况时，会请求驱动程序将GPIO标记为正在用作IRQ:

int gpiochip\_lock\_as\_irq(struct gpio\_chip \*chip, unsigned int offset)

这将防止在GPIO IRQ锁定被释放之前使用非IRQ相关的GPIO APIs:

void gpiochip\_unlock\_as\_irq(struct gpio\_chip \*chip, unsigned int offset)

当在GPIO驱动程序中实现irqchip时，应该在irqchip的.startup（）和.shutdown（）回调中通常调用这两个函数。

当使用gpiolib irqchip助手时，这些回调会自动分配。

#### GPIO IRQ芯片的实时合规性

IRQ芯片的任何提供者都需要仔细地量身定制，以支持实时抢占。所有GPIO子系统中的irqchips应该注意这一点，并进行适当的测试，以确保它们启用了实时。因此，请注意上面的“RT\_FULL：”备注。

以下是准备驱动程序进行实时合规性时应遵循的检查清单：

1. 确保spinlock\_t未用作part irq\_chip实现的一部分
2. 确保不使用可睡眠的API作为part irq\_chip实现。如果必须使用可睡眠的API，则可以从.irq\_bus\_lock（）和.irq\_bus\_unlock（）回调中执行这些操作
3. 链接的GPIO irqchips：确保不使用spinlock\_t或任何可睡眠的API从链接的IRQ处理程序中
4. 通用链接的GPIO irqchips：注意generic\_handle\_irq（）调用并应用相应的解决方法
5. 链接的GPIO irqchips：尽可能摆脱链接的IRQ处理程序并使用通用IRQ处理程序
6. regmap\_mmio：可以通过设置.disable\_locking在GPIO驱动程序中处理锁定来禁用regmap中的内部锁定
7. 使用适当的内核实时测试用例测试驱动程序，用于级别和边缘IRQ

#### 请求自己拥有GPIO引脚

有时允许GPIO芯片驱动程序通过gpiolib API请求其自己的GPIO描述符是有用的。 GPIO驱动程序可以使用以下函数请求和释放描述符:

struct gpio\_desc \*gpiochip\_request\_own\_desc(struct gpio\_desc \*desc,

const char \*label)

void gpiochip\_free\_own\_desc(struct gpio\_desc \*desc)

使用gpiochip\_request\_own\_desc（）请求的描述符必须使用gpiochip\_free\_own\_desc（）释放。

由于这些函数不影响模块使用计数，因此必须谨慎使用这些函数。不要使用这些函数请求调用驱动程序不拥有的gpio描述符。

1. <http://www.spinics.net/lists/linux-omap/msg120425.html>
2. <https://lkml.org/lkml/2015/9/25/494>
3. [https://lkml.org/lkml/2015/9/25/49](https://lore.kernel.org/r/1443209283-20781-3-git-send-email-grygorii.strashko@ti.com)5

## GPIO 描述符消费者接口

本文档描述了GPIO框架的消费者接口。请注意，它描述了基于新描述符的接口。有关弃用的基于整数的GPIO界面的描述，请参阅gpio-legacy.txt。

### GPIO消费者指南

没有标准GPIO调用无法工作的驱动程序应具有依赖于GPIOLIB或选择GPIOLIB的Kconfig条目。允许驱动程序获取和使用GPIO的函数可通过包括以下文件获得:

#include <linux/gpio/consumer.h>

在禁用GPIOLIB的情况下，标头文件中所有函数都有静态内联存根。当调用这些存根时，它们将发出警告。这些桩用于两种用例：

使用例如COMPILE\_TEST的简单编译覆盖范围——当前平台是否启用或选择了GPIOLIB并不重要，因为我们不会执行系统。

真正可选的GPIOLIB支持-在某些编译时配置下，驱动程序确实没有使用某些系统上GPIO，但是将在其他编译时配置下使用它。在这种情况下，使用者必须确保不要调用这些函数，否则用户将遇到可能被视为令人生畏的控制台警告。

所有与描述符-based GPIO界面一起工作的函数都以gpiod\_为前缀。 gpio\_前缀用于传统接口。内核中没有其他函数应使用这些前缀。强烈不建议使用传统函数，新代码应仅使用<linux/gpio/consumer.h>和描述符。

### 获取和处理GPIO

使用基于描述符的接口，GPIO使用不可伪造的不透明处理程序进行标识，该处理程序必须通过对gpiod\_get（）函数之一的调用获取。与许多其他内核子系统一样，gpiod\_get（）接受将使用GPIO的设备和所请求的GPIO应执行的函数:

struct gpio\_desc \*gpiod\_get(struct device \*dev, const char \*con\_id,

enum gpiod\_flags flags)

如果使用多个GPIO一起实现某个函数（例如，显示数字的简单LED设备），则可以指定附加的索引参数:

struct gpio\_desc \*gpiod\_get\_index(struct device \*dev,

const char \*con\_id, unsigned int idx,

enum gpiod\_flags flags)

有关DeviceTree情况下con\_id参数的详细说明，请参见GPIO映射

flags参数可用于选择性地指定GPIO的方向和初始值。值可以为：

1. GPIOD\_ASIS或0，表示不初始化GPIO。必须使用专用函数之一稍后设置方向。
2. GPIOD\_IN以初始化GPIO作为输入。
3. GPIOD\_OUT\_LOW初始化GPIO为输出，值为0。
4. GPIOD\_OUT\_HIGH初始化GPIO为输出，值为1。
5. GPIOD\_OUT\_LOW\_OPEN\_DRAIN与GPIOD\_OUT\_LOW相同，但还强制使用开漏线路。
6. GPIOD\_OUT\_HIGH\_OPEN\_DRAIN与GPIOD\_OUT\_HIGH相同，但还强制使用开漏线路。

最后的两个标志用于必须使用开漏的用例，例如I2C：如果该行在映射中尚未配置为开漏，则无论如何都将强制使用开漏，并且将打印警告，需要更新板配置以匹配用例。

两个函数都返回有效的GPIO描述符或可使用IS\_ERR()检查的错误代码（它们不会返回空指针）。如果尚未将GPIO分配给设备/函数/索引三元组，则将返回-ENOENT；如果已分配GPIO但在尝试获取时发生错误，则使用其他错误代码。这对于区分纯粹的错误和可选GPIO参数的GPIO是否存在非常有用。对于GPIO是可选的常见模式，可以使用gpiod\_get\_optional()和gpiod\_get\_index\_optional()函数。如果尚未将GPIO分配给请求的函数，则这些函数返回NULL而不是-ENOENT:

struct gpio\_desc \*gpiod\_get\_optional(struct device \*dev,

const char \*con\_id,

enum gpiod\_flags flags)

struct gpio\_desc \*gpiod\_get\_index\_optional(struct device \*dev,

const char \*con\_id,

unsigned int index,

enum gpiod\_flags flags)

请注意，与gpiolib API的其余部分不同，gpio\_get \* \_optional()函数（及其管理的变体）还返回NULL，即当gpiolib支持被禁用时。这对于驱动程序作者有帮助，因为他们不需要特殊处理-ENOSYS返回代码。但是，系统集成商应谨慎，以在需要gpiolib的系统上启用gpiolib。

对于使用多个GPIO的函数，可以通过一次调用获取所有GPIO:

struct gpio\_descs \*gpiod\_get\_array(struct device \*dev,

const char \*con\_id,

enum gpiod\_flags flags)

此函数返回一个包含描述符数组的struct gpio\_descs:

struct gpio\_descs {

unsigned int ndescs;

struct gpio\_desc \*desc[];

}

如果未将GPIO分配给请求的函数，则返回NULL而不是-ENOENT:

struct gpio\_descs \*gpiod\_get\_array\_optional(struct device \*dev,

const char \*con\_id,

enum gpiod\_flags flags)

这些函数的设备管理变体也已定义:

struct gpio\_desc \*devm\_gpiod\_get(struct device \*dev, const char \*con\_id,

enum gpiod\_flags flags)

struct gpio\_desc \*devm\_gpiod\_get\_index(struct device \*dev,

const char \*con\_id,

unsigned int idx,

enum gpiod\_flags flags)

struct gpio\_desc \*devm\_gpiod\_get\_optional(struct device \*dev,

const char \*con\_id,

enum gpiod\_flags flags)

struct gpio\_desc \*devm\_gpiod\_get\_index\_optional(struct device \*dev,

const char \*con\_id,

unsigned int index,

enum gpiod\_flags flags)

struct gpio\_descs \*devm\_gpiod\_get\_array(struct device \*dev,

const char \*con\_id,

enum gpiod\_flags flags)

struct gpio\_descs \*devm\_gpiod\_get\_array\_optional(struct device \*dev,

const char \*con\_id,

enum gpiod\_flags flags)

可以使用gpiod\_put()函数处理GPIO描述符:

void gpiod\_put(struct gpio\_desc \*desc)

对于GPIO数组，可以使用此函数:

void gpiod\_put\_array(struct gpio\_descs \*descs)

在这些函数之后禁止使用描述符，禁止从使用gpiod\_get\_array()获取的数组中个别释放描述符（使用gpiod\_put()）。设备管理的变体是:

void devm\_gpiod\_put(struct device \*dev, struct gpio\_desc \*desc)

void devm\_gpiod\_put\_array(struct device \*dev, struct gpio\_descs \*descs)

### 使用GPIO

#### 设置方向

驱动程序必须使用GPIO的第一件事是设置方向。如果未向gpiod\_get\*（）提供方向设置标志，则通过调用gpiod\_direction\_\*（）函数之一来完成此操作:

int gpiod\_direction\_input(struct gpio\_desc \*desc)

int gpiod\_direction\_output(struct gpio\_desc \*desc, int value)

返回为零表示成功，否则为负errno。应检查它，因为获取/设置调用不返回错误，并且可能存在误配置。您通常应该从任务上下文发出这些调用。但是，对于自旋锁安全的GPIO，在任务启用之前使用它们作为早期板设置的一部分是可以的。

对于输出GPIO，提供的值将成为初始输出值。这有助于避免系统启动期间的信号故障。

驱动程序还可以查询GPIO的当前方向:

int gpiod\_get\_direction(const struct gpio\_desc \*desc)

此函数返回0表示输出，1表示输入，或错误代码（如果出现错误）。

请注意，GPIO没有默认方向。因此，在未设置其方向之前使用GPIO是不合法的，将导致未定义的行为！

#### 自旋锁安全的GPIO访问

大多数GPIO控制器可以使用内存读/写指令进行访问。它们不需要休眠，并且可以从硬件（非线程化）IRQ处理程序和类似的上下文中安全地完成。

使用以下调用从原子上下文访问GPIO:

int gpiod\_get\_value(const struct gpio\_desc \*desc);

void gpiod\_set\_value(struct gpio\_desc \*desc, int value);

值是布尔值，低为零，非零为高。读取输出引脚的值时，返回的值应该是引脚上看到的。由于包括开放式驱动信号和输出延迟等问题，这并不总是与指定的输出值相匹配。

获取/设置调用不会返回错误，因为“无效的GPIO”应该已经从gpiod\_direction\_\*（）报告了。但是，请注意，并非所有平台都可以读取输出引脚的值；不能安全访问的GPIO应始终返回零。此外，对于不能安全地访问的GPIO，使用这些调用是错误的（请参见下文）。

#### 可能会睡眠的GPIO访问

有些GPIO控制器必须使用像I2C或SPI这样的基于消息的总线进行访问。读写那些GPIO值的命令需要等待到达队列的头部以传输命令并获得其响应。这需要睡眠，无法从IRQ处理程序内部完成。

支持此类型GPIO的平台通过从此调用返回非零值来将它们与其他GPIO区分开来:

int gpiod\_cansleep(const struct gpio\_desc \*desc)

要访问这样的GPIO，定义了一组不同的访问器:

int gpiod\_get\_value\_cansleep(const struct gpio\_desc \*desc)

void gpiod\_set\_value\_cansleep(struct gpio\_desc \*desc, int value)

访问此类GPIO需要一个可能会睡眠的上下文，例如线程化的IRQ处理程序，并且必须使用具有cansleep（）名称后缀但不安全的自旋锁访问器。

除了这些访问器可能会睡眠，并且可以在无法从hardIRQ处理程序访问的GPIO上工作之外，这些调用的行为与自旋锁安全的调用相同。

#### 低活动和开漏语义

作为消费者不必关心物理线路级别，所有gpiod\_set\_value\_xxx（）或gpiod\_set\_array\_value\_xxx（）函数都使用逻辑值进行操作。因此，它们考虑了低活动属性。这意味着它们检查GPIO是否配置为低活动，如果是，则在驱动物理线路电平之前操作传递的值。

同样适用于开放式漏斗管或开放式源输出线：它们不会主动将其输出高（开放式漏斗管）或低（开放式源），它们只是将其输出切换到高阻抗值。消费者不应该关心。（有关GPIO驱动程序接口中的开放式漏斗管的详细信息，请阅读。）

因此，所有gpiod\_set\_（array）\_value\_xxx（）函数都将参数“value”解释为“断言”（“1”）或“未断言”（“0”）。将驱动物理线路电平。

例如，如果设置专用GPIO的低活动属性，并且gpiod\_set\_（array）\_value\_xxx（）通过“asserted”（“1”）传递，则会将物理线路电平降低。

总而言之:

Function (example) line property physical line

gpiod\_set\_raw\_value(desc, 0); don't care low

gpiod\_set\_raw\_value(desc, 1); don't care high

gpiod\_set\_value(desc, 0); default (active high) low

gpiod\_set\_value(desc, 1); default (active high) high

gpiod\_set\_value(desc, 0); active low high

gpiod\_set\_value(desc, 1); active low low

gpiod\_set\_value(desc, 0); open drain low

gpiod\_set\_value(desc, 1); open drain high impedance

gpiod\_set\_value(desc, 0); open source high impedance

gpiod\_set\_value(desc, 1); open source high

可以使用set\_raw / get\_raw函数覆盖这些语义，但应尽量避免，特别是由系统不可知的驱动程序应该不需要关心实际的物理线路水平，而应该担心逻辑值。

#### 访问原始GPIO值

存在需要管理GPIO线路的逻辑状态（即其设备将实际接收到的值）的消费者。以下一组调用无视GPIO的低活动或开漏属性，并使用原始线值进行操作:

int gpiod\_get\_raw\_value(const struct gpio\_desc \*desc)

void gpiod\_set\_raw\_value(struct gpio\_desc \*desc, int value)

int gpiod\_get\_raw\_value\_cansleep(const struct gpio\_desc \*desc)

void gpiod\_set\_raw\_value\_cansleep(struct gpio\_desc \*desc, int value)

int gpiod\_direction\_output\_raw(struct gpio\_desc \*desc, int value)

可以使用以下调用查询和切换GPIO的主动低电平状态:

int gpiod\_is\_active\_low(const struct gpio\_desc \*desc)

注意这些函数应该谨慎使用；驱动程序不应该关心物理线路电平或开漏语义。

#### 使用单个函数调用访问多个GPIO

以下函数获取或设置GPIO数组的值:

int gpiod\_get\_array\_value(unsigned int array\_size,

struct gpio\_desc \*\*desc\_array,

int \*value\_array);

int gpiod\_get\_raw\_array\_value(unsigned int array\_size,

struct gpio\_desc \*\*desc\_array,

int \*value\_array);

int gpiod\_get\_array\_value\_cansleep(unsigned int array\_size,

struct gpio\_desc \*\*desc\_array,

int \*value\_array);

int gpiod\_get\_raw\_array\_value\_cansleep(unsigned int array\_size,

struct gpio\_desc \*\*desc\_array,

int \*value\_array);

void gpiod\_set\_array\_value(unsigned int array\_size,

struct gpio\_desc \*\*desc\_array,

int \*value\_array)

void gpiod\_set\_raw\_array\_value(unsigned int array\_size,

struct gpio\_desc \*\*desc\_array,

int \*value\_array)

void gpiod\_set\_array\_value\_cansleep(unsigned int array\_size,

struct gpio\_desc \*\*desc\_array,

int \*value\_array)

void gpiod\_set\_raw\_array\_value\_cansleep(unsigned int array\_size,

struct gpio\_desc \*\*desc\_array,

int \*value\_array)

该数组可以是任意一组GPIO。如果对应芯片驱动程序支持，函数将同时尝试访问同一个银行或芯片的GPIO。在这种情况下，可以期望显着改善性能。如果无法同时访问，GPIO将按顺序访问。

函数使用四个参数：

array\_size-数组元素数量

desc\_array-GPIO描述符数组

value\_array-用于存储GPIO值（获取）的数组或要分配给GPIO的值数组（设置）

可以使用gpiod\_get\_array()函数或其变体之一来获取描述符数组。如果该函数返回的描述符组与所需的GPIO组匹配，则可以通过仅使用gpiod\_get\_array()返回的struct gpio\_descs来访问这些GPIO:

struct gpio\_descs \*my\_gpio\_descs = gpiod\_get\_array(...);

gpiod\_set\_array\_value(my\_gpio\_descs->ndescs, my\_gpio\_descs->desc,

my\_gpio\_values);

也可以访问完全任意的描述符数组。可以使用任何组合的gpiod\_get()和gpiod\_get\_array()来获取描述符。之后，必须手动设置描述符数组，然后才能将其传递给上述函数之一。在这种情况下，应将array\_info设置为NULL。

注意，为了获得最佳性能，属于同一芯片的GPIO应该在描述符数组内是连续的。

gpiod\_get\_array\_value()及其变体的返回成功为0或错误为负数。请注意与gpiod\_get\_value()不同，后者在成功时返回0或1以传递GPIO值。使用数组函数时，GPIO值存储在value\_array中而不是作为返回传递回来。

#### GPIO映射到IRQ

GPIO线通常可以用作IRQ。可以使用以下调用获取与给定GPIO对应的IRQ号：

int gpiod\_to\_irq(const struct gpio\_desc \*desc)

如果无法进行映射，它将返回一个IRQ号或一个负的errno代码（最可能是因为该特定GPIO无法用作IRQ）。使用gpiod\_direction\_input()设置为输入时使用未设置为输入的GPIO或使用并非来自gpiod\_to\_irq()的IRQ号是未经检查的错误。不允许gpiod\_to\_irq()睡眠。

从gpiod\_to\_irq()返回的非错误值可以传递给request\_irq()或free\_irq()。它们通常会通过特定于板的初始化代码存储到平台设备的IRQ资源中。请注意，IRQ触发选项是IRQ接口的一部分，例如IRQF\_TRIGGER\_FALLING，系统唤醒功能也是如此。

### GPIO和ACPI

在ACPI系统上，GPIO由设备的\_CRS配置对象列出的GpioIo（）/ GpioInt（）资源描述。这些资源不为GPIO提供连接ID（名称），因此需要使用其他机制来实现此目的。

符合ACPI 5.1或更高版本的系统可以提供\_DSD配置对象，该对象可以用于为\_CRS中的GpioIo（）/ GpioInt（）资源描述的特定GPIO提供连接ID等内容。如果是这种情况，它将由GPIO子系统自动处理。但是，如果不存在\_DSD，则需要由设备驱动程序提供GpioIo（）/ GpioInt（）资源和GPIO连接ID之间的映射。

有关详细信息，请参见Documentation/acpi/gpio-properties.txt

### 与遗留GPIO子系统交互

许多内核子系统和驱动程序仍然使用基于整数的遗留接口处理GPIO。强烈建议将它们更新为新的gpiod接口。对于需要同时使用两个接口的情况，以下两个函数允许将GPIO描述符转换为GPIO整数命名空间，反之亦然：

int desc\_to\_gpio(const struct gpio\_desc \*desc)

struct gpio\_desc \*gpio\_to\_desc(unsigned gpio)

由desc\_to\_gpio()返回的GPIO编号只要GPIO描述符未被释放，就可以安全使用。同样地，传递给gpio\_to\_desc()的GPIO编号必须已经正确获取，且只有在释放了GPIO编号后才能使用返回的GPIO描述符。

使用一个API获取的GPIO，再使用另一个API释放是禁止的且未经检查的错误。

## GPIO 映射

本文档说明GPIO如何分配给给定的设备和功能。请注意，它仅适用于新的基于描述符的接口。关于废弃的整数型GPIO接口的描述，请参考Legacy GPIO接口（实际上，旧接口无法进行真正的映射；只是从某处获取整数，然后请求相应的GPIO）。

所有平台均可启用GPIO库，但如果平台严格要求存在GPIO功能，则需要从其 Kconfig 中选择GPIOLIB。然后，GPIO如何映射取决于平台用于描述其硬件布局的内容。目前，可以通过设备树、ACPI和平台数据定义映射。

### 设备树

GPIO可以很容易地映射到设备和函数上。实际上的映射方式取决于提供GPIO的GPIO控制器，请参阅控制器的设备树绑定。

GPIO映射定义在使用者设备节点中的名为<function>-gpios的属性中，<function> 是驱动程序将通过gpiod\_get()请求的功能。例如:

foo\_device {

compatible = "acme,foo";

...

led-gpios = <&gpio 15 GPIO\_ACTIVE\_HIGH>, /\* red \*/

<&gpio 16 GPIO\_ACTIVE\_HIGH>, /\* green \*/

<&gpio 17 GPIO\_ACTIVE\_HIGH>; /\* blue \*/

power-gpios = <&gpio 1 GPIO\_ACTIVE\_LOW>;

};

名为<function>-gpio的属性也被认为是有效的，旧版本的绑定使用它，但仅出于兼容性原因而受支持，不应用于新版本的绑定中，因为它已被弃用。

此属性将使GPIO 15、16和17在“led”功能下对驱动程序可用，GPIO 1作为“电源”GPIO:

struct gpio\_desc \*red, \*green, \*blue, \*power;

red = gpiod\_get\_index(dev, "led", 0, GPIOD\_OUT\_HIGH);

green = gpiod\_get\_index(dev, "led", 1, GPIOD\_OUT\_HIGH);

blue = gpiod\_get\_index(dev, "led", 2, GPIOD\_OUT\_HIGH);

power = gpiod\_get(dev, "power", GPIOD\_OUT\_HIGH);

led GPIO为高电平有效，而电源GPIO将为低电平有效（即gpiod\_is\_active\_low（power）将为真）。

gpiod\_get()函数的第二个参数，con\_id字符串，必须是设备树中使用的GPIO后缀（“gpios”或“gpio”），它的<function>-前缀。以以上“led-gpios”示例为例，将前缀作为con\_id参数而不带“ - ”：“led”。

在内部，GPIO子系统将<function>-前缀与字符串（“gpios”或“gpio”）相结合，以获取所需的字符串（snprintf（...“％s-％s”，con\_id，gpio\_suffixes[]）。

### ACPI

ACPI也支持类似于DT的GPIO函数名。可以通过ACPI 5.1引入的\_DSD（设备特定数据）将上述DT示例转换为等效的ACPI描述:

Device (FOO) {

Name (\_CRS, ResourceTemplate () {

GpioIo (Exclusive, PullUp, 0, 0, IoRestrictionOutputOnly,

"\\\_SB.GPI0", 0, ResourceConsumer) { 15 } // red

GpioIo (Exclusive, PullUp, 0, 0, IoRestrictionOutputOnly,

"\\\_SB.GPI0", 0, ResourceConsumer) { 16 } // green

GpioIo (Exclusive, PullUp, 0, 0, IoRestrictionOutputOnly,

"\\\_SB.GPI0", 0, ResourceConsumer) { 17 } // blue

GpioIo (Exclusive, PullNone, 0, 0, IoRestrictionOutputOnly,

"\\\_SB.GPI0", 0, ResourceConsumer) { 1 } // power

})

Name (\_DSD, Package () {

ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),

Package () {

Package () {

"led-gpios",

Package () {

^FOO, 0, 0, 1,

^FOO, 1, 0, 1,

^FOO, 2, 0, 1,

}

},

Package () {

"power-gpios",

Package () { ^FOO, 3, 0, 0 }

},

}

})

}

有关ACPI GPIO绑定的更多信息，请参阅与GPIO相关的\_DSD设备属性。

### 平台数据

最后，GPIO可以使用平台数据绑定到设备和功能。希望这样做的板文件需要包含以下标头：

#include <linux/gpio/machine.h>

通过查找表的方式，将GPIO映射到gpiod\_lookup结构的实例中。定义了两个宏以帮助声明这样的映射:

GPIO\_LOOKUP(chip\_label, chip\_hwnum, con\_id, flags)

GPIO\_LOOKUP\_IDX(chip\_label, chip\_hwnum, con\_id, idx, flags)

其中

1. chip\_label是提供GPIO的gpiod\_chip实例的标签
2. chip\_hwnum是芯片内 GPIO 的硬件编号
3. con\_id是来自设备的GPIO功能名称，它可以为NULL，在这种情况下，它将匹配任何功能。
4. idx是功能内的GPIO的索引。
5. flags被定义为指定以下属性：

GPIO\_ACTIVE\_HIGH - GPIO线为高电平有效

GPIO\_ACTIVE\_LOW - GPIO线为低电平有效

GPIO\_OPEN\_DRAIN - GPIO线设置为开漏

GPIO\_OPEN\_SOURCE - GPIO线设置为开源

GPIO\_PERSISTENT - GPIO线在挂起/恢复过程中保持持久并保持其值。

GPIO\_TRANSITORY - GPIO线是短暂的，可能会在挂起/恢复期间失去其电气状态。

将来，这些标志可能会扩展以支持更多属性。

请注意：GPIO\_LOOKUP（）只是GPIO\_LOOKUP\_IDX（）（其中idx = 0）的快捷方式。

可以定义查找表如下，并使用空条目定义其末尾。表的“dev\_id”字段是将使用这些GPIO的设备的标识符。它可以为NULL，在这种情况下，将使用NULL设备来匹配对gpiod\_get（）的调用。

struct gpiod\_lookup\_table gpios\_table = {

.dev\_id = "foo.0",

.table = {

GPIO\_LOOKUP\_IDX("gpio.0", 15, "led", 0, GPIO\_ACTIVE\_HIGH),

GPIO\_LOOKUP\_IDX("gpio.0", 16, "led", 1, GPIO\_ACTIVE\_HIGH),

GPIO\_LOOKUP\_IDX("gpio.0", 17, "led", 2, GPIO\_ACTIVE\_HIGH),

GPIO\_LOOKUP("gpio.0", 1, "power", GPIO\_ACTIVE\_LOW),

{ },

},

};

可以通过以下方式将表添加到板代码中:

gpiod\_add\_lookup\_table(&gpios\_table);

然后控制“foo.0”的驱动程序将能够获取其GPIO，如下所示:

struct gpio\_desc \*red, \*green, \*blue, \*power;

red = gpiod\_get\_index(dev, "led", 0, GPIOD\_OUT\_HIGH);

green = gpiod\_get\_index(dev, "led", 1, GPIOD\_OUT\_HIGH);

blue = gpiod\_get\_index(dev, "led", 2, GPIOD\_OUT\_HIGH);

power = gpiod\_get(dev, "power", GPIOD\_OUT\_HIGH);

由于“led” GPIO被映射为高电平有效，因此该示例将将其信号切换为1，即启用LED。对于“power” GPIO，它被映射为Active-Low，其实际信号在此代码后将为0。与遗留的整数GPIO接口不同，Active-Low属性在映射期间处理，因此对GPIO消费者透明。

一组函数（如gpiod\_set\_value（））可用于使用新的描述符导向界面。

使用平台数据的板还可以通过定义GPIO hog表来占用GPIO线。

struct gpiod\_hog gpio\_hog\_table[] = {

GPIO\_HOG("gpio.0", 10, "foo", GPIO\_ACTIVE\_LOW, GPIOD\_OUT\_HIGH),

{ }

};

可以将表添加到板代码中，如下所示:

gpiod\_add\_hogs(gpio\_hog\_table);

当gpiochip被创建或（如果芯片早先被创建）当hog表被注册时，该线将被占用。

## 使用GPIO的子系统驱动程序

请注意，常见GPIO任务的标准内核驱动程序已经存在，并且将为工作提供正确的内核和用户空间API / ABI，并且这些驱动程序可以很容易地使用硬件描述（如设备树或ACPI）与其他内核子系统进行互连：

leds-gpio：drivers / leds / leds-gpio.c将处理连接到GPIO线的LED，为您提供LED sysfs界面

ledtrig-gpio：drivers / leds / trigger / ledtrig-gpio.c将提供LED触发器，即LED将根据GPIO行的高低点亮/熄灭（并且该LED可以像上面那样使用leds-gpio）。

gpio-keys：drivers / input / keyboard / gpio\_keys.c用于当GPIO线可以响应按键时生成中断。还支持去抖动。

gpio-keys-polled：drivers / input / keyboard / gpio\_keys\_polled.c用于当GPIO线无法生成中断时（因此需要定期轮询计时器来检查）。

gpio\_mouse：drivers / input / mouse / gpio\_mouse.c用于仅使用GPIO和没有鼠标端口提供高达三个按钮的鼠标。您可以剪断鼠标电缆并将电线连接到GPIO线，或者将鼠标连接器焊接到属于这种类型的线上以获得更永久的解决方案。

gpio-beeper：drivers / input / misc / gpio-beeper.c用于通过连接到GPIO线上的外部扬声器提供蜂鸣声。

extcon-gpio：drivers / extcon / extcon-gpio.c用于在需要读取外部连接器状态（例如音频驱动程序或HDMI连接器的耳机线）时使用。它将提供比GPIO更好的用户空间sysfs界面。

restart-gpio：drivers / power / reset / gpio-restart.c用于通过拉动GPIO线重启/重启系统，并将注册重启处理程序，因此用户空间可以发出正确的系统调用来重新启动系统。

poweroff-gpio：drivers / power / reset / gpio-poweroff.c用于通过拉动GPIO线将系统关闭，并注册pm\_power\_off（）回调，以便用户空间可以发出正确的系统调用以关闭系统。

gpio-gate-clock：drivers / clk / clk-gpio.c用于控制使用GPIO的门控时钟（关闭/打开），并与时钟子系统集成。

i2c-gpio：drivers / i2c / busses / i2c-gpio.c用于通过敲两个GPIO线（双线SDA和SCL）来驱动I2C总线。它将出现在系统中，并使其可以像任何其他I2C总线驱动程序一样连接I2C设备的驱动程序。

spi\_gpio: drivers/spi/spi-gpio.c 用于通过 GPIO 接口（bitbang) 驱动一个 SPI 总线(可变数量的线，至少 SCK 线和可选的 MISO、MOSI 和晶片选择线)。它将出现在系统中任何其他 SPI 总线的地方，并可用于与总线上的 SPI 设备驱动程序连接。例如，使用 MMC/SD 卡子系统中的 mmc\_spi 主机，可以将任何 MMC/SD 卡连接到该 SPI 上。

w1-gpio: drivers/w1/masters/w1-gpio.c 用于使用 GPIO 线驱动单线总线，与 W1 子系统集成并像任何其他 W1 设备一样处理总线上的设备。

gpio-fan: drivers/hwmon/gpio-fan.c 用于控制通过 GPIO 线连接的风扇以冷却系统（可选择 GPIO 报警线），为您提供内核和 sysfs 接口。

gpio-regulator: drivers/regulator/gpio-regulator.c 用于通过拉动 GPIO 线控制提供某个电压的稳压器，与稳压器子系统集成并为您提供所有正确的接口。

gpio-wdt: drivers/watchdog/gpio\_wdt.c 用于提供一个看门狗定时器，周期性地对连接到 GPIO 线的硬件进行“ping”（1到0到1的切换）。如果该硬件未定期收到“ping”信号，则会重置系统。

gpio-nand: drivers/mtd/nand/raw/gpio.c 用于通过一组简单的 GPIO 线（RDY、NCE、ALE、CLE、NWP）连接 NAND 闪存芯片。它与 NAND 闪存 MTD 子系统互动，并提供与任何其他 NAND 驱动硬件一样的芯片访问和分区解析。

ps2-gpio: drivers/input/serio/ps2-gpio.c 用于通过 bit banging 两个 GPIO 线驱动 PS/2（IBM）serio 总线，数据和时钟线。它将出现在系统中任何其他 serio 总线的位置，并可用于连接基于 PS/2 协议的设备驱动程序，例如键盘和其他 PS/2 执行的设备。

cec-gpio: drivers/media/platform/cec-gpio/ 仅使用 GPIO 与 CEC 消费电子控制总线进行交互。它用于与 HDMI 总线上的设备通信。

除此之外，还有一些特殊的 GPIO 驱动程序，例如 MMC/SD 中用于读取卡片检测和写保护 GPIO 线的驱动程序，以及 TTY 串行子系统中使用两个 GPIO 线模拟 MCTRL（调制解调器控制）信号 CTS/RTS 的驱动程序。MTD NOR 闪存也有额外的 GPIO 线附加组件，尽管通常地址总线直接连接到闪存上。

使用这些驱动程序而不是直接从用户空间与 GPIO 进行通信，将更好地集成内核框架。毫无疑问，仅使用适当的内核驱动程序将简化和加速您的嵌入式编程，特别是在提供现成组件的情况下。

## 传统 GPIO 接口

这提供了有关 Linux 上 GPIO 访问约定的概述。

这些调用使用 gpio\_\* 命名前缀。其他调用不应使用该前缀或相关的 \_\_gpio\_\* 前缀。

### 什么是 GPIO？

“通用输入/输出”（GPIO）是一种灵活的由软件控制的数字信号。它们来自许多种芯片，并为与嵌入式和定制硬件一起工作的 Linux 开发人员所熟悉。每个 GPIO 代表连接到特定引脚或 BGA 包上的特定 pin 或“ball”的比特位。板子原理图显示了哪些外部硬件连接到哪些 GPIO。驱动程序可以编写通用程序，以便板子设置代码将这种 pin 配置数据传递给驱动程序。

系统级芯片 (SOC) 处理器严重依赖 GPIO。在某些情况下，每个非专用引脚都可以配置为 GPIO。大多数芯片至少有几十个这样的引脚。可编程逻辑设备 (如 FPGA) 可轻松提供 GPIO；像电源管理器和音频编解码器这样的多功能芯片通常具有一些这样的引脚，帮助处理 SOCs 上的引脚不足问题；还有通过 I2C 或 SPI 串行总线连接的“GPIO 扩展器”芯片。大多数 PC 南桥芯片拥有几十个 GPIO 可能引脚（只有 BIOS 固件知道它们如何使用）。

GPIO 的确切能力因系统而异。常见选项：

1. 可写输出值（高= 1，低= 0）。某些芯片还具有有关如何驱动该值的选项，因此可能仅驱动一个值来支持其他值的 “或” 电线和类似的方案（特别是，“开漏”信号）
2. 同样，输入值也是可读的（1, 0）。某些芯片支持将配置为“输出”的引脚的读取反馈，这在这种“或”线路的情况下非常有用（以支持双向信号）。GPIO 控制器可以具有输入去抖动/去弹跳逻辑，有时带有软件控件。
3. 输入经常被用作IRQ信号，通常是边沿触发，但有时也是电平触发。这些IRQ可以配置为系统唤醒事件，以从低功耗状态唤醒系统。
4. 通常，GPIO可根据不同产品板的需要配置为输入或输出；也存在单方向的GPIO。
5. 大多数GPIO在持有自旋锁的同时可访问，但通过串行总线访问的GPIO通常无法访问。一些系统同时支持这两种类型。

在给定的板上，每个GPIO都用于一个特定的目的，如监视MMC/SD卡的插入/移除、检测卡的写保护状态、驱动LED、配置收发器、bitbanging串行总线、戳硬件看门狗、感应开关等等。

### GPIO约定

请注意，这称为“约定”，因为你不需要按照这种方式操作，如果你不这样做也不会犯罪。有些情况下，可移植性并不是主要问题；GPIO通常用于特定于板的胶合逻辑，甚至可能在板上的不同修订版之间发生变化，并且不能在不同的连线板上使用。只有最低公共功能可以非常可移植。其他功能是平台特定的，这对于胶合逻辑非常重要。

此外，这并不需要任何实现框架，只需要一个接口。一个平台可能通过简单的内联函数访问芯片寄存器来实现它；另一个平台可能通过抽象来委托实现几种非常不同类型的GPIO控制器。 (本文档后面描述支持这种实现策略的一些可选代码，但作为GPIO接口的客户端的驱动程序不必关心它是如何实现的。)

也就是说，如果约定在其平台上得到支持，驱动程序应尽可能使用它。如果GPIO功能是严格必需的，则平台必须选择GPIOLIB。不能在没有标准GPIO调用的情况下工作的驱动程序应具有依赖于GPIOLIB的Kconfig条目。当驱动程序使用以下包含文件时，GPIO调用可用，作为“真实代码”或优化掉的存根:

#include <linux/gpio.h>

如果你坚持遵循这个约定，那么其他开发人员更容易看出你的代码在做什么，并帮助维护它。

请注意，这些操作包括在需要使用它们的平台上的I/O障碍；驱动程序不需要显式添加它们。

#### 识别GPIO

GPIO由范围为0..MAX\_INT的无符号整数标识。这保留了“负”数字用于其他用途，如将信号标记为“在此板上不可用”或指示故障。不触摸底层硬件的代码将这些整数视为不透明的cookie。

平台定义它们如何使用这些整数，并通常#define符号用于GPIO线，以便特定于板的设置代码直接对应于相关原理图。相比之下，驱动程序应该只使用从该设置代码传递给它们的GPIO编号，并使用platform\_data保存特定于板的引脚配置数据（以及它们需要的其他特定于板的数据）。这可以避免可移植性问题。

例如，一个平台使用32-159个GPIO；而另一个使用0..63个数字，其中一个GPIO控制器使用64-79个数字，另一个特定的板使用80-95个数字与FPGA。这些数字不需要连续；这些平台中的任何一个还可以使用2000-2063个数字，以标识I2C GPIO扩展器银行中的GPIO。

如果你想用一个无效的GPIO编号来初始化一个结构，请使用一些负数（例如“-EINVAL”）；这将永远不会有效。要测试来自这样一个结构的这样一个数字是否可以引用一个GPIO，你可以使用这个谓词:

int gpio\_is\_valid(int number);

那些无效的数字将被调用拒绝或释放GPIO时拒绝或释放（见下文）。其他数字也可能被拒绝；例如，一个数字可能是有效的，但在给定的板上暂时没有使用。

是否支持多个GPIO控制器是一个特定于平台的实现问题，这样的支持是否可以在GPIO数字空间中留下“空洞”，以及是否可以在运行时添加新的控制器。这些问题可能会影响是否相邻的GPIO编号都有效。

#### 使用GPIO

系统应该使用gpio\_request()调用分配一个GPIO的第一件事；请看后面。

使用GPIO的下一件事，通常是在设置平台设备时（使用GPIO时）在板设置代码中标记其方向:

/\* set as input or output, returning 0 or negative errno \*/

int gpio\_direction\_input(unsigned gpio);

int gpio\_direction\_output(unsigned gpio, int value);

返回为成功时为零，否则为负数 errno。应该检查返回，因为get/set调用没有错误返回，并且可能会发生错误配置。通常应该在任务上下文中发出这些调用。然而，对于自旋锁安全的GPIO，可以在启用任务之前使用它们作为早期板卡设置的一部分。

对于输出GPIO，提供的值成为初始输出值。这有助于在系统启动期间避免信号抖动。

为了与GPIO的传统接口兼容，隐式设置GPIO方向请求该GPIO（见下文），如果该GPIO尚未被请求。这种兼容性正在被可选的gpiolib框架移除。

如果GPIO编号无效或无法在该模式下使用特定GPIO，则设置方向可能失败。通常依靠引导固件正确设置方向是一个不好的主意，因为它可能没有经过验证，仅仅是对Linux进行了引导。 （类似地，该板卡设置代码可能需要将该引脚复用为GPIO，并适当地配置上拉/下拉。）

#### 自旋锁安全的GPIO访问

大多数GPIO控制器可以使用内存读/写指令访问。它们不需要睡眠，并且可以安全地从硬件（非线程）IRQ处理程序和类似上下文中执行。

使用以下调用以访问这些GPIO，对于 gpio\_cansleep() 将始终返回false的GPIO（参见下文）:

/\* GPIO INPUT: return zero or nonzero \*/

int gpio\_get\_value(unsigned gpio);

/\* GPIO OUTPUT \*/

void gpio\_set\_value(unsigned gpio, int value);

这些值是布尔值，零表示低电平，非零表示高电平。当读取输出引脚的值时，返回的值应该是看到的值 ... 由于存在开漏信号和输出延迟等问题，这不总是与指定的输出值匹配。

由于“无效GPIO”应该已经从gpio\_direction\_\*()中报告，所以这些get/set调用没有错误返回。但是，请注意，并非所有平台都可以读取输出引脚的值；不能读取值的那些应该总是返回零。此外，对于无法安全访问而需要睡眠的GPIO使用这些调用是错误的。

鼓励平台特定的实现优化两个调用，以访问GPIO值在GPIO编号（以及输出，值）是恒定的情况下。在这种情况下，它们通常只需要几个指令（读取或写入硬件寄存器），并且不需要自旋锁。这些优化的调用可以使比特带应用程序比在子程序调用上花费数十个指令更有效（在空间和时间上）。

#### 可能休眠的GPIO访问

一些GPIO控制器必须使用消息传递总线（如I2C或SPI）访问。读取或写入这些GPIO值的命令需要等待进入队列的队列以传输命令并获取其响应。这需要睡眠，无法从IRQ处理程序内执行。

支持此类型的GPIO的平台通过从该调用返回非零来将它们与其他GPIO区分开（它需要有效的GPIO编号，这应该已经分配给 gpio\_request之前）:

int gpio\_cansleep(unsigned gpio);

为了访问这些GPIO，定义了不同的访问器集:

/\* GPIO INPUT: return zero or nonzero, might sleep \*/

int gpio\_get\_value\_cansleep(unsigned gpio);

/\* GPIO OUTPUT, might sleep \*/

void gpio\_set\_value\_cansleep(unsigned gpio, int value);

访问这些GPIO需要可能休眠的上下文，例如线程中断处理程序，这些访问器必须使用带有cansleep()名称后缀的自旋锁安全访问器。

除了这些访问器可能会休眠，而且将在无法从hardIRQ handler中访问的GPIO上工作以外，这些调用与自旋锁安全调用相同。

要设置和配置此类GPIO，必须从可能休眠的上下文中进行调用，因为它们可能需要访问GPIO控制器芯片（这些设置调用通常是从板卡设置或驱动程序探测/拆卸代码中进行的，因此这是一个简单的限制）:

gpio\_direction\_input()

gpio\_direction\_output()

gpio\_request()

## gpio\_request\_one()

## gpio\_request\_array()

## gpio\_free\_array()

gpio\_free()

gpio\_set\_debounce()

#### 声明和释放GPIO

为了帮助捕捉系统配置错误，定义了两个调用:

/\* request GPIO, returning 0 or negative errno.

\* non-null labels may be useful for diagnostics.

\*/

int gpio\_request(unsigned gpio, const char \*label);

/\* release previously-claimed GPIO \*/

void gpio\_free(unsigned gpio);

将无效的GPIO编号传递给 gpio\_request() 将失败，就像请求已经使用该调用声明的GPIO一样。必须检查 gpio\_request() 的返回。通常应该在任务上下文中发出这些调用。然而，对于自旋锁安全的GPIO，可以在启用任务之前请求GPIO，作为早期板卡设置的一部分。

这些调用有两个基本目的。一是标记实际使用的信号作为GPIO，以便进行更好的诊断。系统可能有几百个潜在的GPIO，但通常在任何给定的板上只使用十几个。另一个目的是捕获冲突，当（a）两个或多个驱动程序错误地认为它们独占该信号，或者（b）有些东西错误地认为安全地移除了管理处于活动使用状态的信号所需的驱动程序时，识别错误。也就是说，请求GPIO可以作为一种锁。

一些平台还可以使用活动的GPIO的知识来进行电源管理，例如通过关闭未使用的芯片区域以及更容易地关闭未使用的时钟。

对于使用已知pinctrl子系统引脚的GPIO，应该通知该子系统它们的使用； gpiolib驱动程序的.request（）操作可以调用pinctrl\_gpio\_request（），gpiolib驱动程序的.free（）操作可以调用pinctrl\_gpio\_free（）。 pinctrl子系统允许pinctrl\_gpio\_request（）同时成功地拥有设备用于引脚复用的引脚或pingroup。

必要的引脚复用硬件编程，将GPIO信号路由到适当的引脚，应该发生在GPIO驱动程序的.direction\_input（）或.direction\_output（）操作内，并在设置输出GPIO的值之后发生。这允许从引脚的特殊功能到GPIO的无故障迁移。在使用非GPIO HW块通常驱动的信号上实现解决方法时，有时需要这样做。一些平台允许将一些或全部GPIO信号路由到不同的引脚。同样，GPIO或引脚的其他方面也可能需要配置，例如上拉/下拉。平台软件应安排在调用gpio\_request（）之前配置任何此类细节，例如使用pinctrl子系统的映射表，以便GPIO用户无需知道这些细节。

还要注意，您有责任在释放GPIO之前停止使用它。

考虑到大多数情况下GPIO实际上是在它们被声明后立即配置的，还定义了三个额外的调用:

/\* request a single GPIO, with initial configuration specified by

\* 'flags', identical to gpio\_request() wrt other arguments and

\* return value

\*/

int gpio\_request\_one(unsigned gpio, unsigned long flags, const char \*label);

/\* request multiple GPIOs in a single call

\*/

int gpio\_request\_array(struct gpio \*array, size\_t num);

/\* release multiple GPIOs in a single call

\*/

void gpio\_free\_array(struct gpio \*array, size\_t num);

当前定义“标志”以指定以下属性的位置：

GPIOF\_DIR\_IN-将方向配置为输入

GPIOF\_DIR\_OUT-将方向配置为输出

GPIOF\_INIT\_LOW-作为输出，将初始电平设置为低

GPIOF\_INIT\_HIGH-作为输出，将初始电平设置为高

GPIOF\_OPEN\_DRAIN-GPIO引脚为开漏类型。

GPIOF\_OPEN\_SOURCE-GPIO引脚为开源类型。

GPIOF\_EXPORT\_DIR\_FIXED-将GPIO导出到sysfs，保留方向

GPIOF\_EXPORT\_DIR\_CHANGEABLE-还要导出，允许更改方向

由于当配置为输出时，GPIOF\_INIT\_\*有效，因此将适当的组合分组为：

GPIOF\_IN-配置为输入

GPIOF\_OUT\_INIT\_LOW-配置为输出，初始电平为低

GPIOF\_OUT\_INIT\_HIGH-配置为输出，初始电平为高

当将标志设置为GPIOF\_OPEN\_DRAIN时，它将假定引脚为开漏类型。这些引脚在输出模式下不会被驱动到1。在这些引脚上需要连接上拉电阻。通过启用此标志，gpio lib会在以输出模式设置值为1时将方向设置为输入，以使引脚为高电平。通过在输出模式中驱动值0将引脚设为低电平。

当将标志设置为GPIOF\_OPEN\_SOURCE时，它将假定引脚为开源类型。这些引脚在输出模式下不会被驱动到0.在这些引脚上需要连接下拉电阻。通过启用此标志，gpio lib会在以输出模式设置值为0时将方向设置为输入，以使引脚LOW。通过在输出模式中驱动值为1使引脚HIGH。

在未来，这些标志可以扩展以支持更多属性。

此外，为了方便声明/释放多个GPIO，引入了“struct gpio”以封装所有三个字段:

struct gpio {

unsigned gpio;

unsigned long flags;

const char \*label;

};

一个典型的用法示例:

static struct gpio leds\_gpios[] = {

{ 32, GPIOF\_OUT\_INIT\_HIGH, "Power LED" }, /\* default to ON \*/

{ 33, GPIOF\_OUT\_INIT\_LOW, "Green LED" }, /\* default to OFF \*/

{ 34, GPIOF\_OUT\_INIT\_LOW, "Red LED" }, /\* default to OFF \*/

{ 35, GPIOF\_OUT\_INIT\_LOW, "Blue LED" }, /\* default to OFF \*/

{ ... },

};

err = gpio\_request\_one(31, GPIOF\_IN, "Reset Button");

if (err)

...

err = gpio\_request\_array(leds\_gpios, ARRAY\_SIZE(leds\_gpios));

if (err)

...

gpio\_free\_array(leds\_gpios, ARRAY\_SIZE(leds\_gpios));

#### GPIO 映射到 IRQ

GPIO 号和 IRQ 号都是无符号整数。它们构成两个逻辑上不同的名称空间（GPIO 0 不必使用 IRQ 0）。您可以使用如下调用来映射它们:

/\* map GPIO numbers to IRQ numbers \*/

int gpio\_to\_irq(unsigned gpio);

/\* map IRQ numbers to GPIO numbers (avoid using this) \*/

int irq\_to\_gpio(unsigned irq);

这些调用返回另一个名称空间中的相应数字，否则返回负的错误码（例如，有些 GPIO 不能用作 IRQ）。使用未经设置为输入的 GPIO 数字（即 gpio\_direction\_input()），或使用未经 gpio\_to\_irq() 原始输出的 IRQ 数字，都是未检查的错误。

gpio\_to\_irq() 返回的非错误值可以传递给 request\_irq() 或 free\_irq()。它们经常被存储到平台设备的 IRQ 资源中，由特定于板子的初始化代码完成。请注意，IRQ 触发选项是 IRQ 接口的一部分，例如 IRQF\_TRIGGER\_FALLING，系统唤醒功能也是。

irq\_to\_gpio() 返回的非错误值通常与 gpio\_get\_value() 一起使用，例如，在 IRQ 边缘触发时初始化或更新驱动程序状态。请注意，某些平台不支持此反向映射，应避免使用它。

#### 模拟开漏信号

有时，共享信号需要使用“开漏”信号传输，仅低电平实际上被驱动。这个术语适用于 CMOS 晶体管；TTL 则使用“开集合器”。上拉电阻引起高电平。这有时被称为“线与”；或者更实际地，从负逻辑（低=true）的角度来看，这是“线或”。

一个常见的开漏信号示例是共享的主动低电平 IRQ 线。此外，双向数据总线信号有时使用开漏信号传输。

一些 GPIO 控制器直接支持开漏输出；许多不支持。当您需要开漏信号传输，但硬件不直接支持它时，您可以使用一种常用的方法来模拟任何既可以用作输入又可以用作输出的 GPIO 引脚：

低：gpio\_direction\_output(gpio, 0) … 这驱动信号并覆盖上拉电阻。

高：gpio\_direction\_input(gpio) … 这关掉输出，所以上拉电阻（或其他设备）控制信号。

如果您“驱动”信号高，但 gpio\_get\_value(gpio) 报告低值（经过适当的上升时间后），则知道某些其他组件将共享信号驱动为低电平。这不一定是错误。作为一个常见的例子，这是 I2C 时钟被拉伸的方式：需要较慢时钟的从设备延迟 SCK 上升沿，I2C 主机调整其信号速率。

#### GPIO 控制器和 pinctrl 子系统

SOC 上的 GPIO 控制器可能与 pinctrl 子系统紧密耦合，因为使用 pin 组件功能的其他函数可以一起使用 pin。我们已经讨论了例如 GPIO 控制器需要通过以下任何调用来保留引脚或设置引脚方向的情况:

pinctrl\_gpio\_request()

pinctrl\_gpio\_free()

pinctrl\_gpio\_direction\_input()

pinctrl\_gpio\_direction\_output()

但是，引脚控制子系统如何将 GPIO 数字（这是全局业务）与特定 pin 控制器上的某个引脚相互关联？

这是通过注册“范围”来实现的，它们本质上是交叉参考表。这些在 PINCTRL（PIN CONTROL）子系统中有描述。

虽然引脚分配完全由 pinctrl 子系统管理，但 gpio（在 gpiolib 下）仍由 gpio 驱动程序维护。在 SOC 中，可能有不同的引脚范围由不同的 gpio 驱动程序管理。

将每个引脚范围公告给引脚控制子系统后，才会调用 ‘pinctrl\_gpio\_request’ 来请求 pinctrl 子系统在任何 gpio 使用之前准备相应的引脚。对此，GPIO 控制器可以向 pinctrl 子系统注册其 pin 范围。目前有两种方法可以实现这一点：具有 DT 或无 DT 的支持。

有关具有 DT 支持的信息，请参阅 Documentation/devicetree/bindings/gpio/gpio.txt。

对于无 DT 支持，用户可以调用 gpiochip\_add\_pin\_range() 并传递适当的参数，以在 pinctrl 驱动程序中注册一个 gpio 引脚范围。对于这个确切的名字字符串，pinctrl 设备必须作为该例程的参数之一传递过去。

### 这些约定忽略了什么？

这些约定中最大的问题之一是引脚复用，因为这是高度特定于芯片并且不可移植的。一个平台可能不需要显式的复用；另一个可能只有两个选项来使用任何给定的引脚。另一个可能每个引脚有八个选项；另一个可能能够将给定的 GPIO 路由到多个引脚中的任何一个。 （是的，这些示例都来自今天运行 Linux 的系统。）

与复用相关的是一些平台上集成的上拉或下拉的配置和启用。并非所有平台都支持它们，或以相同方式支持它们；任何给定的板可能使用外部上拉（或下拉），因此不能使用芯片上的上拉或下拉。（当电路需要5千欧姆时，芯片上的100千欧姆电阻器无法完成。）同样，驱动强度（2毫安比和20毫安比）和电压（1.8V和3.3V）是平台特定的问题，像（不）具有可配置引脚和GPIO之间的一对一对应关系等模型也是如此。

其他的系统特定机制不在这里说明，例如上述输入去抖动和写入OR输出的选项。硬件可能支持以组的形式读取或写入GPIO，但通常与配置相关联：对于共享同一银行的GPIO。 （GPIO通常以16个或32个的银行分组，具有几个这样的银行的特定SOC。）某些系统可以从输出GPIO触发IRQ，或从未管理为GPIO的引脚读取值。依赖此类机制的代码必然是不可移植的。

动态定义GPIO目前不是标准的；例如，通过配置带有一些GPIO扩展器的附加板来实现。

### GPIO实现者的框架（可选）

如前所述，有一个可选的实现框架，使得平台能够使用相同的编程接口支持不同类型的GPIO控制器。该框架称为“gpiolib”。

作为调试辅助工具，如果可用debugfs，则可以在那里找到/sys/kernel/debug/gpio文件。那将列出通过此框架注册的所有控制器，以及当前使用的GPIO的状态。

#### 控制器驱动程序：gpio\_chip

在该框架中，每个GPIO控制器都打包为一个“struct gpio\_chip”，其中包含该类型的每个控制器的常见信息：

建立GPIO方向的方法

用于访问GPIO值的方法

标志说是否可以让其方法睡眠

可选的debugfs转储方法（显示额外状态，如上拉配置）

诊断标签

还有每个实例数据，可能来自device.platform\_data：其第一个GPIO的编号以及它公开的GPIO数量。

实现gpio\_chip的代码应支持控制器的多个实例，可能使用驱动程序模型。该代码将配置每个gpio\_chip并发出gpiochip\_add（）。删除GPIO控制器应该很少见；不可避免地使用gpiochip\_remove（）。

通常，gpio\_chip是实例特定结构的一部分，其状态不会通过GPIO接口公开，例如寻址、电源管理等等。像编解码器这样的芯片将具有复杂的非GPIO状态。

任何debugfs转储方法通常应忽略未被请求为GPIO的信号。它们可以使用gpiochip\_is\_requested（），它会在GPIO请求时返回NULL或与该GPIO相关联的标签。

#### 平台支持

为了强制启用此框架，平台的Kconfig将“选择”GPIOLIB，否则用户要自己配置GPIO的支持。

它还可以提供自定义值ARCH\_NR\_GPIO，以使其更好地反映该平台上实际使用的GPIO数量，而不浪费静态表空间。 （它应将内置/SoC GPIO和也反映在GPIO扩展器上。

如果没有选择这些选项，则平台不支持通过GPIO-lib进行GPIO，代码无法由用户启用。

这些函数的微不足道的实现可以直接使用框架代码，该代码始终通过gpio\_chip分派:

#define gpio\_get\_value \_\_gpio\_get\_value

#define gpio\_set\_value \_\_gpio\_set\_value

#define gpio\_cansleep \_\_gpio\_cansleep

更华丽的实现可以将这些定义为内联函数，以具有优化访问特定基于SOC的GPIO的逻辑。例如，如果引用的GPIO是常量“12”，则获取或设置其值的成本可能只有两到三个指令，永远不会睡眠。当这种优化不可能时，这些调用必须委托给框架代码，成本至少为几十个指令。对于位倒腾I / O，这样的指令节省可能非常显着。

对于SOC，平台特定的代码为每个芯片上GPIO的银行定义并注册gpio\_chip实例。这些GPIO应该按照芯片供应商的文档进行编号/标记，并直接匹配板子的原理图。它们可能从零开始，直到特定于平台的限制。这些GPIO通常被集成到平台初始化中，以使它们始终可用，从arch\_initcall（）或更早版本开始；它们经常充当IRQ。

#### Board 支持

对于外部 GPIO 控制器——例如 I2C 或 SPI 扩展器、ASIC、多功能设备、FPGA 或 CPLD——最常见的是特定于电路板的代码处理注册控制器设备，并确保它们的驱动程序知道要使用 gpiochip\_add() 的 GPIO 编号。它们的编号通常紧接在特定于平台的 GPIO 之后。

例如，板子设置代码可以创建结构体，确定芯片将公开的GPIO范围，并使用platform\_data将它们传递给每个GPIO扩展芯片。然后，芯片驱动程序的probe（）函数可以将那些数据传递给gpiochip\_add（）。

初始化顺序可能很重要。例如，当设备依赖于基于I2C的GPIO时，它的probe（）例程应该在那个GPIO可用后才调用。这可能意味着在那个GPIO可以工作之前不应该注册该设备。解决这种依赖关系的一种方法是，让这些gpio\_chip控制器提供设置（setup（））和卸载（teardown（））回调给板子特定的代码。这些板子特定的回调将在所有必要的资源可用时注册设备，并在GPIO控制器设备不可用时稍后删除它们。

### 用户空间的Sysfs接口（可选）

使用“gpiolib”实现者框架的平台可以选择配置sysfs用户界面以控制GPIO。这与debugfs接口不同，因为它提供了对GPIO方向和值的控制，而不仅仅是显示gpio状态摘要。此外，它可以在生产系统中存在而不需要调试支持。

根据系统的适当硬件文档，用户空间可以知道例如GPIO＃23控制用于保护闪存存储器中的引导加载程序段的写保护线路。系统升级程序可能需要临时删除该保护，首先导入GPIO，然后更改其输出状态，然后更新代码，然后重新启用写保护。在正常使用中，GPIO＃23将永远不会被触及，内核也不需要知道它。

再次取决于适当的硬件文档，在一些系统上，用户空间GPIO可以用于确定标准内核不知道的系统配置数据。并且对于某些任务，简单的用户空间GPIO驱动程序可能是系统实际所需的全部。

请注意，常见的“LED和按钮” GPIO任务有标准内核驱动程序：“leds-gpio”和“gpio\_keys”。使用它们而不是直接与GPIO通信；它们与内核框架集成得比您的用户空间代码好。

#### Sysfs中的路径

在/sys/class/gpio中有三种条目：

1. 控制接口：用于让用户空间控制GPIO
2. GPIO本身
3. GPIO控制器（“gpio\_chip”实例）

这些是“device”符号链接等标准文件的补充

控制接口是只写的:

/sys/class/gpio/

export”…用户空间可以通过将其数字写入此文件来请求内核将GPIO的控制权导出到用户空间。

示例：“echo 19> export”将为GPIO＃19创建一个“gpio19”节点，如果内核代码没有请求该节点。

“unexport”…撤销将GPIO导出到用户空间的效果。

示例：“echo 19> unexport”将删除使用“export”文件导出的“gpio19”节点。

GPIO信号的路径如/sys/class/gpio/gpio42/（对于GPIO＃42），并具有以下读/写属性:

/sys/class/gpio/gpioN/

“方向”…读取为“in”或“out”。通常可以写此值。将“out”写入会将该值初始化为低值。为确保无故障操作，值“低”和“高”可以写入以配置GPIO作为具有该初始值的输出。

请注意，如果内核不支持更改GPIO的方向，或者它是由内核代码导出的，没有明确允许用户空间重新配置该GPIO的方向，则不会存在此属性。

“值”…读取为0（低）或1（高）。如果将GPIO配置为输出，则可以写入此值；任何非零值都将被视为高电平。

如果可以将引脚配置为产生中断，则可以在该文件上进行poll（2），并且只要触发中断，poll（2）就会返回。如果使用poll（2），请将事件设置为POLLPRI。如果使用select（2），请将文件描述符设置为exceptfds。在poll（2）返回后，要么将lseek（2）指向sysfs文件的开头并读取新值，要么关闭该文件并重新打开它以读取该值。

“edge”…读取为“none”、“rising”、“falling”或“both”。将这些字符串写入以选择将在“value”文件上进行poll（2）返回的信号边缘。

该文件仅存在如果可以将引脚配置为产生中断输入引脚。

“active\_low”…读取为0（false）或1（true）。写入任何非零值以反转读取和写入的值属性。现有和随后的poll（2）支持配置通过“rising”和“falling”边缘的edge属性将遵循此设置。

GPIO控制器的路径如/sys/class/gpio/gpiochip42/（对于实现从＃42开始的GPIO的控制器），并具有以下只读属性:

/sys/class/gpio/gpiochipN/

“base”...与N相同，由该芯片管理的第一个GPIO

“label”...用于诊断（不总是唯一的）

“ngpio”...这个由该芯片管理的GPIO数量（从N到N + ngpio-1）

在大多数情况下，板子文档应该涵盖哪些GPIO用于什么目的。但是，这些编号并不总是稳定的；插在子卡上的GPIO可能会因使用的基板或堆叠中的其他卡而有所不同。在这种情况下，您可能需要使用gpiochip节点（可能结合框图）来确定要用于给定信号的正确GPIO编号。

#### 从内核代码导出

内核代码可以明确管理已经使用gpio\_request()请求的GPIO的导出:

/\* export the GPIO to userspace \*/

int gpio\_export(unsigned gpio, bool direction\_may\_change);

/\* reverse gpio\_export() \*/

void gpio\_unexport();

/\* create a sysfs link to an exported GPIO node \*/

int gpio\_export\_link(struct device \*dev, const char \*name, unsigned gpio)

内核驱动程序请求GPIO后，只能通过gpio\_export()将其在sysfs接口中提供。驱动程序可以控制信号方向是否可以更改。这有助于驱动程序防止用户空间代码无意中破坏重要系统状态。

这种显式的导出可以帮助调试（通过使某些类型的实验变得更容易），或者可以提供一个适合作为板支持包的一部分文档的始终可用的接口。

GPIO已经被导出后，gpio\_export\_link()允许从sysfs中的其他位置创建符号链接到GPIO sysfs节点。驱动程序可以使用这个来在sysfs中的自己的设备下提供具有描述性名称的接口。

### API参考

本节中列出的函数已弃用。应在新代码中使用基于GPIO描述符的API。

#### int gpio\_request\_one(unsigned gpio, unsigned long flags, const char \*label)

请求单个GPIO并进行初始配置

**参数**

unsigned gpio

GPIO编号

unsigned long flags

GPIO配置，由GPIOF\_\*指定

const char \*label

此GPIO的文字说明字符串

#### int gpio\_request\_array(const struct gpio \*array, size\_t num)

在单个调用中请求多个GPIO

**参数**

const struct gpio \*array

“struct gpio”的数组

size\_t num

数组中有多少个GPIO

#### void gpio\_free\_array(const struct gpio \*array, size\_t num)

在单个调用中释放多个GPIO

**参数**

const struct gpio \*array

“struct gpio”的数组

size\_t num

数组中有多少个GPIO

## 核心

### struct gpio\_irq\_chip

#### GPIO中断控制器定义

struct gpio\_irq\_chip {

struct irq\_chip \*chip;

struct irq\_domain \*domain;

const struct irq\_domain\_ops \*domain\_ops;

irq\_flow\_handler\_t handler;

unsigned int default\_type;

struct lock\_class\_key \*lock\_key;

struct lock\_class\_key \*request\_key;

irq\_flow\_handler\_t parent\_handler;

void \*parent\_handler\_data;

unsigned int num\_parents;

unsigned int parent\_irq;

unsigned int \*parents;

unsigned int \*map;

bool threaded;

bool need\_valid\_mask;

unsigned long \*valid\_mask;

unsigned int first;

};

#### 成员

chip

GPIO IRQ芯片实现，由GPIO驱动程序提供

domain

中断转换域；负责在GPIO hwirq数字和Linux IRQ数字之间进行映射

domain\_ops

为此IRQ芯片提供的中断域操作表

handler

要使用的IRQ处理程序（通常是预定义的IRQ内核函数），由GPIO驱动程序提供

default\_type

在GPIO驱动程序初始化期间应用的默认IRQ触发类型，由GPIO驱动程序提供

lock\_key

每个GPIO IRQ芯片lockdep类。

parent\_handler

GPIO芯片的父中断的中断处理程序，如果父中断是嵌套而不是级联，则可以为空。

parent\_handler\_data

与父中断的处理程序相关联并传递给它的数据。

num\_parents

GPIO芯片的中断父级的数量。

parent\_irq

供gpiochip\_set\_cascaded\_irqchip()使用

parents

一个GPIO芯片的中断父级列表。它归驱动程序所有，因此核心仅引用此列表，而不修改它

map

每个GPIO芯片的每一行的中断父级的列表

threaded

如果设置了中断处理，则返回True使用嵌套线程

need\_valid\_mask

如果设置，core将使用完全设置为1的valid\_mask进行分配

valid\_mask

如果不为NULL，则保持可包含在芯片的IRQ域中的GPIO的位掩码

first

对于静态IRQ分配是必需的。如果设置，irq\_domain\_add\_simple()将在初始化期间分配和映射所有IRQ

### struct gpio\_chip

抽象一个GPIO控制器

#### 定义

struct gpio\_chip {

const char \*label;

struct gpio\_device \*gpiodev;

struct device \*parent;

struct module \*owner;

int (\*request)(struct gpio\_chip \*chip, unsigned offset);

void (\*free)(struct gpio\_chip \*chip, unsigned offset);

int (\*get\_direction)(struct gpio\_chip \*chip, unsigned offset);

int (\*direction\_input)(struct gpio\_chip \*chip, unsigned offset);

int (\*direction\_output)(struct gpio\_chip \*chip, unsigned offset, int value);

int (\*get)(struct gpio\_chip \*chip, unsigned offset);

int (\*get\_multiple)(struct gpio\_chip \*chip,unsigned long \*mask, unsigned long \*bits);

void (\*set)(struct gpio\_chip \*chip, unsigned offset, int value);

void (\*set\_multiple)(struct gpio\_chip \*chip,unsigned long \*mask, unsigned long \*bits);

int (\*set\_config)(struct gpio\_chip \*chip,unsigned offset, unsigned long config);

int (\*to\_irq)(struct gpio\_chip \*chip, unsigned offset);

void (\*dbg\_show)(struct seq\_file \*s, struct gpio\_chip \*chip);

int base;

u16 ngpio;

const char \*const \*names;

bool can\_sleep;

#if IS\_ENABLED(CONFIG\_GPIO\_GENERIC);

unsigned long (\*read\_reg)(void \_\_iomem \*reg);

void (\*write\_reg)(void \_\_iomem \*reg, unsigned long data);

bool be\_bits;

void \_\_iomem \*reg\_dat;

void \_\_iomem \*reg\_set;

void \_\_iomem \*reg\_clr;

void \_\_iomem \*reg\_dir;

bool bgpio\_dir\_inverted;

int bgpio\_bits;

spinlock\_t bgpio\_lock;

unsigned long bgpio\_data;

unsigned long bgpio\_dir;

#endif;

#ifdef CONFIG\_GPIOLIB\_IRQCHIP;

struct gpio\_irq\_chip irq;

#endif;

bool need\_valid\_mask;

unsigned long \*valid\_mask;

#if defined(CONFIG\_OF\_GPIO);

struct device\_node \*of\_node;

unsigned int of\_gpio\_n\_cells;

int (\*of\_xlate)(struct gpio\_chip \*gc, const struct of\_phandle\_args \*gpiospec, u32 \*flags);

#endif;

};

#### 成员

label

GPIO 设备的功能名称，例如零件号或实现它的 SoC IP-block 的名称。

gpiodev

内部状态持有者，不透明结构

parent

可选的提供 GPIO 的上级设备

owner

有助于防止导出活动 GPIO 的模块被移除。

request

可选的钩子用于芯片特定的激活，例如启用模块电源和时钟；可能会睡眠。

free

可选的钩子用于芯片特定的停用，例如禁用模块电源和时钟；可能会睡眠。

get\_direction

返回信号“偏移”的方向，0=输出，1=输入（与 GPIOF\_DIR\_XXX 相同），或负错误。

direction\_input

将信号“偏移”配置为输入，或返回错误。

direction\_output

将信号“偏移”配置为输出，或返回错误。

get

返回信号“偏移”的值，0=低，1=高，或负错误。

get\_multiple

读取由“mask”定义的多个信号的值并将它们存储在“bits”中，在成功时返回0或负错误。

set

为信号“偏移”分配输出值

set\_multiple

为由“mask”定义的多个信号分配输出值

set\_config

所有类型的设置的可选钩子。使用与通用 pinconf 相同的打包配置格式。

to\_irq

支持非静态 gpio\_to\_irq() 映射的可选钩子；实现可能睡眠。

dbg\_show

可选的调试文件系统中显示内容的例程；当省略时将使用默认代码，但自定义代码可以显示额外状态（例如上拉/下拉配置）。

base

标识此芯片处理的第一个 GPIO 编号；或者，在注册期间为负时，请求动态 ID 分配。已弃用：提供任何非负值并钉住 GPIO 芯片的基本偏移量已被弃用。请将 -1 传递为基准，以便 gpiolib 在所有可能的情况下选择芯片基准。我们希望长远来说摆脱静态 GPIO 编号空间。

ngpio

此控制器处理的 GPIO 数量；最后一个受理的 GPIO 是（base + ngpio - 1）。

names

如果设置，必须是字符串数组，用作此芯片中 GPIO 的替代名称。数组中的任何条目都可以为空，如果 GPIO 没有别名，则数组必须具有 ngpio 个条目。名称可以包括一个无符号整数的单个 printk 格式说明符。它由 gpio 的实际编号替换。

can\_sleep

当通过 I2C 或 SPI 访问 GPIO 扩展器芯片时，必须设置 flag，以便 get()/set()方法睡眠。这意味着，如果芯片支持 IRQ，这些 IRQ 需要线程化，因为当读取 IRQ 状态寄存器时，芯片访问可能会睡眠。

read\_reg

通用 GPIO 的 reader 函数

write\_reg

通用 GPIO 的 writer 函数

be\_bits

如果通用 GPIO 具有大端位顺序（位 31 表示线 0，位 30 表示线 1 ... 位 0 表示线 31），则通过通用 GPIO 核心将其设置为 true。它仅用于内部事务处理。

reg\_dat

通用 GPIO 的数据（输入）寄存器

reg\_set

通用 GPIO 的输出设置寄存器（out=高）

reg\_clr

通用 GPIO 的输出清除寄存器（out=低）

reg\_dir

通用 GPIO 的方向设置寄存器

bgpio\_dir\_inverted

指示方向寄存器是否反转（gpiolib 私有状态变量）。

bgpio\_bits

用于通用 GPIO 的注册位数，即 <寄存器宽度> \* 8

bgpio\_lock

用于锁定 chip->bgpio\_data。此外，这需要将影子和实际数据寄存器写在一起。

bgpio\_data

通用GPIO的影子数据寄存器，以安全地清除/设置位。

bgpio\_dir

通用GPIO的影子方向寄存器，以安全地清除/设置方向。

irq

将中断芯片功能与 GPIO 芯片集成。可用于处理大多数实际情况的 IRQ。

need\_valid\_mask

如果 core 分配了所有位设置为 1 的 valid\_mask。

valid\_mask

如果不为 NULL，则持有可从芯片使用的 GPIO 的位掩码。

of\_node

指向表示此 GPIO 控制器的设备树节点的指针。

of\_gpio\_n\_cells

用于构成GPIO specifier的单元格数。

of\_xlate

将设备树GPIO specifier翻译成芯片相关的GPIO编号和标志的回调函数

#### 说明

gpio\_chip可以帮助平台将各种GPIO源抽象化，以便它们可以通过一个通用的编程接口进行访问。示例源包括SOC控制器、FPGA、多功能芯片、专用GPIO扩展器等等。

每个芯片控制一些信号，在方法调用中由范围0..(ngpio-1)中的"offset"值标识。当这些信号通过调用gpio\_get\_value(gpio)进行引用时，通过从gpio编号中减去基数来计算偏移量。

### gpiochip\_add\_data

gpiochip\_add\_data (chip, data)

注册 gpio\_chip

#### 参数

chip

待注册的芯片，已初始化chip->base。

data

与该芯片相关联的驱动程序专用数据

#### 上下文

潜在地在irqs之前工作

#### 说明

当在引导期间非常早期调用gpiochip\_add\_data()使得GPIO可以自由使用时，必须在gpio框架的arch\_initcall()之前注册gc->parent设备。否则，GPIO的sysfs初始化将会失效。

gpiochip\_add\_data()只能在gpiolib初始化后才能调用，即在core\_initcall()之后。

如果gc->base是负数，则请求动态分配一系列有效的GPIO。

#### 返回

如果无法注册芯片，则返回负的errno，例如因为gc->base无效或已与不同的芯片关联。否则它将返回成功代码零。

### struct gpio\_pin\_range

由GPIO芯片控制的引脚范围

#### 定义

struct gpio\_pin\_range {

struct list\_head node;

struct pinctrl\_dev \*pctldev;

struct pinctrl\_gpio\_range range;

};

#### 成员

node

用于维护一组引脚范围的列表，仅在内部使用

pctldev

处理相应引脚的pinctrl设备

range

由GPIO控制器控制的实际引脚范围

### struct gpio\_desc \*gpio\_to\_desc(unsigned gpio)

将GPIO编号转换为其描述符

#### 参数

unsigned gpio

全局GPIO编号

#### 返回

与给定GPIO相关联的GPIO描述符；如果没有系统中具有给定编号的GPIO，则返回NULL。

### int desc\_to\_gpio(const struct gpio\_desc \*desc)

将GPIO描述符转换为整数命名空间

#### 参数

const struct gpio\_desc \*desc

GPIO描述符

#### 说明

未来应该会消失，但由于我们仍然为错误消息和sysfs节点使用GPIO编号，因此仍然需要它。

#### 返回

由其描述符指定的GPIO的全局编号。

### struct [gpio\_chip](https://www.kernel.org/doc/html/latest/driver-api/gpio/index.html" \l "c.gpio_chip" \o "gpio_chip) \*gpiod\_to\_chip(const struct gpio\_desc \*desc)

返回GPIO描述符所属的GPIO芯片

#### 参数

const struct gpio\_desc \*desc

返回芯片的描述符

### int gpiod\_get\_direction(struct gpio\_desc \*desc)

返回GPIO的当前方向

#### 参数

struct gpio\_desc \*desc

GPIO获取方向

#### 说明

返回输出的0，输入的1，或者在出现错误的情况下返回错误代码。

如果gpiod\_cansleep()为真，则此函数可能会休眠。

### void \*gpiochip\_get\_data(struct [gpio\_chip](https://www.kernel.org/doc/html/latest/driver-api/gpio/index.html" \l "c.gpio_chip" \o "gpio_chip) \*chip)

获取芯片的每个子驱动程序数据

#### 参数

struct gpio\_chip \*gc

GPIO 芯片

#### 返回

芯片的每个子驱动程序数据

### void gpiochip\_remove(struct [gpio\_chip](https://www.kernel.org/doc/html/latest/driver-api/gpio/index.html" \l "c.gpio_chip" \o "gpio_chip) \*chip)

注销gpio\_chip

#### 参数

struct gpio\_chip \*gc

要注销的芯片

#### 说明

仍然请求任何GPIO的gpio\_chip可能无法移除

### int devm\_gpiochip\_add\_data(struct [device](https://www.kernel.org/doc/html/v4.19/driver-api/infrastructure.html" \l "c.device" \o "device) \* dev, struct [gpio\_chip](https://www.kernel.org/doc/html/v4.19/driver-api/gpio/index.html" \l "c.gpio_chip" \o "gpio_chip) \* chip, void \* data)

资源管理器 [gpiochip\_add\_data()](https://www.kernel.org/doc/html/v4.19/driver-api/gpio/index.html" \l "c.gpiochip_add_data" \o "gpiochip_add_data)

#### 参数

struct device \* dev

irq\_chip所属的设备指针。

struct gpio\_chip \*chip

待注册的芯片，已初始化chip->base。

void \*data

与该芯片相关联的驱动程序专用数据。

#### 上下文

潜在地在irqs之前工作

#### 说明

当设备未绑定时，gpio芯片会自动释放。

#### 返回

如果无法注册芯片，则返回负的errno，例如因为chip->base无效或已与不同的芯片关联。否则它将返回成功代码零。

### void devm\_gpiochip\_remove(struct [device](https://www.kernel.org/doc/html/v4.19/driver-api/infrastructure.html" \l "c.device" \o "device) \* dev, struct [gpio\_chip](https://www.kernel.org/doc/html/v4.19/driver-api/gpio/index.html" \l "c.gpio_chip" \o "gpio_chip) \* chip)

[gpiochip\_remove()](https://www.kernel.org/doc/html/v4.19/driver-api/gpio/index.html" \l "c.gpiochip_remove" \o "gpiochip_remove)的资源管理器

#### 参数

struct device \* dev

为其分配资源的设备。

struct gpio\_chip \*chip

要删除的芯片

#### 说明

仍然请求任何GPIO的gpio\_chip可能无法移除。

### struct [gpio\_chip](https://www.kernel.org/doc/html/latest/driver-api/gpio/index.html" \l "c.gpio_chip" \o "gpio_chip) \*gpiochip\_find(void \*data, int (\*match)(struct [gpio\_chip](https://www.kernel.org/doc/html/latest/driver-api/gpio/index.html" \l "c.gpio_chip" \o "gpio_chip) \*chip, void \*data))

iterator for locating a specific gpio\_chip

#### 参数

void \*data

传递给匹配函数的数据

int (\*match)(struct gpio\_chip \*chip, void \*data)

检查gpio\_chip的回调函数

#### 说明

类似于bus\_find\_device。它返回用户提供的匹配回调确定的gpio\_chip的引用。如果回调返回0，则设备不匹配，如果回调返回非零，则此函数将返回给调用者，并且不会遍历更多的gpio\_chip。

### void gpiochip\_set\_chained\_irqchip(struct [gpio\_chip](https://www.kernel.org/doc/html/v4.19/driver-api/gpio/index.html" \l "c.gpio_chip" \o "gpio_chip) \* gpiochip, struct [irq\_chip](https://www.kernel.org/doc/html/v4.19/core-api/genericirq.html" \l "c.irq_chip" \o "irq_chip) \* irqchip, unsigned int parent\_irq, irq\_flow\_handler\_t parent\_handler)

将链接的irqchip连接到gpiochip

#### 参数

struct gpio\_chip \*gpiochip

要设置irqchip链的gpiochip

struct irq\_chip \* irqchip

连接到gpiochip的中断芯片

unsigned int parent\_irq

相应于级联irqchip的父级IRQ的中断号

irq\_flow\_handler\_t parent\_handler

来自gpiochip的累积IRQ的父中断处理程序。如果中断是嵌套的而不是级联的，则在此处理程序参数中传递NULL

### void gpiochip\_set\_nested\_irqchip(struct [gpio\_chip](https://www.kernel.org/doc/html/v4.19/driver-api/gpio/index.html" \l "c.gpio_chip" \o "gpio_chip) \* gpiochip, struct [irq\_chip](https://www.kernel.org/doc/html/v4.19/core-api/genericirq.html" \l "c.irq_chip" \o "irq_chip) \* irqchip, unsigned int parent\_irq)

将嵌套irqchip连接到gpiochip

#### 参数

struct gpio\_chip \* gpiochip

要将irqchip嵌套处理程序设置为的gpiochip

struct irq\_chip \* irqchip

要嵌套到gpiochip的irqchip

unsigned int parent\_irq

相应于此嵌套irqchip的父级IRQ的中断号

### int gpiochip\_irq\_map(struct irq\_domain \*d, unsigned int irq, irq\_hw\_number\_t hwirq)

将IRQ映射到GPIO irqchip

#### 参数

struct irq\_domain \*d

此irqchip使用的irqdomain

unsigned int irq

该GPIO irqchip IRQ使用的全局irq号

irq\_hw\_number\_t hwirq

本gpiochip上的本地IRQ / GPIO行偏移量

#### 说明

此函数将通过将gpiochip分配为芯片数据并使用存储在gpiochip内部的irqchip来设置gpiochip上某个IRQ线的映射。

### int gpiochip\_irqchip\_add\_key(struct [gpio\_chip](https://www.kernel.org/doc/html/v4.19/driver-api/gpio/index.html" \l "c.gpio_chip" \o "gpio_chip) \* gpiochip, struct [irq\_chip](https://www.kernel.org/doc/html/v4.19/core-api/genericirq.html" \l "c.irq_chip" \o "irq_chip) \* irqchip, unsigned int first\_irq, irq\_flow\_handler\_t handler, unsigned int type, bool threaded, struct lock\_class\_key \* lock\_key, struct lock\_class\_key \* request\_key)

将irqchip添加到gpiochip

#### 参数

struct gpio\_chip \* gpiochip

要向其添加irqchip的gpiochip

struct irq\_chip \* irqchip

要添加到gpiochip的irqchip

unsigned int first\_irq

如果未动态分配，则分配gpiochip irqs的基础（第一个）IRQ

irq\_flow\_handler\_t handler

要使用的irq处理程序（通常是预定义的irq核心函数）

unsigned int type

用于此irqchip上的IRQ的默认类型，传递IRQ\_TYPE\_NONE以使核心避免在硬件中设置任何默认类型。

bool threaded

此irqchip是否使用嵌套线程处理程序

struct lock\_class\_key \* lock\_key

IRQ锁的lockdep类

struct lock\_class\_key \* request\_key

IRQ请求的lockdep类

#### 说明

此函数将某个irqchip密切关联到某个gpiochip，提供irq域以将本地IRQ转换为gpiolib核心中的全局IRQ，并确保将gpiochip传递为与所有相关函数相关的芯片数据。驱动程序回调需要使用gpiochip\_get\_data（）从传递为芯片数据的gpiochip中获取其本地状态容器。 gpiochip中将存储一个irqdomain，该irqdomain将由驱动程序用于处理IRQ编号转换。在调用此函数之前，需要初始化和注册gpiochip。

此函数将处理两个单元格：简单IRQs，并假定gpiochip上的所有引脚都可以生成唯一的IRQ。其他一切需要进行开放编码。

### int gpiochip\_generic\_request(struct [gpio\_chip](https://www.kernel.org/doc/html/latest/driver-api/gpio/index.html" \l "c.gpio_chip" \o "gpio_chip) \*chip, unsigned int offset)

请求引脚的gpio功能

#### 参数

struct gpio\_chip \*chip

拥有GPIO的gpiochip

unsigned int offset

要请求GPIO功能的GPIO的偏移量

### void gpiochip\_generic\_free(struct [gpio\_chip](https://www.kernel.org/doc/html/latest/driver-api/gpio/index.html" \l "c.gpio_chip" \o "gpio_chip) \*chip, unsigned int offset)

从引脚释放gpio功能

#### 参数

struct gpio\_chip \*chip

要请求的gpio功能的gpiochip

unsigned int offset

要从GPIO功能中释放的GPIO的偏移量

### int gpiochip\_generic\_config(struct [gpio\_chip](https://www.kernel.org/doc/html/latest/driver-api/gpio/index.html" \l "c.gpio_chip" \o "gpio_chip) \*chip, unsigned int offset, unsigned long config)

应用引脚的配置

#### 参数

struct gpio\_chip \*chip

拥有GPIO的gpiochip

unsigned int offset

要应用配置的GPIO的偏移量

unsigned long config

要应用的配置

### int gpiochip\_add\_pingroup\_range(struct [gpio\_chip](https://www.kernel.org/doc/html/latest/driver-api/gpio/index.html" \l "c.gpio_chip" \o "gpio_chip) \*chip, struct pinctrl\_dev \*pctldev, unsigned int gpio\_offset, const char \*pin\_group)

为GPIO <->引脚映射添加范围

#### 参数

struct gpio\_chip \*chip

要添加范围的gpiochip

struct pinctrl\_dev \*pctldev

要映射到的引脚控制器

unsigned int gpio\_offset

在当前gpio\_chip编号空间中的起始偏移量

const char \*pin\_group

引脚控制器内部针脚组的名称

#### 说明

从支持设备树的pinctrl驱动程序直接调用此函数是不推荐的。请参阅Documentation / devicetree / bindings / gpio / gpio.txt第2.1节，了解如何通过“ gpio-ranges”属性将pinctrl和gpio驱动程序绑定。

### int gpiochip\_add\_pin\_range(struct [gpio\_chip](https://www.kernel.org/doc/html/latest/driver-api/gpio/index.html" \l "c.gpio_chip" \o "gpio_chip) \*chip, const char \*pinctl\_name, unsigned int gpio\_offset, unsigned int pin\_offset, unsigned int npins)

为GPIO <->引脚映射添加范围

#### 参数

struct gpio\_chip \*chip

要添加范围的gpiochip

const char \*pinctl\_name

要映射到的引脚控制器的dev\_name（）

unsigned int gpio\_offset

在当前gpio\_chip编号空间中的起始偏移量

unsigned int pin\_offset

引脚控制器编号空间中的起始偏移量

unsigned int npins

每个引脚空间偏移量（GPIO和引脚控制器）的引脚数目累积到该范围中

#### 返回

成功时返回0，失败时返回负错误代码。

直接从支持设备树的pinctrl驱动程序调用此函数是不推荐的。请参见Documentation/devicetree/bindings/gpio/gpio.txt的第2.1节，了解如何通过“gpio-ranges”属性绑定pinctrl和gpio驱动程序。

### void gpiochip\_remove\_pin\_ranges(struct [gpio\_chip](https://www.kernel.org/doc/html/latest/driver-api/gpio/index.html" \l "c.gpio_chip" \o "gpio_chip) \*chip)

删除所有GPIO<->引脚映射。

#### 参数

struct gpio\_chip \*chip

要删除所有映射的芯片

### const char \*gpiochip\_is\_requested(struct [gpio\_chip](https://www.kernel.org/doc/html/latest/driver-api/gpio/index.html" \l "c.gpio_chip" \o "gpio_chip) \*chip, unsigned int offset)

仅返回字符串，如果信号已被请求

#### 参数

struct gpio\_chip \*chip

控制信号的控制器

unsigned int offset

控制器中信号的范围为0..（ngpio - 1）

#### 说明

如果GPIO当前未被请求，则返回NULL，否则返回一个字符串。返回的字符串是传递给gpio\_request（）的标签；如果没有传递标签，则它是一个无意义的非空常量。

此函数供GPIO控制器驱动程序使用。标签可以帮助诊断，并且知道信号被用作GPIO可以帮助避免意外地将其多路复用到另一个控制器。

### struct gpio\_desc \*gpiochip\_request\_own\_desc(struct [gpio\_chip](https://www.kernel.org/doc/html/latest/driver-api/gpio/index.html" \l "c.gpio_chip" \o "gpio_chip) \*chip, u16 hwnum, const char \*label)

允许GPIO芯片请求其自己的说明符

#### 参数

struct gpio\_chip \*chip

GPIO芯片

unsigned int hwnum

要请求说明符的GPIO的硬件编号

const char \*label

GPIO的标签

#### 说明

函数允许GPIO芯片驱动程序通过gpiolib API请求和使用其自己的GPIO说明符。与gpiod\_request（）的区别在于，此函数不会增加GPIO芯片模块的引用计数。这使得可以根据需要卸载GPIO芯片模块（我们假设GPIO芯片驱动程序处理了其已请求的GPIO的释放）。

#### 返回

指向GPIO说明符的指针，或者编码为ERR\_PTR（）的负错误代码。

### void gpiochip\_free\_own\_desc(struct gpio\_desc \*desc)

释放芯片驱动程序请求的GPIO

#### 参数

struct gpio\_desc \*desc

要释放的GPIO说明符

#### 说明

函数释放先前使用gpiochip\_request\_own\_desc（）请求的给定GPIO。

### int gpiod\_direction\_input(struct gpio\_desc \*desc)

释放芯片驱动程序请求的GPIO

#### 参数

struct gpio\_desc \*desc

要释放的GPIO说明符

#### 说明

函数释放先前使用gpiochip\_request\_own\_desc（）请求的给定GPIO。

### int gpiod\_direction\_output\_raw(struct gpio\_desc \*desc, int value)

将GPIO方向设置为输出

#### 参数

struct gpio\_desc \*desc

要设置为输出的GPIO

int value

GPIO的初始输出值

#### 说明

将传递的GPIO方向设置为输出，例如可以安全地在其上调用gpiod\_set\_value（）。必须指定输出的初始值作为物理线上的原始值，而不考虑ACTIVE\_LOW状态。

返回0表示成功，否则表示错误代码。

### int gpiod\_direction\_output(struct gpio\_desc \*desc, int value)

将GPIO方向设置为输出

#### 参数

struct gpio\_desc \*desc

要设置为输出的GPIO

int value

GPIO的初始输出值

#### 说明

将传递的GPIO方向设置为输出，例如可以安全地在其上调用gpiod\_set\_value（）。必须将输出的初始值指定为GPIO的逻辑值，即考虑其ACTIVE\_LOW状态。

返回0表示成功，否则表示错误代码。

### int gpiod\_set\_debounce(struct gpio\_desc \*desc, unsigned int debounce)

设置GPIO的抖动时间

#### 参数

struct gpio\_desc \*desc

要设置抖动时间的GPIO的说明符

unsigned int debounce

微秒的抖动时间

#### 返回

在成功时返回0，在控制器不支持设置抖动时间时返回-ENOTSUPP。

### int gpiod\_set\_transitory(struct gpio\_desc \*desc, bool transitory)

在挂起或重置时丢失或保留GPIO状态

#### 参数

struct gpio\_desc \*desc

要为其配置持久性的GPIO的说明符

bool transitory

True表示在挂起或重置时丢失状态，false表示保持不变

#### 返回

在成功时返回0，否则返回负错误代码。

### int gpiod\_is\_active\_low(const struct gpio\_desc \*desc)

测试GPIO是否为低电平活动

#### 参数

const struct gpio\_desc \*desc

要测试的gpio说明符

#### 说明

如果GPIO为低电平活动，则返回1，否则返回0。

### int gpiod\_get\_raw\_value(const struct gpio\_desc \*desc)

返回GPIO的原始值

#### 参数

const struct gpio\_desc \*desc

要返回其值的gpio

#### 说明

返回GPIO的原始值，即忽略其ACTIVE\_LOW状态的物理线的值，或者返回负的errno表示失败。

此函数可以从我们无法睡眠的上下文中调用，并且如果GPIO芯片函数可能睡眠，它将发出投诉。

### int gpiod\_get\_value(const struct gpio\_desc \*desc)

返回GPIO的值

#### 参数

const struct gpio\_desc \*desc

要返回其值的gpio

#### 说明

返回GPIO的逻辑值，即考虑ACTIVE\_LOW状态，否则返回负的errno表示失败。

此功能可以从无法睡眠的上下文中调用，并且如果GPIO芯片函数可能睡眠，则会发出投诉。

### int gpiod\_get\_raw\_array\_value(unsigned int array\_size, struct gpio\_desc \*\*desc\_array, int \*value\_array)

从一组GPIO中读取原始值

#### 参数

unsigned int array\_size

说明符数组/值位图中的元素数量

struct gpio\_desc \*\* desc\_array

要读取值的GPIO说明符数组

int \* value\_array

用于存储读取值的数组

#### 说明

读取GPIO的原始值，即物理行的值，不考虑其ACTIVE\_LOW状态。成功时返回0，否则返回错误代码。

此功能可以从无法睡眠的上下文中调用，如果GPIO芯片函数可能睡眠，则会发出投诉。

### int gpiod\_get\_array\_value(unsigned int array\_size, struct gpio\_desc \*\*desc\_array, int \*value\_array)

从一组GPIO中读取值

#### 参数

unsigned int array\_size

说明符数组/值位图中的元素数量

struct gpio\_desc \*\* desc\_array

要读取值的GPIO说明符数组

int \* value\_array

用于存储读取值的数组

#### 说明

读取GPIO的逻辑值，即考虑其ACTIVE\_LOW状态。成功时返回0，否则返回错误代码。

此功能可以从无法睡眠的上下文中调用，如果GPIO芯片函数可能睡眠，则会发出投诉。

### void gpiod\_set\_raw\_value(struct gpio\_desc \*desc, int value)

分配gpio的原始值

#### 参数

struct gpio\_desc \* desc

要分配值的gpio

int value

要分配的值

#### 说明

设置GPIO的原始值，即其物理线的值，而不考虑其ACTIVE\_LOW状态。

此功能可以从无法睡眠的上下文中调用，并且如果GPIO芯片函数可能睡眠，则会发出投诉。

### void gpiod\_set\_value（struct gpio\_desc \* desc，int value）

分配gpio的值

#### 参数

struct gpio\_desc \* desc

要分配值的gpio

int value

要分配的值

#### 说明

设置GPIO的逻辑值，即考虑其ACTIVE\_LOW，OPEN\_DRAIN和OPEN\_SOURCE标志。

此功能可以从无法睡眠的上下文中调用，并且如果GPIO芯片函数可能睡眠，则会发出投诉。

### int gpiod\_set\_raw\_array\_value（unsigned int array\_size，struct gpio\_desc \*\* desc\_array，int \* value\_array）

将值分配给一组GPIO

#### 参数

unsigned int array\_size

说明符数组/值位图中的元素数量

struct gpio\_desc \*\* desc\_array

要分配值的GPIO说明符数组

int \* value\_array

用于存储读取值的数组

#### 说明

设置GPIO的原始值，即物理线的值，而不考虑其ACTIVE\_LOW状态。

此功能可以从无法睡眠的上下文中调用，并且如果GPIO芯片函数可能睡眠，则会发出投诉。

### void gpiod\_set\_array\_value（unsigned int array\_size，struct gpio\_desc \*\* desc\_array，int \* value\_array）

将值分配给一组GPIO

#### 参数

unsigned int array\_size

说明符数组/值位图中的元素数量

struct gpio\_desc \*\* desc\_array

要分配值的GPIO说明符数组

int \* value\_array

用于存储读取值的数组

#### 说明

设置GPIO的逻辑值，即考虑其ACTIVE\_LOW状态。成功时返回0，否则返回错误代码。

此功能可以从无法睡眠的上下文中调用，如果GPIO芯片函数可能睡眠，则会发出投诉。

### int gpiod\_cansleep（const struct gpio\_desc \* desc）

报告gpio值访问可能会睡眠

#### 参数

const struct gpio\_desc \* desc

要检查的gpio

### void gpiod\_set\_consumer\_name（struct gpio\_desc \* desc，const char \* name）

为说明符设置消费者名称

#### 参数

struct gpio\_desc \* desc

要在其上设置消费者名称的gpio

const char \* name

新的消费者名称

### int gpiod\_to\_irq（const struct gpio\_desc \* desc）

返回与GPIO对应的IRQ

#### 参数

const struct gpio\_desc \* desc

要返回其IRQ的gpio（已请求）

#### 说明

返回与传递的GPIO对应的IRQ，或在出现错误时返回错误代码。

### int gpiochip\_lock\_as\_irq（struct gpio\_chip \* chip，unsigned int offset）

将GPIO锁定为用于IRQ

#### 参数

struct gpio\_chip \* chip

要锁定GPIO所属的芯片

unsigned int offset

要锁定为IRQ的GPIO的偏移量

#### 说明

这是由希望将某个GPIO行锁定为IRQ的GPIO驱动程序直接使用的。

### void gpiochip\_unlock\_as\_irq（struct gpio\_chip \* chip，unsigned int offset）

解锁用作IRQ的GPIO

#### 参数

struct gpio\_chip \* gc

要锁定GPIO所属的芯片

unsigned int offset

要锁定为IRQ的GPIO的偏移量

#### 说明

这是由希望指示某个GPIO不再仅用于IRQ的GPIO驱动程序直接使用的。

### int gpiod\_get\_raw\_value\_cansleep（const struct gpio\_desc \* desc）

返回GPIO的原始值

#### 参数

const struct gpio\_desc \*desc

要返回其值的GPIO

说明

返回GPIO的原始值，即物理线路的值而不考虑其ACTIVE\_LOW状态，或失败的负错误代码。

此函数应从可以睡眠的上下文中调用。

### int gpiod\_get\_value\_cansleep（const struct gpio\_desc \*desc）

返回GPIO的值

#### 参数

const struct gpio\_desc \*desc

要返回其值的GPIO

#### 说明

返回GPIO的逻辑值，即考虑ACTIVE\_LOW状态，或失败的负错误代码。

此函数应从可以睡眠的上下文中调用。

### int gpiod\_get\_raw\_array\_value\_cansleep（unsigned int array\_size，struct gpio\_desc \*\* desc\_array，int \* value\_array）

从GPIO数组中读取原始值

#### 参数

unsigned int array\_size

说明符数组/值位图中的元素数量

struct gpio\_desc \*\* desc\_array

要读取其值的GPIO说明符数组

int \* value\_array

用于存储读取值的数组

#### 说明

读取GPIO的原始值，即物理线路的值，而不考虑其ACTIVE\_LOW状态。成功返回0，否则返回错误代码。

此函数应从可以睡眠的上下文中调用。

### int gpiod\_get\_array\_value\_cansleep（unsigned int array\_size，struct gpio\_desc \*\* desc\_array，int \* value\_array）

从GPIO数组中读取值

#### 参数

unsigned int array\_size

说明符数组/值位图中的元素数量

struct gpio\_desc \*\* desc\_array

要读取其值的GPIO说明符数组

int \* value\_array

用于存储读取值的数组

#### 说明

读取GPIO的逻辑值，即考虑其ACTIVE\_LOW状态。成功返回0，否则返回错误代码。

此函数应从可以睡眠的上下文中调用。

### void gpiod\_set\_raw\_value\_cansleep（struct gpio\_desc \* desc，int value）

分配GPIO的原始值

#### 参数

struct gpio\_desc \* desc

要分配其值的GPIO

int value

要分配的值

#### 说明

设置GPIO的原始值，即其物理线路的值，而不考虑其ACTIVE\_LOW状态。

此函数应从可以睡眠的上下文中调用。

### void gpiod\_set\_value\_cansleep（struct gpio\_desc \* desc，int value）

分配GPIO的值

#### 参数

struct gpio\_desc \* desc

要分配其值的GPIO

int value

要分配的值

#### 说明

设置GPIO的逻辑值，即考虑其ACTIVE\_LOW状态。

此函数应从可以睡眠的上下文中调用。

### int gpiod\_set\_raw\_array\_value\_cansleep（unsigned int array\_size，struct gpio\_desc \*\* desc\_array，int \* value\_array）

分配数组的值到GPIO

#### 参数

unsigned int array\_size

说明符数组/值位图中的元素数量

struct gpio\_desc \*\* desc\_array

要分配其值的GPIO说明符数组

int \* value\_array

用于存储读取值的数组

#### 说明

设置GPIO的原始值，即其物理线路的值，而不考虑其ACTIVE\_LOW状态。

此函数应从可以睡眠的上下文中调用。

### int gpiod\_set\_array\_value\_cansleep（unsigned int array\_size，struct gpio\_desc \*\* desc\_array，int \* value\_array）

分配数组的值到GPIO

#### 参数

unsigned int array\_size

说明符数组/值位图中的元素数量

struct gpio\_desc \*\* desc\_array

要分配其值的GPIO说明符数组

int \* value\_array

用于存储读取值的数组

#### 说明

设置GPIO的逻辑值，即考虑其ACTIVE\_LOW状态。

此函数应从可以睡眠的上下文中调用。

### void gpiod\_add\_lookup\_table（struct gpiod\_lookup\_table \*表）

注册GPIO设备消费者

#### 参数

struct gpiod\_lookup\_table \* table

要注册的消费者表

### void gpiod\_remove\_lookup\_table（struct gpiod\_lookup\_table \*表）

注销GPIO设备消费者

#### 参数

struct gpiod\_lookup\_table \* table

要注销的消费者表

### void gpiod\_add\_hogs（struct gpiod\_hog \* hog）

从机器代码注册一组GPIO hog

#### 参数

struct gpiod\_hog \* hogs

gpio hog条目的表，末尾为零

### int gpiod\_count（struct device \* dev，const char \* con\_id）

返回与设备/功能关联的GPIO数量，如果没有将GPIO分配给请求的功能，则返回-ENOENT

#### 参数

struct device \* dev

GPIO消费者，可以为系统全局GPIO为NULL

const char \* con\_id

GPIO消费者中的功能

### struct gpio\_desc \* gpiod\_get（struct device \* dev，const char \* con\_id，enum gpiod\_flags flags）

获取给定GPIO功能的GPIO

#### 参数

struct device \* dev

GPIO消费者，可以为系统全局GPIO为NULL

const char \* con\_id

GPIO消费者中的功能

enum gpiod\_flags flags

可选GPIO初始化标志

#### 说明

返回与设备dev中的con\_id功能对应的GPIO说明符，如果没有将GPIO分配给请求的功能，则返回-ENOENT，或者如果在尝试获取GPIO时发生错误，则返回另一个IS\_ERR（）代码。

### struct gpio\_desc \* gpiod\_get\_optional（struct device \* dev，const char \* con\_id，enum gpiod\_flags flags）

为给定的GPIO功能获取一个可选的GPIO

#### 参数

struct device \* dev

GPIO消费者，可以为NULL以表示全局系统GPIO

const char \* con\_id

GPIO消费者内的功能

enum gpiod\_flags flags

可选的GPIO初始化标志

#### 说明

这等同于gpiod\_get()，除了当没有GPIO分配给请求的函数时将返回NULL。这对于需要处理可选GPIO的驱动程序非常方便。

### struct gpio\_desc \* gpiod\_get\_index(struct device \* dev，const char \* con\_id，unsigned int idx，enum gpiod\_flags flags)

从多索引GPIO函数获取GPIO

#### 参数

struct device \* dev

GPIO消费者，可以为NULL以表示全局系统GPIO

const char \* con\_id

GPIO消费者内的功能

unsigned int idx

要在消费者中获取的GPIO的索引

enum gpiod\_flags flags

可选的GPIO初始化标志

#### 说明

此gpiod\_get()的变体允许访问定义了多个GPIO的函数的第一个之外的GPIO。

返回一个有效的GPIO说明符， -ENOENT如果未为请求的函数和/或索引分配GPIO，则返回，或者如果在尝试获取GPIO时发生错误，则返回另一个IS\_ERR()代码。

### struct gpio\_desc \* gpiod\_get\_from\_of\_node(struct device\_node \* node，const char \* propname，int index，enum gpiod\_flags dflags，const char \* label)

从OF节点获取GPIO

#### 参数

struct device\_node \* node

OF节点的处理

const char \* propname

表示GPIO的DT属性的名称

int index

要为使用者获取的GPIO的索引

enum gpiod\_flags dflags

GPIO初始化标志

const char \* label

附加到请求的GPIO的标签

#### 返回

在成功请求时，GPIO引脚将按照提供的dflags进行配置。如果节点没有所请求的GPIO属性，则返回NULL。

在错误的情况下，返回ERR\_PTR()。

### struct gpio\_desc \* fwnode\_get\_named\_gpiod(struct fwnode\_handle \* fwnode，const char \* propname，int index，enum gpiod\_flags dflags，const char \* label)

从固件节点获取GPIO

#### 参数

struct fwnode\_handle \* fwnode

固件节点的处理

const char \* propname

表示GPIO的固件属性的名称

int index

要为使用者获取的GPIO的索引

enum gpiod\_flags dflags

GPIO初始化标志

const char \* label

附加到请求的GPIO的标签

#### 说明

此功能可用于从不透明固件获取其配置的驱动程序。

该函数正确使用底层固件接口查找相应的GPIO，然后确保在将其返回给调用者之前请求GPIO说明符。

#### 返回

在成功请求时，GPIO引脚将按照提供的dflags进行配置。

在错误的情况下，返回ERR\_PTR()。

### struct gpio\_desc \* gpiod\_get\_index\_optional(struct device \* dev，const char \* con\_id，unsigned int index，enum gpiod\_flags flags)

从多索引GPIO函数获取可选GPIO

#### 参数

struct device \* dev

GPIO消费者，可以为NULL以表示全局系统GPIO

const char \* con\_id

GPIO消费者内的功能

unsigned int index

要在消费者中获取的GPIO的索引

enum gpiod\_flags flags

可选的GPIO初始化标志

#### 说明

这等同于gpiod\_get\_index()，但是当没有分配带有指定索引的GPIO时，将返回NULL。这对于需要处理可选GPIO的驱动程序非常方便。

### struct gpio\_descs \* gpiod\_get\_array(struct device \* dev，const char \* con\_id，enum gpiod\_flags flags)

从多索引GPIO函数获取多个GPIO

#### 参数

struct device \* dev

GPIO消费者，可以为NULL以表示全局系统GPIO

const char \* con\_id

GPIO消费者内的功能

enum gpiod\_flags flags

可选的GPIO初始化标志

#### 说明

此函数获取给定功能下定义的所有GPIO。

返回一个包含说明符数组的struct gpio\_descs，如果未为请求的函数分配GPIO，则为-ENOENT，或者如果在获取GPIO时发生错误，则为另一个IS\_ERR()代码。

### struct gpio\_descs \* gpiod\_get\_array\_optional(struct device \* dev，const char \* con\_id，enum gpiod\_flags flags)

从多索引GPIO函数获取多个GPIO

#### 参数

struct device \* dev

GPIO消费者，可以为NULL以表示全局系统GPIO

const char \* con\_id

GPIO消费者内的功能

enum gpiod\_flags flags

可选的GPIO初始化标志

#### 说明

这等同于gpiod\_get\_array()，但是当没有分配给请求的函数时将返回NULL的GPIO。

### void gpiod\_put(struct gpio\_desc \* desc)

处理GPIO说明符

#### 参数

struct gpio\_desc \* desc

要处理的GPIO说明符

#### 说明

在调用gpiod\_put()之后，无法使用任何说明符。

### void gpiod\_put\_array(struct gpio\_descs \* descs)

处理多个GPIO说明符

#### 参数

struct gpio\_descs \* descs

包含说明符数组的struct gpio\_descs

## ACPI支持

void acpi\_gpiochip\_request\_interrupts(struct gpio\_chip \* chip)

为gpio芯片ACPI事件注册isr

#### 参数

struct gpio\_chip \* chip

GPIO芯片

#### 说明

ACPI5平台可以使用GPIO信号的ACPI事件。这些GPIO中断由需要从GPIO芯片的中断处理程序调用的ACPI事件方法处理。acpi\_gpiochip\_request\_interrupts查找哪些gpio引脚具有acpi事件方法，并分配中断处理程序来调用这些引脚的acpi事件方法。

### void acpi\_gpiochip\_free\_interrupts（struct gpio\_chip \* chip）

释放GPIO ACPI事件中断。

#### 参数

结构体gpio\_chip \* chip

GPIO芯片

#### 说明

释放与给定GPIO芯片的GPIO ACPI事件方法相关联的中断。

### int acpi\_dev\_gpio\_irq\_get（struct acpi\_device \* adev，int index）

查找GpioInt并将其转换为Linux IRQ号码

#### 参数

structacpi\_device \* adev

指向要从中获取IRQ的ACPI设备的指针

int index

GpioInt资源的索引（从0开始）

#### 说明

如果设备具有一个或多个GpioInt资源，则可以使用此函数将资源中的GPIO偏移量转换为Linux IRQ号码。

该函数是幂等的，虽然每次运行它都会根据GpioInt资源中的标志配置GPIO引脚方向。

#### 返回

成功时为Linux IRQ号码（> 0），失败时为负的errno。

## 设备树支持

### int of\_gpio\_simple\_xlate（struct gpio\_chip \* gc，const struct of\_phandle\_args \* gpiospec，u32 \* flags）

将gpiospec转换为GPIO编号和标志

#### 参数

struct gpio\_chip \* gc

指向gpio\_chip结构的指针

const struct of\_phandle\_args \* gpiospec

在设备树中找到的GPIO指示符

u32 \* flags

要填写的标志指针

#### 说明

这是一个简单的翻译函数，适用于大多数1：1映射的GPIO芯片。此函数仅执行一个正确性检查：GPIO是否小于ngpios（在gpio\_chip中指定）。

### int of\_mm\_gpiochip\_add\_data（struct device\_node \* np，struct of\_mm\_gpio\_chip \* mm\_gc，void \* data）

添加内存映射GPIO芯片（bank）

#### 参数

struct device\_node \* np

GPIO芯片的设备节点

struct of\_mm\_gpio\_chip \* mm\_gc

指向of\_mm\_gpio\_chip分配结构的指针

void \* data

要存储在struct gpio\_chip中的驱动程序数据

#### 说明

要使用此函数，您应该分配并填充mm\_gc：

在gpio\_chip结构中：-所有回调-of\_gpio\_n\_cells-of\_xlate回调（可选）

在of\_mm\_gpio\_chip结构中：-save\_regs回调（可选）

如果成功，此函数将映射银行的内存，并为您执行所有必要的工作。然后，您可以使用。回调来从.reg中管理GPIO。

### void of\_mm\_gpiochip\_remove（struct of\_mm\_gpio\_chip \* mm\_gc）

删除内存映射GPIO芯片（bank）

#### 参数

struct of\_mm\_gpio\_chip \* mm\_gc

指向of\_mm\_gpio\_chip分配结构的指针

## 设备管理API

结构体gpio\_desc \* devm\_gpiod\_get（struct device \* dev，const char \* con\_id，enum gpiod\_flags flags）

资源管理gpiod\_get（）

#### 参数

struct device \* dev

GPIO消费者

const char \* con\_id

GPIO消费者内的函数

enum gpiod\_flags flags

可选的GPIO初始化标志

#### 说明

gpiod\_get（）返回的GPIO说明符会在驱动程序分离时自动处理。从此函数返回的GPIO说明符也会自动处理。有关行为和返回的详细信息，请参见gpiod\_get（）。

### struct gpio\_desc \* devm\_gpiod\_get\_optional（struct device \* dev，const char \* con\_id，enum gpiod\_flags flags）

资源管理的gpiod\_get\_optional（）

#### 参数

struct device \* dev

GPIO消费者

const char \* con\_id

GPIO消费者内的函数

enum gpiod\_flags flags

可选的GPIO初始化标志

#### 说明

gpiod\_get\_optional（）的管理。从此函数返回的GPIO说明符也会自动处理。有关行为和返回的详细信息，请参见gpiod\_get\_optional（）。

### struct gpio\_desc \* devm\_gpiod\_get\_index（struct device \* dev，const char \* con\_id，unsigned int idx，enum gpiod\_flags flags）

资源管理gpiod\_get\_index（）

#### 参数

struct device \* dev

GPIO消费者

const char \* con\_id

GPIO消费者内的函数

unsigned int idx

要在消费者中获取的GPIO的索引

enum gpiod\_flags flags

可选的GPIO初始化标志

#### 说明

管理gpiod\_get\_index（）。从此函数返回的GPIO说明符也会自动处理。有关行为和返回的详细信息，请参见gpiod\_get\_index（）。

### struct gpio\_desc \* devm\_gpiod\_get\_from\_of\_node（struct device \* dev，const struct device\_node \* node，const char \* propname，int index，enum gpiod\_flags dflags，const char \* label）

从OF节点获取GPIO

#### 参数

struct device \* dev

用于生命周期管理的设备

const struct device\_node \* node

OF节点的句柄

const char \* propname

表示GPIO的DT属性的名称

int index

要获取的GPIO在消费者中的索引

enum gpiod\_flags dflags

GPIO初始化标志

const char \* label

附加到请求的GPIO的标签

#### 返回

在成功请求时，GPIO引脚会根据提供的dflags进行配置。

发生错误时，ERR\_PTR（）会返回。

### struct gpio\_desc \* devm\_fwnode\_get\_index\_gpiod\_from\_child（struct device \* dev，const char \* con\_id，int index，struct fwnode\_handle \* child，enum gpiod\_flags flags，const char \* label）

从设备的子节点获取GPIO说明符

#### 参数

struct device \* dev

GPIO消费者

const char \* con\_id

GPIO消费者内的函数

int index

要在消费者中获取的GPIO的索引

struct fwnode\_handle \* child

固件节点（dev的子节点）

enum gpiod\_flags flags

GPIO初始化标志

const char \*label

附加到所请求的GPIO上的标签

#### 说明

此函数返回的GPIO说明符会在驱动程序分离时自动处理。

成功请求后，GPIO引脚将按照提供的标志进行配置。

### struct gpio\_desc \*devm\_gpiod\_get\_index\_optional(struct device \*dev，const char \*con\_id，unsigned int index，enum gpiod\_flags flags)

受资源管理的gpiod\_get\_index\_optional()

#### 参数

struct device \*dev

GPIO使用者

const char \*con\_id

GPIO使用者内部的函数

unsigned int index

在使用者中获取GPIO的索引

enum gpiod\_flags flags

可选的GPIO初始化标志

#### 说明

管理的gpiod\_get\_index\_optional()。此函数返回的GPIO说明符会在驱动程序分离时自动处理。有关行为和返回的详细信息，请参见gpiod\_get\_index\_optional()。

### struct gpio\_descs \*devm\_gpiod\_get\_array(struct device \*dev，const char \*con\_id，enum gpiod\_flags flags)

受资源管理的gpiod\_get\_array()

#### 参数

struct device \*dev

GPIO使用者

const char \*con\_id

GPIO使用者内部的函数

enum gpiod\_flags flags

可选的GPIO初始化标志

#### 说明

管理的gpiod\_get\_array()。此函数返回的GPIO说明符会在驱动程序分离时自动处理。有关行为和返回的详细信息，请参见gpiod\_get\_array()。

### struct gpio\_descs \*devm\_gpiod\_get\_array\_optional(struct device \*dev，const char \*con\_id，enum gpiod\_flags flags)

受资源管理的gpiod\_get\_array\_optional()

#### 参数

struct device \*dev

GPIO使用者

const char \*con\_id

GPIO使用者内部的函数

enum gpiod\_flags flags

可选的GPIO初始化标志

#### 说明

受资源管理的gpiod\_get\_array\_optional()。此函数返回的GPIO说明符会在驱动程序分离时自动处理。有关行为和返回的详细信息，请参见gpiod\_get\_array\_optional()。

### void devm\_gpiod\_put(struct device \*dev，struct gpio\_desc \*desc)

受资源管理的gpiod\_put()

#### 参数

struct device \*dev

GPIO使用者

struct gpio\_desc \*desc

要处理的GPIO说明符

#### 说明

处理使用devm\_gpiod\_get()或devm\_gpiod\_get\_index()获取的GPIO说明符。通常不会调用此函数，因为GPIO将由资源管理代码处理。

### void devm\_gpiod\_put\_array(struct device \*dev，struct gpio\_descs \*descs)

受资源管理的gpiod\_put\_array()

#### 参数

struct device \*dev

GPIO使用者

struct gpio\_descs \*descs

要处理的GPIO说明符数组

#### 说明

处理使用devm\_gpiod\_get\_array()获取的GPIO说明符数组。通常不会调用此函数，因为GPIO将由资源管理代码处理。

### int devm\_gpio\_request(struct device \*dev，unsigned gpio，const char \*label)

为受管理的设备请求GPIO

#### 参数

struct device \*dev

要请求GPIO的设备

unsigned gpio

要分配的GPIO

const char \*label

所请求的GPIO的名称

除了额外的dev参数外，此函数与gpio\_request()使用相同的参数并执行相同的功能。使用此函数请求的GPIO将在驱动程序分离时自动释放。

### int devm\_gpio\_request\_one(struct device \*dev，unsigned gpio，unsigned long flags，const char \*label)

请求初始化设置的单个GPIO

#### 参数

struct device \*dev

要请求的设备

unsigned gpio

GPIO编号

unsigned long flags

由GPIOF\_\*指定的GPIO配置

const char \*label

此GPIO的文字说明字符串

### void devm\_gpio\_free(struct device \*dev，unsigned int gpio)

释放GPIO

#### 参数

struct device \*dev

要为其释放GPIO的设备

unsigned int gpio

要释放的GPIO

#### 说明

除了额外的dev参数外，此函数与gpio\_free()使用相同的参数并执行相同的功能。与gpio\_free()相比，应使用此函数手动释放使用devm\_gpio\_request()分配的GPIO。

## sysfs助手

### int gpiod\_export(struct gpio\_desc \*desc，bool direction\_may\_change)

通过sysfs导出GPIO

#### 参数

struct gpio\_desc \*desc

要使其可用的GPIO，已请求

bool direction\_may\_change

如果用户空间可以更改GPIO方向，则为true

#### 上下文

arch\_initcall或更高

#### 说明

当驱动程序想在已请求的GPIO之后使其对用户空间可访问（例如在调试期间或作为其公共接口的一部分）时，可以使用此例程。如果GPIO可以更改方向（有些不能），并且调用者允许，则用户空间将看到“direction”sysfs属性，该属性可用于更改gpio的方向。始终提供“value”属性。

成功时返回零，否则返回错误。

### int gpiod\_export\_link(struct device \*dev，const char \*name，struct gpio\_desc \*desc)

创建到已导出GPIO节点的sysfs链接

#### 参数

struct device \*dev

要在其下创建符号链接的设备

const char \*name

符号链接的名称

struct gpio\_desc \*desc

要创建符号链接的GPIO，已导出

#### 说明

设置一个符号链接从/sys/…/dev/name到/sys/class/gpio/gpioN节点。调用者负责取消链接。

成功时返回零，否则返回错误。

### void gpiod**\_**unexport(struct gpio\_desc \*desc)

撤销gpiod\_export()的效果

#### 参数

struct gpio\_desc \*desc

要使其不可用的GPIO

#### 说明

这对于gpiod\_free()是隐含的。

# 其他设备

## int misc\_register(struct miscdevice \*misc)

注册杂项设备

#### 参数

struct miscdevice \*misc

设备结构

#### 说明

在内核中注册一个杂项设备。如果次要号设置为MISC\_DYNAMIC\_MINOR，则会分配并将次要号放入结构体的次要字段中。对于其他情况，将使用请求的次要号。

传递的结构与内核链接，在注销之前不能被销毁。默认情况下，对设备的open()系统调用将file->private\_data设置为指向该结构。驱动程序不需要在fops中使用open()。

成功返回零，失败返回负数的错误代码。

## void misc\_deregister(struct miscdevice \*misc)

注销一个杂项设备

#### 参数

struct miscdevice \*misc

要注销的设备

#### 说明

注销先前已成功注册的杂项设备misc\_register()。

# DMAEngine文档

DMAEngine文档提供了有关DMAEngine框架各个方面的文档。

## DMAEngine文档

本书介绍了DMAEngine内部API和DMAEngine设备驱动程序编写指南。.

### DMAengine 控制器文档

#### 硬件介绍

大多数从设备DMA控制器都有相同的操作原则。

它们有一定数量的可用于DMA传输的通道和请求线。

请求和通道几乎是正交的。通道可以用于为多个请求提供服务。为简化起见，通道是将进行复制的实体，而请求是涉及的端点。

请求数字实际上对应于从DMA可用设备到控制器本身的物理线路。每当设备想要启动传输时，它将通过断言请求线来断言DMA请求(DRQ)。

非常简单的DMA控制器只会考虑一个参数：传输大小。在每个时钟周期，它将从一个缓冲区传输一个字节的数据到另一个缓冲区，直到达到传输大小为止。

这在现实世界中不起作用，因为从设备可能需要在单个周期内传输特定数量的位。例如，我们可能希望在执行简单的内存复制操作时传输尽可能多的数据以最大化性能，但我们的音频设备可能有一个需要一次性写入精确的16或24位数据的较窄FIFO。这就是为什么大多数如果不是所有的DMA控制器都可以使用调整这个参数，这个参数叫做传输宽度。

此外，一些DMA控制器无论何时使用RAM作为源或目的地，都可以将内存中的读取或写入分组成缓冲区，因此您将获得多个大型传输，而不是许多小型内存访问，这不是非常高效的。这是使用一个称为burst size的参数完成的，该参数定义了在控制器将传输拆分成更小的子传输之前允许进行多少个单独的读取/写入。

我们的理论DMA控制器将只能进行涉及单个连续数据块的传输。然而，我们通常进行的一些传输并非如此，并希望将数据从非连续的缓冲区复制到连续的缓冲区，这称为分散-聚集。

至少对于mem2dev传输，DMAEngine需要支持分散-聚集。因此，在这里我们留下了两种情况：要么我们有一个非常简单的DMA控制器，不支持它，我们将不得不在软件中实现它，要么我们有一个更高级的DMA控制器，它以硬件方式实现分散-聚集。

后者通常是使用一组用于传输的块编程的，每当传输开始时，控制器将在该集合中进行，执行我们在那里编程的任何操作。

此集合通常是一个表或链接列表。然后，您将将表的地址及其元素数量或列表的第一个项推送到DMA控制器的一个通道中，每当DRQ被断言时，它将通过该集合以知道从何处获取数据。

无论哪种方式，这个集合的格式完全取决于您的硬件。每个DMA控制器都需要不同的结构，但是它们都需要至少为每个块提供源和目标地址，无论是否应该增加这些地址，以及前面看到的三个参数：burst size、transfer width和transfer size。

最后一件事通常是，从设备不会默认发出DRQ，您必须首先在从设备驱动程序中启用它，每当您愿意使用DMA时。

这些仅是一般的内存到内存(mem2mem)或内存到设备(mem2dev)传输。大多数设备通常支持DMAEngine支持的其他传输或内存操作，这将在本文档的后面详细说明。

#### Linux中的DMA支持

历史上，DMA控制器驱动程序使用异步TX API来实现，以卸载内存复制、XOR、加密等操作，基本上是任何内存到内存操作。

随着时间的推移，内存到设备传输的需求出现了，dmaengine得到了扩展。现在，异步TX API被编写为dmaengine上部的一层，并充当客户端。尽管如此，在某些情况下，dmaengine仍然可以容纳该API，并做出一些设计选择，以确保其保持兼容。

有关异步TX API的更多信息，请查看异步传输/变换API中的相关文档文件。

#### DMAEngine APIs

**结构体dma\_device 初始化**

与任何其他内核框架一样，整个DMAEngine注册都依赖于驱动程序填充一个结构并针对该框架进行注册。在我们的情况下，该结构是dma\_device。

您在驱动程序中需要做的第一件事是分配此结构。任何通常的内存分配器都可以使用，但您还需要在其中初始化一些字段：

1）通道channels：应使用INIT\_LIST\_HEAD宏初始化为列表

src\_addr\_widths：应包含支持的源传输宽度的位掩码

dst\_addr\_widths：应包含支持的目标传输宽度的位掩码

方向directions：应包含支持的从属方向的位掩码（即不包括mem2mem传输）  
余数粒度residue\_granularity：dma\_set\_residue报告传输剩余物的粒度。这可以是：

1. 描述符Descriptor：您的设备不支持任何种类的残留物报告。框架只会知道特定事务描述符已完成。
2. 段Segment：您的设备能够报告已传输的块
3. 突发Burst：您的设备能够报告已传输的突发

dev：应持有与您当前驱动程序实例关联的struct device指针。

#### 支持的事务类型

接下来，您需要设置设备（和驱动程序）支持的事务类型。

我们的dma\_device结构具有一个称为cap\_mask的字段，该字段保存支持的各种事务类型，您需要使用dma\_cap\_set函数修改此掩码，并根据您支持的事务类型使用各种标志作为参数。

所有这些功能都在dma\_transaction\_type枚举中定义，包括/include/linux/dmaengine.h  
目前可用的类型为：

DMA\_MEMCPY

设备能够执行内存到内存复制

DMA\_XOR

设备能够在内存区域上执行XOR操作

用于加速XOR密集任务，例如RAID5

DMA\_XOR\_VAL

设备能够使用XOR算法针对内存缓冲区执行奇偶校验

DMA\_PQ

设备能够执行RAID6 P+Q计算，其中P是简单的XOR，而Q是Reed-Solomon算法

DMA\_PQ\_VAL

设备能够使用RAID6 P+Q算法针对内存缓冲区执行奇偶校验

DMA\_INTERRUPT

设备能够触发将生成周期性中断的虚拟传输。

客户端驱动程序使用此功能在DMA控制器中断期间定期调用回调

DMA\_PRIVATE

该设备仅支持从属传输，并且因此不可用于异步传输

DMA\_ASYNC\_TX

设备不得设置，并且将在需要时由框架设置

待办事项：关于什么？

DMA\_SLAVE

设备可以处理设备到内存传输，包括散裂传输

在mem2mem情况下，我们有两种不同的类型来处理单个要复制的块或其中的集合，在这里，我们只有一种单一的事务类型，应该处理两者。

如果要传输单个连续的内存缓冲区，只需构建仅包含一个项目的散布列表

DMA\_CYCLIC

设备可以处理循环传输

循环传输是一种块集合将循环自己，最后一项指向第一项的传输

通常用于音频传输，其中您想要操作单个环形缓冲区，您将使用它来填充音频数据

DMA\_INTERLEAVE

该设备支持交错传输

这些传输可以将数据从非连续缓冲区传输到非连续缓冲区，而DMA\_SLAVE可以将数据从非连续数据集传输到连续目标缓冲区

通常用于2D内容转移，这种情况下，您希望直接将未压缩的数据部分传输到显示器以打印

这些不同的类型也会影响源和目的地地址随时间变化的方式

指向RAM的地址通常会在每次传输之后增加（或减小）。在环形缓冲区的情况下，它们可能会循环（DMA\_CYCLIC）。指向设备寄存器（例如FIFO）的地址通常是固定的。

**设备操作**

我们的dma\_device结构还需要一些函数指针，以实现实际逻辑，现在我们已经描述了我们能够执行的操作。

我们必须在那里填充的功能，因此必须实现，显然取决于您报告为支持的交易类型。

device\_alloc\_chan\_resources

device\_free\_chan\_resources

1. 每当驱动程序在与该驱动程序关联的通道上第一次/最后一次调用dma\_request\_channel或dma\_release\_channel时，这些函数都将被调用
2. 他们负责分配/释放所有需要的资源，以使该通道对您的驱动程序有用
3. 这些功能可能会休眠。

device\_prep\_dma\_\*

1. 这些功能与您先前注册的功能相匹配。
2. 这些功能都采用与所准备传输相关的缓冲区或分散列表，并应从中创建硬件描述符或硬件描述符列表
3. 这些功能可以从中断上下文中调用
4. 您可能进行的任何分配都应使用GFP\_NOWAIT标志，以避免潜在休眠，但同时不耗尽紧急池
5. 驱动程序应尝试预分配可能需要的任何内存，以避免在探测时对立即分配器施加过多压力
6. 它应返回dma\_async\_tx\_descriptor结构的唯一实例，该结构进一步表示特定的传输。
7. 可以使用函数dma\_async\_tx\_descriptor\_init初始化此结构。
8. 您还需要在此结构中设置两个字段：

flags：待定：驱动程序本身是否可以修改它，还是应始终传递的标志

tx\_submit：指向您必须实现的函数的指针，该函数应将当前事务描述符推送到等待发出的挂起队列中。

1. 在此结构中，可以通过初始化函数指针callback\_result来通知提交者事务已完成。在早期的代码中，已使用函数指针callback。但是它不提供交易状态，并且将被弃用。传递给回调结果的定义为dmaengine\_result的结果结构具有两个字段：

result：提供dmaengine\_tx\_result定义的传输结果。成功或某些错误情况。

残留residue：为那些支持残余的传输提供残余字节。

device\_issue\_pending

1. 获取挂起队列中的第一个事务描述符，并开始传输。每当该传输完成时，它都应移动到列表中的下一个事务
2. 该函数可以在中断上下文中调用

device\_tx\_status

1. 应报告要在给定通道上继续进行的字节数
2. 只应关注作为参数传递的事务描述符，而不是给定通道上当前活动的事务描述符
3. tx\_state参数可能为NULL
4. 应使用dma\_set\_residue报告它
5. 在循环传输的情况下，它只应考虑循环缓冲区的总大小。
6. 如果设备不支持有序完成，并且正在按顺序完成操作，则应返回DMA\_OUT\_OF\_ORDER。
7. 此函数可以在中断上下文中调用。

device\_config

1. 使用作为参数给出的配置重新配置通道
2. 此命令不应同步执行，也不应在任何当前排队传输上执行，而仅应在随后的传输中执行
3. 在这种情况下，函数将作为参数接收一个dma\_slave\_config结构指针，该结构将详细说明要使用的配置。
4. 即使该结构包含方向字段，该字段也已弃用，而应优先使用prep\_\*函数中给定的方向参数
5. 此调用仅适用于从设备操作。对于memcpy操作，不应设置此项，也不应期望设置此项。如果驱动程序支持两者，则应仅将此调用用于从机操作，而不用于memcpy操作。

device\_pause

1. 暂停通道上的传输
2. 此命令应在通道上同步运行，立即暂停给定通道的工作

device\_resume

1. 在通道上恢复传输
2. 此命令应在通道上同步运行，并立即恢复给定通道的工作。

device\_terminate\_all

1. 中止通道上所有待处理和正在进行的传输。
2. 对于已中止的传输，不应调用完整的回调函数。
3. 可以从原子上下文或描述符的完整回调函数中调用。不得休眠。驱动程序必须能够正确处理此操作。
4. 终止可能是异步的。驱动程序不必等待当前激活的传输完全停止。请参阅device\_synchronize。

device\_synchronize

1. 必须将通道的终止与当前上下文同步。
2. 必须确保DMA控制器不再访问先前提交的描述符的内存。
3. 必须确保所有先前提交的描述符的完整回调已经运行完成，且没有计划要运行。
4. 可以休眠。

#### 其它注意事项

（应该记录的内容，但不知道该放在哪里）

1. dma\_run\_dependencies

应在异步TX传输结束时调用，在从设备传输的情况下可以忽略。

确保在标记为完成之前运行相关的操作。

1. dma\_cookie\_t

是一个DMA事务ID，会随时间递增。

自从引入了抽象事务的virt-dma之后，这不再重要。

1. DMA\_CTRL\_ACK

如果清除，提供者不能重新使用描述符，直到客户端确认接收到消息，即有机会建立任何依赖关系链

可以通过调用async\_tx\_ack()来确认

如果设置，则不意味着可以重复使用描述符

1. DMA\_CTRL\_REUSE

如果设置，则在完成后可以重复使用描述符。如果设置了此标志，提供者不应释放描述符。

应通过调用dmaengine\_desc\_set\_reuse()准备重用描述符，这将设置DMA\_CTRL\_REUSE。

dmaengine\_desc\_set\_reuse()只有在通道支持可重复使用的描述符的情况下才会成功。

因此，如果设备驱动程序想跳过2个传输之间的dma\_map\_sg()和dma\_unmap\_sg()，因为DMA的数据没有被使用，它可以在完成后立即重新提交传输。

描述符可以通过以下几种方式释放：

1. 通过调用dmaengine\_desc\_clear\_reuse()清除DMA\_CTRL\_REUSE，并提交到最后一个事务
2. 显式调用dmaengine\_desc\_free()，只有当DMA\_CTRL\_REUSE已经设置时才能成功
3. 终止通道
4. DMA\_PREP\_CMD

如果设置，客户端驱动程序告诉DMA控制器传递给DMA API的数据是命令数据。

命令数据的解释是DMA控制器特定的。它可用于发出指令到其他外围设备/寄存器读取/寄存器写入，其描述符应与正常数据描述符的格式不同。

#### 一般设计说明

你会看到大多数DMAEngine驱动程序基于类似的设计，处理处理程序中的传输结束中断，但延迟大多数工作到任务处理程序中，包括在前一个传输结束后启动新传输。

然而，这是一种相当低效的设计，因为跨传输的延迟不仅是中断延迟，还包括任务处理程序的调度延迟，在两者之间留下通道空闲，降低了全局传输速率。

你应该避免这种做法，而不是在任务处理程序中选举一个新的传输，将该部分移动到中断处理程序中，以便具有更短的空闲窗口（无论如何我们无法避免）。

#### 词汇表

突发Burst：在刷新到内存之前，可以将多个连续的读取或写入操作排队到缓冲区的操作数。

块Chunk：一组连续的突发

传输Transfer：一组块（无论是连续的还是不连续的）

## DMAEngine客户端文档

本书是一个指南，介绍了如何使用DMAEngine的Slave-DMA API编写设备驱动程序。这仅适用于Slave DMA使用情况。

### [DMA Engine API 指](https://www.kernel.org/doc/html/latest/driver-api/dmaengine/client.html)南

Vinod Koul <vinod dot koul at intel.com>

**注**

有关在async\_tx中使用DMA Engine的信息，请参见：Documentation/crypto/async-tx-api.rst

以下是指南，介绍如何使用DMA Engine的Slave-DMA API编写设备驱动程序。这仅适用于Slave DMA使用情况。

#### DMA 用法

从属DMA使用包括以下步骤：

1. 分配DMA从设备的通道
2. 设置从设备和控制器特定的参数
3. 获取事务的描述符
4. 提交事务
5. 发出待处理请求并等待回调通知

这些操作的详细信息如下：

1）分配DMA从设备的通道

在Slave DMA上下文中，通道分配略有不同，客户端驱动程序通常仅需要来自特定DMA控制器的通道，在某些情况下，需要特定的通道。要请求通道，请使用dma\_request\_chan（）API。

接口:

struct dma\_chan \*dma\_request\_chan(struct device \*dev, const char \*name);

它将查找并返回与“dev”设备相关联的名称DMA通道。通过DT、ACPI或基于板文件的dma\_slave\_map匹配表完成关联。

通过该接口分配的通道对调用者是独占的，直到调用dma\_release\_channel()为止。

2）设置从设备和控制器特定参数

下一步总是将一些特定信息传递给DMA驱动程序。大多数从设备DMA可使用的通用信息都在struct dma\_slave\_config中。这允许客户端为外围设备指定DMA方向、DMA地址、总线宽度、DMA突发长度等。

如果某些DMA控制器有更多要发送的参数，则应尝试将struct dma\_slave\_config嵌入其控制器特定结构中。如果需要，客户端就可以传递更多参数。

接口:

int dmaengine\_slave\_config(struct dma\_chan \*chan,

struct dma\_slave\_config \*config)

请参见dmaengine.h中的dma\_slave\_config结构定义，以获取有关结构成员的详细说明。请注意，“方向”成员将消失，因为它重复了在准备调用中给出的方向。

3）获取事务的描述符

对于从设备使用，DMA引擎支持的各种从设备传输模式为：

slave\_sg：从/到外围设备DMA散列表

dma\_cyclic：从/到外围设备执行周期性DMA操作，直到显式停止操作。

interleaved\_dma：这对从设备和M2M客户端都是通用的。对于外围设备，设备FIFO的地址可能已经为驱动程序所知。可以通过将适当的值设置为“dma\_interleaved\_template”成员来表达各种类型的操作。如果通道支持，则可以通过设置DMA\_PREP\_REPEAT传输标志来执行循环交错DMA传输。

此传输API的非NULL返回表示给定事务的“描述符”。

接口:

struct dma\_async\_tx\_descriptor \*dmaengine\_prep\_slave\_sg(

struct dma\_chan \*chan, struct scatterlist \*sgl,

unsigned int sg\_len, enum dma\_data\_direction direction,

unsigned long flags);

struct dma\_async\_tx\_descriptor \*dmaengine\_prep\_dma\_cyclic(

struct dma\_chan \*chan, dma\_addr\_t buf\_addr, size\_t buf\_len,

size\_t period\_len, enum dma\_data\_direction direction);

struct dma\_async\_tx\_descriptor \*dmaengine\_prep\_interleaved\_dma(

struct dma\_chan \*chan, struct dma\_interleaved\_template \*xt,

unsigned long flags);

在调用dmaengine\_prep\_slave\_sg()之前，期望外围驱动程序已将散列表映射为DMA操作，并且必须保持直到DMA操作完成。散列表必须使用DMA struct device进行映射。如果需要稍后同步映射，则必须使用DMA struct device调用dma\_sync\_\*\_for\_\*()。因此，正常设置应如下所示:

nr\_sg = dma\_map\_sg(chan->device->dev, sgl, sg\_len);

if (nr\_sg == 0)

/\* error \*/

desc = dmaengine\_prep\_slave\_sg(chan, sgl, nr\_sg, direction, flags);

一旦获得描述符并添加了回调信息，就必须提交描述符。一些DMA引擎驱动程序可能在成功准备和提交之间保持自旋锁，因此这两个操作非常重要。

**注**

尽管async\_tx API指定完成回调例程不能提交任何新操作，但对于从设备/循环DMA而言，情况并非如此。

对于从设备DMA，后续事务可能在回调函数被调用之前无法提交，因此允许从设备DMA回调准备并提交新事务。

对于循环DMA，回调函数可能希望通过dmaengine\_terminate\_async()终止DMA。

因此，在调用回调函数可能导致死锁之前，DMA引擎驱动程序放弃任何锁是很重要的。

请注意，回调将始终从DMA引擎tasklet调用，而不是从中断上下文调用。

4）提交事务

一旦准备了描述符并添加了回调信息，就必须将其放在DMA引擎驱动程序的挂起队列中。

接口:

dma\_cookie\_t dmaengine\_submit(struct dma\_async\_tx\_descriptor \*desc)

这返回一个可用于通过本文档未涉及的其他DMA引擎调用检查DMA引擎活动进度的cookie。

dmaengine\_submit()不会启动DMA操作，它只是将它添加到挂起队列中。有关此操作，请参见第5步，dma\_async\_issue\_pending。

5）发出挂起的DMA请求并等待回调通知

通过调用issue\_pending API可以激活排队的事务。如果通道空闲，则启动队列中的第一个事务并排队其余事务。

在每个DMA操作完成后，会启动队列中的下一个，并触发一个tasklet。如果设置了，则任务会调用客户端驱动程序完成回调例程进行通知。

接口:

void dma\_async\_issue\_pending(struct dma\_chan \*chan);

#### 更多APIs

**终止APIs**

int dmaengine\_terminate\_sync(struct dma\_chan \*chan)

int dmaengine\_terminate\_async(struct dma\_chan \*chan)

int dmaengine\_terminate\_all(struct dma\_chan \*chan) /\* DEPRECATED \*/

这会导致DMA通道的所有活动都停止，并且可能会丢弃尚未完全传输的DMA FIFO中的数据。任何不完整的传输都不会调用回调函数。

此函数有两个变体可用。

dmaengine\_terminate\_async() 可能不会等待 DMA 完全停止或任何正在运行的完成回调结束。但可以从原子上下文或完整回调内调用 dmaengine\_terminate\_async()。dmaengine\_synchronize() 必须在释放 DMA 传输访问的内存或释放在完整回调中访问的资源之前调用才是安全的。

dmaengine\_terminate\_sync() 将等待传输和任何正在运行的完成回调完成，然后才返回。但是，不能从原子上下文或完整回调内调用该函数。

dmaengine\_terminate\_all() 已过时，新代码中不应使用。

**暂停 API**

int dmaengine\_pause(struct dma\_chan \*chan)

这会暂停 DMA 通道上的活动而不丢失数据。

**恢复API**

int dmaengine\_resume(struct dma\_chan \*chan)

恢复之前暂停的 DMA 通道。不允许恢复当前未暂停的通道。

**检查 Txn 完成**

enum dma\_status dma\_async\_is\_tx\_complete(struct dma\_chan \*chan,

dma\_cookie\_t cookie, dma\_cookie\_t \*last, dma\_cookie\_t \*used)

可以用来检查通道的状态。有关此 API 的更完整的描述，请参见 include/linux/dmaengine.h 中的文档。

可以与 dmaengine\_submit() 返回的 cookie 和 dma\_async\_is\_complete() 一起使用，以检查特定 DMA 事务的完成情况。

**注**

并非所有 DMA 引擎驱动程序都能为运行中的 DMA 通道返回可靠信息。建议 DMA 引擎用户在使用此 API 之前暂停或停止（通过 dmaengine\_terminate\_all()）该通道。

**同步终止API**

void dmaengine\_synchronize(struct dma\_chan \*chan)

将 DMA 通道的终止与当前上下文同步。

该函数应在 dmaengine\_terminate\_async() 之后使用以将 DMA 通道的终止与当前上下文同步。该函数将等待传输和任何正在运行的完成回调完成后才返回。

如果使用 dmaengine\_terminate\_async() 停止 DMA 通道，则必须在安全释放先前提交的描述符访问的内存或在先前提交的描述符的完成回调中访问的任何资源之前调用此函数。

如果在 dmaengine\_terminate\_async() 和此函数之间调用了 dma\_async\_issue\_pending()，则此函数的行为未定义。

## DMA 测试文档

本书介绍如何使用 dmatest 模块测试 DMA 驱动程序。

### [DMA 测](https://www.kernel.org/doc/html/latest/driver-api/dmaengine/dmatest.html)试指南

Andy Shevchenko <[andriy.shevchenko@linux.intel.com](mailto:andriy.shevchenko@linux.intel.com)>

本小文档介绍了如何使用 dmatest 模块测试 DMA 驱动程序。

注

测试套件仅适用于至少具有以下一种能力的通道：DMA\_MEMCPY（内存到内存），DMA\_MEMSET（const 到内存或内存到内存，在仿真情况下），DMA\_XOR，DMA\_PQ。

#### 第 1 部分-如何构建测试模块

菜单包含一个选项，可以按以下路径找到：

设备驱动程序-> DMA 引擎支持-> DMA 测试客户端

在配置文件中，名为 CONFIG\_DMATEST 的选项。 dmatest 可以构建为模块或内核内部。让我们考虑这些情况。

#### 第 2 部分-当 dmatest 构建为模块时

用法实例:

% modprobe dmatest timeout=2000 iterations=1 channel=dma0chan0 run=1

或:

% modprobe dmatest

% echo dma0chan0 > /sys/module/dmatest/parameters/channel

% echo 2000 > /sys/module/dmatest/parameters/timeout

% echo 1 > /sys/module/dmatest/parameters/iterations

% echo 1 > /sys/module/dmatest/parameters/run

或在内核命令行上:

dmatest.channel=dma0chan0 dmatest.timeout=2000 dmatest.iterations=1 dmatest.run=1

提示：

可用通道列表可以通过运行以下命令提取:

% ls -1 /sys/class/dma/

启动后，会发出类似于“ dmatest的消息：添加了 1 个线程使用 dma0chan0”的消息。之后只报告测试失败消息，直到测试停止。

请注意，运行新测试不会停止正在进行的任何测试。

以下命令返回测试的状态。

% cat /sys/module/dmatest/parameters/run

为了等待测试完成，用户空间可以轮询“运行”，直到其为假，或使用 wait 参数。在加载模块时指定“wait=1”会导致模块初始化暂停，直到测试运行完成，而读取 /sys/module/dmatest/parameters/wait 则等待任何正在运行的测试完成后返回。例如，以下脚本在退出之前等待 42 个测试完成。请注意，如果“iterations”设置为“ infinite”，则禁用等待。

示例:

% modprobe dmatest run=1 iterations=42 wait=1

% modprobe -r dmatest

或:

% modprobe dmatest run=1 iterations=42

% cat /sys/module/dmatest/parameters/wait

% modprobe -r dmatest

#### 第 3 部分-当内置在内核中时

提供给内核命令行的模块参数将用于执行的第一次测试。用户获得控制权后，可以使用相同或不同的参数重新运行测试。有关详细信息，请参见上面的第 2 部分-当 dmatest 构建为模块时。

在这两种情况下，模块参数被用作测试用例的实际值。您可以通过运行以下命令在运行时进行检查：

% grep -H . /sys/module/dmatest/parameters/\*

#### 第4部分-收集测试结果

测试结果以以下格式打印到内核日志缓冲区中:

"dmatest: result <channel>: <test id>: '<error msg>' with src\_off=<val> dst\_off=<val> len=<val> (<err code>)"

输出示例:

% dmesg | tail -n 1

dmatest: result dma0chan0-copy0: #1: No errors with src\_off=0x7bf dst\_off=0x8ad len=0x3fea (0)

消息格式在不同类型的错误之间统一。括号中的数字表示附加信息，例如错误代码、错误计数或状态。测试线程还在完成时发出总结行，列出执行的测试数、失败的数目和结果代码。

示例:

% dmesg | tail -n 1

dmatest: dma0chan0-copy0: summary 1 test, 0 failures 1000 iops 100000 KB/s (0)

数据不匹配错误的详细信息也会被发出，但不符合上述格式。

## PXA DMA 文档

这本书增加了一些关于PXA DMA的注释。

### [PXA/MMP - DMA 从](https://www.kernel.org/doc/html/latest/driver-api/dmaengine/pxa_dma.html)设备控制器

#### 限制

1. 传输热排队一个驱动程序提交一个传输并发出它，应该允许即使在运行的DMA通道上也排队等待传输。这意味着队列不等待前一个传输结束，并且描述符链接不仅在传输结束时由irq/tasklet代码完成。在物理上提交并发出的传输不需要等待物理传输停止和重新启动，但是在运行中的通道上提交。其他驱动程序，特别是mmp\_pdma，需要等待物理传输停止后才能重新启动新的传输。
2. 所有要求确认的转移都应该被通知发出DMA\_PREP\_INTERRUPT的任何发出的传输都应该触发回调调用。这意味着即使irq/tasklet是由tx1结束触发的，但在irq/dma tx2的时间结束，tx1->complete()和tx2->complete()也应该被调用。
3. 通道运行状态驱动程序应该能够查询通道是否运行或停止。对于多媒体案例，例如视频捕获，如果提交了一个传输，然后检查DMA通道报告“停止通道”，则在下一个“帧开始中断”之前不应发出传输，因此需要知道通道是否处于运行或停止状态。
4. 带宽保证PXA架构具有4个级别的DMA优先级:高、正常、低。高优先级的带宽是正常的两倍，正常的带宽是低优先级的两倍。驱动程序应该能够请求优先级，特别是实时的，例如pxa\_camera，带有（大的）吞吐量。

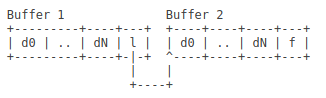
#### 设计

1. 虚拟通道与sa11x0驱动程序相同的概念，即为请求者线路分配“虚拟通道”，当传输发出时，将动态分配物理DMA通道。
2. 散列表传输的传输解剖图

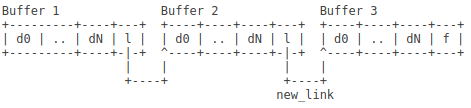


此结构指向dma->sg\_cpu。描述符使用如下：

1. desc-sg[i]：第i个描述符，将第i个sg元素传输到视频缓冲散射聚集区
2. 状态更新器将单个u32传输到已知的dma一致内存中，以留下此传输完成的痕迹。 “well known”是唯一的物理通道，这意味着阅读此值将告诉此时的最后一个已完成的传输。
3. 完成器：具有ddadr=DADDR\_STOP，dcmd=ENDIRQEN
4. 链接器：具有ddadr =下一个传输的desc-sg [0]，dcmd = 0
5. 传输热链假设正在运行的链为:



调用dmaengine\_submit（b3）后，链接如下所示:



如果创建new\_link时DMA通道停止，则不会重新启动。热链不会打破使用dma\_async\_issue\_pending（）确保实际启动传输的假设。

对这个规则的一种例外：

1. 如果Buffer1和Buffer2的所有地址都对齐为8个字节
2. 并且如果Buffer3至少有一个地址不是4个字节对齐的

那么热链接不能发生，因为必须停止通道，设置“对准位”，然后重新启动通道。因此，这种传输tx\_submit（）将排队在提交队列中，并且特定情况下，如果DMA已在对齐模式下运行。

4）传输完成更新器每次在通道上完成一次传输时，可能会生成中断或不生成，取决于客户的请求。但在每种情况下，传输的最后一个描述符，“状态更新器”，将把最新完成的传输写入物理通道的完成标记中。

这将加速残留计算，对于像视频缓冲区这样持有约6k个描述符或更多的大型传输非常有用。这也允许在不锁定的情况下找到运行DMA链中最新完成的传输。

5）传输完成，中断和tasklet。当完成标记为“DMA\_PREP\_INTERRUPT”的传输完成后，dma irq会被触发。在这个中断上，一个tasklet会被调度到物理通道上。

Tasklet负责：

读取物理通道的最后一个更新标记

调用所有完成的传输的传输回调函数，基于该标记和每个传输标志。

如果在进行此处理时完成了传输，则会触发DMA中断，并且将再次安排Tasklet，具有新的更新器标记。

6）残留粒度将基于描述符。已发出但尚未完成的传输将扫描其描述符，与当前运行的描述符进行比较。

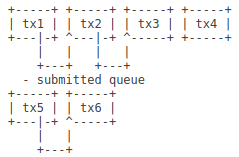
7）驱动程序tx队列的最复杂情况是：

最棘手的情况是：

1. 没有“acked”传输（tx0）
2. 驱动程序提交了一个对齐的tx1，不是链式的
3. 驱动程序提交了一个对齐的tx2 => tx2冷链到tx1
4. 驱动程序发出了tx1 + tx2 =>通道在对齐模式下运行
5. 驱动程序提交了一个对齐的tx3 => tx3是热链
6. 驱动程序提交了不对齐的tx4 => tx4放在提交队列中，未链式
7. 驱动程序发出了tx4 => tx4放在发出队列中，未链式
8. 驱动程序提交了一个对齐的tx5 => tx5被放在提交队列中，未链式
9. 驱动程序提交了一个对齐的tx6 => tx6放在提交队列中，冷链到tx5

这转化为（tx4发出后）：

1. 发出队列



1. 完成的队列：空
2. 分配队列：tx0

应该注意的是，在tx3完成后，通道将停止，并以“非对齐模式”重新启动以处理tx4。

# Linux内核SLIMbus支持

## 概述

### 什么是SLIMbus?

SLIMbus（串行低功耗互连媒体总线）是由MIPI（移动行业处理器接口）联盟开发的规范。总线使用主/从配置，并且是2线路多点实现（时钟和数据）。  
目前，SLIMbus用于应用处理器和外围组件（通常是编解码器）之间的接口。SLIMbus使用时分多路复用来容纳多个数据通道和一个控制通道。

控制通道用于各种控制功能，例如总线管理、配置和状态更新。这些消息可以是单播的（例如读/写设备特定值），也可以是多播的（例如，数据通道重新配置序列是向所有设备公布的广播消息）。数据通道用于在2个SLIMbus设备之间传输数据。数据通道使用设备上的专用端口。

## 硬件说明:

SLIMbus规范具有基于其功能的不同类型的设备分类。管理器设备负责枚举、配置和动态通道分配。每个总线都有1个活动管理器。

通用设备是提供应用功能的设备（例如编解码器）。

帧设备负责时钟总线，并在总线上传输帧同步和帧信息。

每个SLIMbus组件都有一个用于监视物理层的接口设备。

通常，每个SoC都包含一个SLIMbus组件，其中包含一个管理器、一个帧设备、一个通用设备（用于数据通道支持）和一个接口设备。外部外围SLIMbus组件通常有1个通用设备（用于功能/数据通道支持）和一个相关的接口设备。通用设备的寄存器被映射为“值元素”，以便可以使用SLIMbus控制通道交换控制/状态类型的信息进行写入/读取。如果在同一总线上有多个帧设备，则管理器设备负责选择活动帧设备以时钟总线。

根据规范，SLIMbus使用“时钟齿轮”对基于当前频率和带宽要求的功率管理进行处理。有10个时钟齿轮，每个齿轮使SLIMbus频率变为前一个齿轮的两倍。

每个设备都有一个6字节的枚举地址，并且管理器在设备在总线上报告存在后为每个设备分配一个1字节的逻辑地址。

### 软件说明:

有两种类型的SLIMbus驱动程序：

slim\_controller表示SLIMbus的“控制器”。这个驱动程序应该实现SoC需要的职责（管理器设备、关联的接口设备以监视层并报告错误、默认帧设备）。

slim\_device 表示 SLIMbus 的“通用设备/组件”，slim\_driver 应该为该 slim\_device 实现驱动程序。

### 向驱动发送的设备通知:

因为 SLIMbus 设备有报告其存在的机制，所以框架允许驱动程序在相应设备报告其在总线上存在时进行绑定。然而，可能需要先探测驱动程序，以便它能启用相应的 SLIMbus 设备（例如，对其供电和/或使其退出重置）。为了支持这种行为，框架也允许驱动程序先进行探测（例如，使用标准的DeviceTree兼容字段）。这就需要驱动程序在设备正常运行时（即报告其存在时）知道设备已经可用。因此，当设备报告存在并被控制器分配逻辑地址时，使用 device\_up 回调函数。

类似地，当 SLIMbus 设备停机时，它们会“报告不存在”。当设备报告缺席并且其逻辑地址分配被控制器无效时， 'device\_down' 回调函数会通知驱动程序。

另一个通知“boot\_device”用于在控制器重置总线时通知 slim\_driver，这个通知允许驱动程序采取必要的步骤来启动设备，以使其在总线被重置后正常运行。

### 驱动程序和控制器APIs:

#### struct slim\_eaddr

SLIMbus设备的枚举地址

**定义**

struct slim\_eaddr {

u8 instance;

u8 dev\_index;

u16 prod\_code;

u16 manf\_id;

};

**成员**

instance

实例值

dev\_index

设备索引

prod\_code

产品代码

manf\_id

设备的制造商ID

#### enum slim\_device\_status

slim设备状态

**常量**

SLIM\_DEVICE\_STATUS\_DOWN

slim设备不存在或尚未报告

SLIM\_DEVICE\_STATUS\_UP

slim设备在总线上发布

SLIM\_DEVICE\_STATUS\_RESERVED

保留供将来使用

#### struct slim\_device

Slim设备句柄

**定义**

struct slim\_device {

struct device dev;

struct slim\_eaddr e\_addr;

struct slim\_controller \*ctrl;

enum slim\_device\_status status;

u8 laddr;

bool is\_laddr\_valid;

struct list\_head stream\_list;

spinlock\_t stream\_list\_lock;

};

**成员**

dev

设备的驱动程序模型表示

e\_addr

此设备的枚举地址

ctrl

slim控制器实例

status

slim设备状态

laddr

该设备的1字节逻辑地址。

is\_laddr\_valid

如果 laddr 有效，则表示为真

stream\_list

此设备上的流列表

stream\_list\_lock

保护流列表的锁

**说明**

当向控制器注册 SLIMbus 设备时，将返回此客户端/设备句柄。客户端驱动程序使用该结构的指针作为句柄。

#### struct slim\_driver

SLIMbus“通用设备”（从设备）设备驱动程序（类似于 SPI 上的 “ spi\_device”）

**定义**

struct slim\_driver {

int (\*probe)(struct slim\_device \*sl);

void (\*remove)(struct slim\_device \*sl);

void (\*shutdown)(struct slim\_device \*sl);

int (\*device\_status)(struct slim\_device \*sl, enum slim\_device\_status s);

struct device\_driver driver;

const struct slim\_device\_id \*id\_table;

};

**成员**

probe

将此驱动程序绑定到 SLIMbus 设备

remove

从 SLIMbus 设备中取消此驱动程序的绑定

shutdown

在断电/停机期间使用的标准关机回调

device\_status

当该设备报告其存在并被分配 laddr 时，调用此回调函数；当设备报告缺席或总线关闭时，也会调用该回调函数

driver

SLIMbus设备驱动程序应该初始化此结构的名称和所有者字段

id\_table

由该驱动程序支持的SLIMbus设备列表

#### struct slim\_val\_inf

Slimbus值或信息元素

**定义**

struct slim\_val\_inf {

u16 start\_offset;

u8 num\_bytes;

u8 \*rbuf;

const u8 \*wbuf;

struct completion \*comp;

};

**成员**

start\_offset

在信息/值元素映射中指定起始偏移量

num\_bytes

高达16，这确保消息符合 SLIMbus 规范的分片大小

rbuf

读取值的缓冲区

wbuf

写入缓冲区

comp

用于异步操作的完成，仅在需要TID进行事务时有效，例如请求操作。其他事务都是同步的

#### struct slim\_stream\_config

在音频驱动程序的 hw\_params 或 prepare 调用中进行了流配置，其中它们具有有关速率、通道数量等所需信息。每个通道和端口之间具有1:1映射。

**定义**

struct slim\_stream\_config {

unsigned int rate;

unsigned int bps;

unsigned int ch\_count;

unsigned int \*chs;

unsigned long port\_mask;

int direction;

};

**成员**

rate

数据速率

bps

数据样本每位数

ch\_count

通道数

chs

通道编号列表的指针

port\_mask

用于该流的端口的端口掩码

direction

流的方向，SNDRV\_PCM\_STREAM\_PLAYBACK or SNDRV\_PCM\_STREAM\_CAPTURE.

#### module\_slim\_driver (\_\_slim\_driver)

用于注册SLIMbus驱动程序的辅助宏

**参数**

\_\_slim\_driver

slimbus\_driver结构体

**说明**

用于 SLIMbus 驱动程序，不在模块 init/exit 中进行任何特殊处理的情况。这消除了许多样板文件。每个模块只能使用此宏一次，并且调用它将替换module\_ init() 和 module\_exit()。

#### struct slim\_framer

代表 SLIMbus 成帧器。每个控制器可能有多个成帧器。有 1 个主动成帧器设备负责为总线计时。管理者负责成帧器的移交。

**定义**

struct slim\_framer {

struct device dev;

struct slim\_eaddr e\_addr;

int rootfreq;

int superfreq;

};

成员

dev

设备的驱动模型表示

e\_addr

帧枚举地址

rootfreq

帧同步器可运行的根频率。这是总线可以运行的最大频率（“时钟齿轮10”）

superfreq

每个根频率的超级帧。每个帧都是6144位。

#### struct slim\_msg\_txn

控制器要发送的消息。此结构有数据包头、有效负载和要填充的缓冲区（如果存在）。

**定义**

struct slim\_msg\_txn {

u8 rl;

u8 mt;

u8 mc;

u8 dt;

u16 ec;

u8 tid;

u8 la;

struct slim\_val\_inf \*msg;

struct completion \*comp;

};

**成员**

rl

报头字段。剩余长度

mt

报头字段。消息类型

mc

报头字段。最低有效位是消息代码，用于类型mt

dt

报头字段。目标类型

ec

元素代码。用于元素访问API

tid

事务ID。用于期望响应的消息（适用于涉及读操作的消息代码）

la

此消息要发送到的设备的逻辑地址。（当目标类型是广播时不使用）

msg

要读取/写入的元素访问消息

comp

完成时，如果读/写是同步的，则用于tid基础事务的内部使用。

#### enum slim\_clk\_state

**常量**

SLIM\_CLK\_ACTIVE

SLIMbus时钟处于活动状态

SLIM\_CLK\_ENTERING\_PAUSE

SLIMbus时钟暂停序列正在总线上发送。如果成功，状态将更改为SLIM\_CLK\_PAUSED。如果转换失败，则状态将更改回SLIM\_CLK\_ACTIVE

SLIM\_CLK\_PAUSED

SLIMbus控制器时钟已暂停

**说明**

维护当前时钟状态

#### struct slim\_sched

**定义**

struct slim\_sched {

enum slim\_clk\_state clk\_state;

struct completion pause\_comp;

struct mutex m\_reconf;

};

**成员**

clk\_state

来自枚举slim\_clk\_state的控制器时钟状态

pause\_comp

信号时钟暂停序列完成。当客户端尝试在控制器进入时钟暂停时调用SLIMbus事务时，这很有用

m\_reconf

此互斥锁被保持，直到完成当前重新配置（数据通道调度、消息带宽预留）。当保持此互斥锁时，消息API可以并行使用总线，因为在重新配置进行时，元素访问消息可以在总线上发送

#### enum slim\_port\_direction

**常量**

SLIM\_PORT\_SINK

SLIMbus端口为接收端

SLIM\_PORT\_SOURCE

SLIMbus端口是源

#### enum slim\_port\_state

**常量**

SLIM\_PORT\_DISCONNECTED

SLIMbus 端口在 DISCONNECT\_PORT 或 REMOVE\_CHANNEL 内核命令后断开连接，从未配置/配置状态进入

SLIM\_PORT\_UNCONFIGURED

SLIMbus端口处于未配置状态。从CONNECT\_SOURCE/SINK核心命令后，从断开状态进入

SLIM\_PORT\_CONFIGURED

SLIMbus端口处于配置状态。从未配置状态经过DEFINE\_CHANNEL、DEFINE\_CONTENT和ACTIVATE\_CHANNEL核心命令后进入。准备好数据传输

**描述**

根据 SLIMbus 规范 2.0

#### enum slim\_channel\_state

**常量**

SLIM\_CH\_STATE\_DISCONNECTED

SLIMbus通道已断开连接

SLIM\_CH\_STATE\_ALLOCATED

SLIMbus通道已分配

SLIM\_CH\_STATE\_ASSOCIATED

SLIMbus通道已与端口关联

SLIM\_CH\_STATE\_DEFINED

定义了SLIMbus通道参数

SLIM\_CH\_STATE\_CONTENT\_DEFINED

定义了SLIMbus通道内容

SLIM\_CH\_STATE\_ACTIVE

SLIMbus通道处于活动状态，准备好数据

SLIM\_CH\_STATE\_REMOVED

SLIMbus通道处于不活动状态并已删除

#### enum slim\_ch\_data\_fmt

**常量**

SLIM\_CH\_DATA\_FMT\_NOT\_DEFINED

未定义

SLIM\_CH\_DATA\_FMT\_LPCM\_AUDIO

LPCM音频

SLIM\_CH\_DATA\_FMT\_IEC61937\_COMP\_AUDIO

IEC61937 压缩音频

SLIM\_CH\_DATA\_FMT\_PACKED\_PDM\_AUDIO

打包的PDM音频

**说明**

SLIMbus 规范 1.01.01 表 60

#### enum slim\_ch\_aux\_bit\_fmt

**常量**

SLIM\_CH\_AUX\_FMT\_NOT\_APPLICABLE

未定义

SLIM\_CH\_AUX\_FMT\_ZCUV\_TUNNEL\_IEC60958

用于隧道IEC60958的ZCUV

SLIM\_CH\_AUX\_FMT\_USER\_DEFINED

用户定义

**说明**

SLIMbus规范2.0的第63表

#### struct slim\_channel

用于状态机的SLIMbus通道

**定义**

struct slim\_channel {

int id;

int prrate;

int seg\_dist;

enum slim\_ch\_data\_fmt data\_fmt;

enum slim\_ch\_aux\_bit\_fmt aux\_fmt;

enum slim\_channel\_state state;

};

**成员**

id

通道ID

prrate

来自SLIMbus 2.0规范第66表的通道存在率

seg\_dist

来自SLIMbus 2.0规范第20表的段分配代码

data\_fmt

通道的数据格式

aux\_fmt

此通道的辅助格式

state

通道状态机

#### struct slim\_port

SLIMbus端口

**定义**

struct slim\_port {

int id;

enum slim\_port\_direction direction;

enum slim\_port\_state state;

struct slim\_channel ch;

};

**成员**

id

端口号

direction

端口方向，源或接收端

state

端口状态机

ch

与此端口关联的通道

#### enum slim\_transport\_protocol

**常量**

SLIM\_PROTO\_ISO

等时协议，没有流控，因为数据速率匹配通道速率，流控嵌入数据中

SLIM\_PROTO\_PUSH

推送协议，包括流控，用于携带速率等于或低于通道速率的数据

SLIM\_PROTO\_PULL

拉取协议，用法与推送协议相似，但拉取是单播

SLIM\_PROTO\_LOCKED

锁定协议

SLIM\_PROTO\_ASYNC\_SMPLX

异步协议-单向

SLIM\_PROTO\_ASYNC\_HALF\_DUP

异步协议-半双工

SLIM\_PROTO\_EXT\_SMPLX

扩展异步协议-单向

SLIM\_PROTO\_EXT\_HALF\_DUP

扩展异步协议-半双工

**说明**

SLIMbus 2.0规范第47表

#### struct slim\_stream\_runtime

SLIMbus流运行时实例

**定义**

struct slim\_stream\_runtime {

const char \*name;

struct slim\_device \*dev;

int direction;

enum slim\_transport\_protocol prot;

unsigned int rate;

unsigned int bps;

unsigned int ratem;

int num\_ports;

struct slim\_port \*ports;

struct list\_head node;

};

**成员**

name

流的名称

dev

与此流关联的SLIM设备实例

direction

流的方向

prot

在此流中使用的传输协议

rate

样本的数据率 \*

bps

每个样本的位数

ratem

速率乘数，即超帧速率/数据速率

num\_ports

端口数量

ports

指向端口实例的指针

node

与slim设备关联的流的列表头

#### struct slim\_controller

控制SLIM总线的每个实例（类似于SPI上的“主”）

**定义**

struct slim\_controller {

struct device \*dev;

unsigned int id;

char name[SLIMBUS\_NAME\_SIZE];

int min\_cg;

int max\_cg;

int clkgear;

struct ida laddr\_ida;

struct slim\_framer \*a\_framer;

struct mutex lock;

struct list\_head devices;

struct idr tid\_idr;

spinlock\_t txn\_lock;

struct slim\_sched sched;

int (\*xfer\_msg)(struct slim\_controller \*ctrl, struct slim\_msg\_txn \*tx);

int (\*set\_laddr)(struct slim\_controller \*ctrl, struct slim\_eaddr \*ea, u8 laddr);

int (\*get\_laddr)(struct slim\_controller \*ctrl, struct slim\_eaddr \*ea, u8 \*laddr);

int (\*enable\_stream)(struct slim\_stream\_runtime \*rt);

int (\*disable\_stream)(struct slim\_stream\_runtime \*rt);

int (\*wakeup)(struct slim\_controller \*ctrl);

};

**成员**

dev

驱动程序的设备接口

id

此控制器/总线的特定板号标识符

name

此控制器的名称

min\_cg

此控制器支持的最小时钟档位（默认值：1）

max\_cg

此控制器支持的最大时钟档位（默认值：10）

clkgear

此总线正在运行的当前时钟档位

laddr\_ida

逻辑地址ID分配器

a\_framer

正在运行管理此控制器的总线时钟的活动框架

lock

保护控制器数据结构的互斥量

devices

Slim设备列表

tid\_idr

tid id分配器

txn\_lock

保护事务表的锁定

sched

控制器使用的调度器结构

xfer\_msg

在此控制器上传输消息（这可以是像数据信道设置那样的广播控制/状态消息，也可以是像值元素读取/写入那样的单播消息

set\_laddr

设置逻辑地址（laddr）以与元素地址（e\_addr）匹配的从设备。实现控制器驱动程序将会期望向此设备发送单播消息，并与其逻辑地址进行通讯

get\_laddr

在某些情况下（例如，当主设备在远程处理器端时），可能需要控制器设置固定的逻辑地址表，可以使用get\_laddr进行分配。当控制器需要分配逻辑地址时，可以使用此表进行控制。此用例是由在远程处理器端的主设备负责分配逻辑地址

enable\_stream

此函数指针实现了控制器特定的启用流程

disable\_stream

此函数指针实现了控制器特定的禁用流程

wakeup

此函数指针实现了控制器特定的唤醒过程。框架将调用此函数，以从时钟暂停中将控制器唤醒

**说明**

“管理器设备”负责设备管理、带宽分配、通道设置和每个通道的端口关联。设备管理意味着基于设备的枚举（报告-存在或报告-不存在）进行逻辑地址分配/删除。带宽分配由管理器根据总线上活动通道和SLIM总线设备发出的消息带宽请求动态完成。根据当前带宽使用情况，管理器选择一个频率来运行总线（按照“时钟档位”步骤，从1到10，每个时钟档位表示前一个档位的两倍频率）。管理器还负责进入（和退出）低功耗模式（称为“时钟暂停”）。如果总线上有多个帧同步器，且某个用例需要使用特定的帧同步器以避免保持先前的帧同步器开启，则管理器可以执行帧同步器的交接。

控制器在此处执行管理员设备和“接口设备”的职责。接口设备负责监视总线并报告信息，如丢失同步和数据时隙冲突。

#### int slim\_unregister\_controller(struct [slim\_controller](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_controller" \o "slim_controller) \*ctrl)

控制器拆卸

**参数**

struct slim\_controller \*ctrl

需要拆卸的控制器

#### void slim\_report\_absent(struct [slim\_device](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_device" \o "slim_device) \*sbdev)

当设备报告不存在时，或者当无法与设备通信时，控制器调用此函数

**参数**

struct slim\_device \*sbdev

设备无法连接或发送的报告不存在

#### struct [slim\_device](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_device" \o "slim_device) \*slim\_get\_device(struct [slim\_controller](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_controller" \o "slim_controller) \*ctrl, struct [slim\_eaddr](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_eaddr" \o "slim_eaddr) \*e\_addr)

获取设备的句柄

**参数**

struct slim\_controller \*ctrl

将在其上添加/查询此设备的控制器

struct slim\_eaddr \*e\_addr

要查询的设备的枚举地址

**返回**

如果设备已经报告，则返回指向设备的指针。如果尚未枚举设备，则创建一个新设备并返回指向该设备的指针

#### struct [slim\_device](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_device" \o "slim_device) \*of\_slim\_get\_device(struct [slim\_controller](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_controller" \o "slim_controller) \*ctrl, struct device\_node \*np)

使用dt节点获取设备的句柄

**参数**

struct slim\_controller \*ctrl

将在其上添加/查询此设备的控制器

struct device\_node \*np

设备的节点指针

**返回**

如果设备已经报告，则返回指向设备的指针。如果尚未枚举设备，则创建一个新设备并返回指向该设备的指针

#### int slim\_device\_report\_present(struct [slim\_controller](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_controller" \o "slim_controller) \*ctrl, struct [slim\_eaddr](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_eaddr" \o "slim_eaddr) \*e\_addr, u8 \*laddr)

报告已经枚举的设备

**参数**

struct slim\_controller \*ctrl

枚举设备的控制器

struct slim\_eaddr \*e\_addr

设备的枚举地址

u8 \*laddr

返回逻辑地址（如果有效标志为假）

**说明**

在响应REPORT\_PRESENT时，由控制器调用。框架将为此枚举地址分配逻辑地址。函数返回-EXFULL，表示所有逻辑地址已被占用

#### int slim\_get\_logical\_addr(struct [slim\_device](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_device" \o "slim_device) \*sbdev)

获取/分配一个SLIM总线设备的逻辑地址

**参数**

struct slim\_device \*sbdev

请求地址的客户端句柄

**返回**

如果逻辑地址有效或已分配新的逻辑地址，则为零。如果出现错误，则返回错误代码

### 时钟暂停:

SLIM总线规定，在总线进入低功耗模式之前，必须向总线上的所有活动设备广播重新配置序列（称为时钟暂停）。当控制器决定进入低功耗模式时，控制器使用此序列，以便关闭相应的时钟和/或电源电路以节省电能。通过唤醒帧设备（如果控制器驱动程序启动退出低功耗模式），或者切换数据线（如果从设备想要启动它），可以退出时钟暂停。

### 时钟暂停 APIs:

#### int slim\_ctrl\_clk\_pause(struct [slim\_controller](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_controller" \o "slim_controller) \*ctrl, bool wakeup, u8 restart)

由SLIM总线控制器调用以进入/退出“时钟暂停”

**参数**

struct slim\_controller \*ctrl

请求暂停或唤醒总线的控制器

bool wakeup

唤醒此控制器退出时钟暂停

u8 restart

根据规范使用的重新启动时间值，用于时钟暂停。当要唤醒控制器时，该值不被使用

**说明**

Slimbus规范需要这个序列来关闭总线上的时钟。序列涉及发送3个广播消息（重新配置序列）以通知总线上的所有设备。要退出时钟暂停，控制器通常会唤醒活动帧设备。如果wakeup为false，则此API执行时钟暂停重新配置序列。如果wakeup为true，则调用控制器的wakeup。如果有待处理的消息txn，则返回-EBUSY以进入时钟暂停。

### 消息:

框架支持regmap和读/写api以交换与SLIM总线设备的控制信息。API可以是同步或异步的。头文件<linux/slimbus.h>有关于消息API的更多文档。

### 消息传递APIs:

#### void slim\_msg\_response(struct [slim\_controller](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_controller" \o "slim_controller) \*ctrl, u8 \*reply, u8 tid, u8 len)

将从设备收到的消息响应传递给框架

**参数**

struct slim\_controller \*ctrl

控制器句柄

u8 \*reply

从设备收到的回复

u8 tid

与之相关的框架可以关联回复的事务ID

u8 len

回复的长度

**说明**

由控制器呼叫，以通知框架收到的响应。这有助于使API异步，控制器驱动程序不需要管理除框架映射TID与缓冲区之外的其他表

#### int slim\_alloc\_txn\_tid(struct [slim\_controller](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_controller" \o "slim_controller) \*ctrl, struct [slim\_msg\_txn](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_msg_txn" \o "slim_msg_txn) \*txn)

为txn分配一个tid

**参数**

struct slim\_controller \*ctrl

控制器句柄

struct slim\_msg\_txn \*txn

要分配tid的事务

**返回**

成功时为零，具有有效txn->tid，失败时为错误代码

#### void slim\_free\_txn\_tid(struct [slim\_controller](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_controller" \o "slim_controller) \*ctrl, struct [slim\_msg\_txn](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_msg_txn" \o "slim_msg_txn) \*txn)

释放txn的tid

**参数**

struct slim\_controller \*ctrl

控制器句柄

struct slim\_msg\_txn \*txn

应释放其tid的事务

#### int slim\_do\_transfer(struct [slim\_controller](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_controller" \o "slim_controller) \*ctrl, struct [slim\_msg\_txn](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_msg_txn" \o "slim_msg_txn) \*txn)

处理一个SLIM总线消息传递事务

**参数**

struct slim\_controller \*ctrl

控制器句柄

struct slim\_msg\_txn \*txn

要通过SLIMbus发送的事务

**说明**

由控制器呼叫以传输不涉及接口/值元素的消息事务（例如，传输消息以为从设备分配逻辑地址）

**返回**

-ETIMEDOUT: 如果此消息的传输超时（例如，由于总线线路未被控制器时钟或驱动程序驱动）

#### int slim\_xfer\_msg(struct [slim\_device](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_device" \o "slim_device) \*sbdev, struct [slim\_val\_inf](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_val_inf" \o "slim_val_inf) \*msg, u8 mc)

在slim设备上传输价值信息消息

**参数**

struct slim\_device \*sbdev

要将此消息传输到的Slim设备

struct slim\_val\_inf \*msg

值信息消息指针

u8 mc

消息的消息代码

**说明**

由想要传输值或信息元素的驱动程序调用

**返回**

-ETIMEDOUT: 如果此消息的传输超时

#### int slim\_read(struct [slim\_device](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_device" \o "slim_device) \*sdev, u32 addr, size\_t count, u8 \*val)

读取SLIM总线值元素

**参数**

struct slim\_device \*sdev

客户端句柄

u32 addr

要读取的值元素的地址

size\_t count

要读取的字节数。允许的最大字节数为16

u8 \*val

将返回价值元素的值

**返回**

对于无效参数，返回-EINVAL，如果传输此消息超时，则返回-ETIMEDOUT（例如，由于总线线路未被控制器时钟或驱动程序驱动）

#### int slim\_readb(struct [slim\_device](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_device" \o "slim_device) \*sdev, u32 addr)

从SLIM值元素中读取一个字节

**参数**

struct slim\_device \*sdev

客户端句柄

u32 addr

要读取的值元素中的地址

**返回**

值元素的字节值

#### int slim\_write(struct [slim\_device](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_device" \o "slim_device) \*sdev, u32 addr, size\_t count, u8 \*val)

Write SLIMbus value element

**参数**

struct slim\_device \*sdev

客户端句柄

u32 addr

要写入的值元素中的地址

size\_t count

要写入的字节数。最大允许字节数为16

u8 \*val

要写入值元素的值

**返回**

-EINVAL表示参数无效，-ETIMEDOUT表示传输此消息超时（例如，由于总线线路未被控制器时钟或驱动而超时)

#### int slim\_writeb(struct [slim\_device](https://www.kernel.org/doc/html/latest/driver-api/slimbus.html" \l "c.slim_device" \o "slim_device) \*sdev, u32 addr, u8 value)

向SLIMbus值元素写入字节

**参数**

struct slim\_device \*sdev

客户端句柄

u32 addr

要写入值元素的地址

u8 value

要写入值元素的值

**返回**

-EINVAL表示参数无效，-ETIMEDOUT表示传输此消息超时（例如，由于总线线路未被控制器时钟或驱动而超时)

### 流APIs

# SoundWire文档

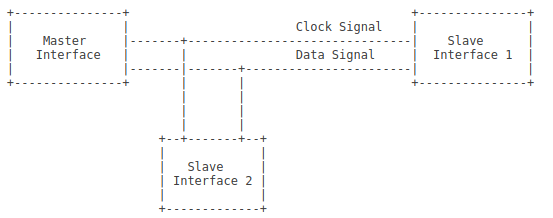
## SoundWire子系统概述

SoundWire是由MIPI联盟于2015年批准的一种新接口。SoundWire用于传输通常与音频功能相关的数据。 SoundWire接口经过优化，可将音频设备集成到移动或移动启发式系统中。

SoundWire是一种具有数据和时钟线的2引脚多点接口。它便于开发低成本，高效，高性能的系统。 SoundWire接口的广泛关键特征包括：

1. 通过单个双针接口传输所有有效负载数据通道，控制信息和设置命令。
2. 通过使用DDR（双数据速率）数据传输降低时钟频率，从而降低功耗。
3. 时钟缩放和可选的多个数据通道可为系统提供广泛的数据速率灵活性
4. 设备状态监视，包括向主控制器发送中断样式警报。

SoundWire协议支持最多11个从接口。所有接口共享包含数据和时钟线的公共总线。每个从站最多支持14个数据端口。 13个数据端口专用于音频传输。 Data Port0专用于传输批量控制信息，每个音频数据端口（1..14）都可以支持最多8个通道在发送或接收模式下（通常是固定方向，但由规范启用可配置方向）。由于带宽限制在约19.2..24.576Mbits / s，因此无法同时传输11 \* 13 \* 8个通道。下图显示了SoundWire主控和两个从设备之间的连接示例。



### 术语

MIPI SoundWire规范使用术语“设备”来指称主控制器或从接口，这显然会令人困惑。在本概述和代码中，我们仅使用术语接口来指代硬件。我们按照Linux设备模型，将连接在总线上的每个从接口映射为由特定驱动程序管理的设备。 Linux SoundWire子系统提供了一个框架，以实现具有API的SoundWire从驱动程序，允许第三方供应商启用实现定义的功能，而通用的设置/配置任务由总线处理。

总线：实现处理SoundWire协议的SoundWire Linux总线。编程所有MIPI定义的从站寄存器。代表SoundWire主控。系统中可以存在多个总线实例。

从站slave：注册为SoundWire从设备（Linux设备）。多个从站设备可以注册到总线实例。

从站驱动程序Slave driver：控制从站设备的驱动程序。总线直接控制MIPI指定的寄存器（并通过主控驱动程序/接口传输）。任何实现定义的从站寄存器由从站驱动程序控制。实际上，预计从站驱动程序依赖于regmap并且不请求直接寄存器访问。

### 编程接口（SoundWire主控接口驱动程序）

SoundWire总线支持SoundWire主控实现和SoundWire从设备的编程接口。所有代码使用SoC设计人员和第三方供应商通常使用的“sdw”前缀。

每个SoundWire主控接口都需要注册到总线上。总线实现API以读取标准主控MIPI属性，并为主控操作提供回调以实现其自己的提供功能信息的功能。目前尚未实现DT支持，但应该很容易添加，因为设备\_property\_ API启用了功能。

根据板文件，DT或ACPI注册主控接口以及主控接口功能。

以下是用于注册SoundWire总线的总线API:

int sdw\_add\_bus\_master(struct sdw\_bus \*bus){

if (!bus->dev)

return -ENODEV;

mutex\_init(&bus->lock);

INIT\_LIST\_HEAD(&bus->slaves);

/\* Check ACPI for Slave devices \*/

sdw\_acpi\_find\_slaves(bus);

/\* Check DT for Slave devices \*/

sdw\_of\_find\_slaves(bus);

return 0;

}

这将初始化sdw\_bus对象的主设备。提供“sdw\_master\_ops”和“sdw\_master\_port\_ops”回调函数以处理总线。

“sdw\_master\_ops”用于以硬件特定的方式控制总线。它包括总线控制函数，如在总线上传递SoundWire读/写消息，设置时钟频率和流同步点（SSP）。 “sdw\_master\_ops”结构在总线中抽象了主节点的硬件细节。

“sdw\_master\_port\_ops”用于设置主接口端口的端口参数。由MIPI规范未定义主接口端口寄存器映射，因此总线调用“sdw\_master\_port\_ops”回调函数进行端口操作，如“Port Prepare”，“Port Transport params set”，“Port enable and disable”等。然后主驱动程序的实现可以执行硬件特定的配置。

### 编程接口（SoundWire从驱动程序）

MIPI规范要求每个Slave接口公开存储在6个只读dev\_id寄存器中的唯一48位标识符。此dev\_id标识符包含供应商和零件信息，以及使其与相同组件区分开的字段。另外，一个类字段目前未使用。从驱动程序写入特定供应商和部件标识符，总线根据这两个标识符枚举从设备。基于这两个标识符的设备和驱动程序匹配。在成功匹配设备和驱动程序ID之间，总线会调用从驱动程序的Probe。主/从设备之间强制执行父/子关系（逻辑表示与物理连通性对齐）。

主/ 从依赖关系的信息存储在平台数据，板文件，ACPI或DT中。MIPI软件规范为具有多个主接口的控制器定义了其他link\_id参数。在链路范围内，dev\_id寄存器仅是唯一的，并且link\_id在控制器范围内是唯一的。dev\_id和link\_id在系统级别上不一定是唯一的，但父/子信息用于避免歧义。

static const struct sdw\_device\_id slave\_id[] = {

SDW\_SLAVE\_ENTRY(0x025d, 0x700, 0),

{},

};

MODULE\_DEVICE\_TABLE(sdw, slave\_id);

static struct sdw\_driver slave\_sdw\_driver = {

.driver = {

.name = "slave\_xxx",

.pm = &slave\_runtime\_pm,

},

.probe = slave\_sdw\_probe,

.remove = slave\_sdw\_remove,

.ops = &slave\_slave\_ops,

.id\_table = slave\_id,

};

对于功能，总线实现了API来读取标准Slave MIPI属性，并在从操作中提供回调以使Slave驱动程序实现自己的提供功能信息的功能。总线需要知道一组从能力来程序从寄存器并控制总线重新配置。

### 未来要做的改进

批量寄存器访问（BRA）传输。

多数据通道支持。

### 链接

SoundWire MIPI规范1.1可在以下位置找到：<https://members.mipi.org/wg/All-Members/document/70290>

SoundWire MIPI DisCo（发现和配置）规范可在以下位置找到：<https://www.mipi.org/specifications/mipi-disco-soundwire（注册后可公开访问或MIPI成员可直接访问）>

MIPI 联盟制造商ID页面：mid.mipi.org

## SoundWire中的音频流

音频流是在之间创建的逻辑或虚拟连接

1. 系统内存缓冲区和编解码器之间
2. DSP内存缓冲区和编解码器之间
3. FIFO和编解码器之间
4. 编解码器和编解码器之间

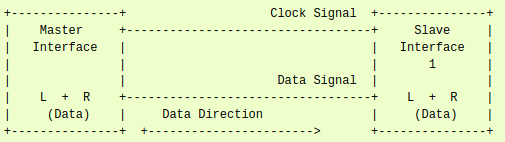
通常通过数据链路驱动的DMA通道，音频流包含一个或多个数据通道。流中的所有通道必须具有相同的采样率和相同的采样大小的数据通道。假设使用SoundWire接口打开带有两个通道（左和右）的流。下面是在SoundWire中表示流的一些方式。

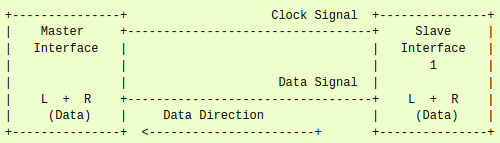
内存中的样本流（系统存储器，DSP存储器或FIFO）

-------------------------

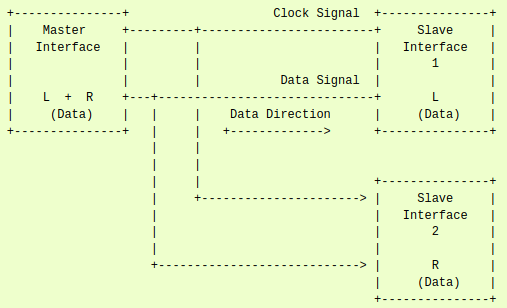
| L | R | L | R | L | R |

-------------------------

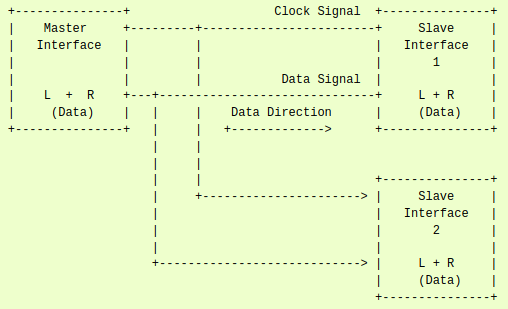
示例1：使用单个端口从主设备呈现带有L和R通道的立体声流到从设备。

示例2：使用单个端口从从设备捕获带有L和R通道的立体声流到主设备。

示例3：由主节点呈现带有L和R通道的立体声流。每个L和R通道由两个不同的从设备接收。主设备和两个从设备都使用单个端口。



示例4：由主节点呈现带有L和R通道的立体声流。两个L和R通道都由两个不同的从设备接收。主设备和两个从设备都使用单个处理L + R的端口。每个从设备基于静态配置或动态方向通常会在本地处理L + R数据并驱动一个或多个扬声器。



### SoundWire流管理流程

#### 流定义

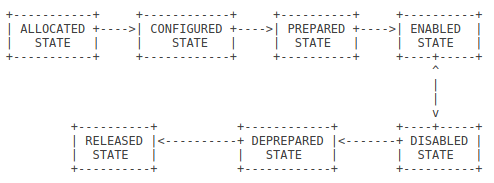
当前流：这被分类为必须执行操作的流，如准备，启用，禁用，取消准备等

活动流：这被分类为除当前流之外已经在总线上活动的流。可以在总线上有多个活动流。

SoundWire总线管理SoundWire总线上每个渲染/捕获流的流操作。本部分解释了总线为总线上分配/释放的每个流执行的操作。以下是总线为每个音频流维护的流状态。

#### SoundWire流状态

以下显示了SoundWire流状态和状态转换图。



注意：规范支持准备和去准备之间的状态转换，但在软件（子系统）中不支持。

注意2：流状态转换检查需要由调用框架（例如ALSA / ASoC）处理。 SoundWire子系统中不存在流转换的检查。

#### 流状态操作

以下部分解释总线在主从上执行的操作，作为流状态转换的一部分。

#### SDW\_STREAM\_**A**LLOCATED

流的分配状态。这是流的入口状态。进入此状态之前执行以下操作：

1. 为流分配流运行时。该流运行时用作对流执行的所有操作的参考。
2. 分配并初始化保存流运行时信息所需的资源。其中包含所有与流相关的信息，例如流类型（PCM / PDM）和参数，与流关联的主和从接口，流状态等。

完成上述所有操作后，将流状态设置为SDW\_STREAM\_ALLOCATED。

总线为分配流实现以下API，需要针对每个流调用一次。从ASoC DPCM框架来看，此流状态可能与.startup（）操作关联。

int sdw\_alloc\_stream(char \* stream\_name);

#### SDW\_STREAM\_CONFIGURED

流的配置状态。在进入此状态之前执行以下操作：

1. 在SDW\_STREAM\_ALLOCATED状态下为流信息分配的资源在此处得到更新。这包括与当前流相关的流参数、主、从运行时信息。
2. 所有与当前流相关的主和从提供端口信息给总线，其中包括主和从为当前流分配的端口号及其通道掩码。

完成所有上述操作后，将流状态设置为SDW\_STREAM\_CONFIGURED。

总线为CONFIG状态实现以下API，这些API需要由与流相关的相应主和从调用，且只能由相应的主和从调用一次。从ASoC DPCM框架来看，此流状态与.hw\_params（）操作相关联

int sdw\_stream\_add\_master(struct sdw\_bus \* bus,

struct sdw\_stream\_config \* stream\_config,

struct sdw\_ports\_config \* ports\_config,

struct sdw\_stream\_runtime \* stream);

int sdw\_stream\_add\_slave(struct sdw\_slave \* slave,

struct sdw\_stream\_config \* stream\_config,

struct sdw\_ports\_config \* ports\_config,

struct sdw\_stream\_runtime \* stream);

#### SDW\_STREAM\_PREPARED

流的准备状态。在进入此状态之前执行以下操作：

1. 带宽、帧形状、时钟频率等总线参数是根据当前流和已经在总线上活动的流计算的。需要重新计算以适应总线上的当前流。
2. 基于步骤1中计算的帧形状和时钟频率，为当前流和已经激活的所有流计算所有主从端口的传输和端口参数。
3. 将计算的总线和传输参数编程到主和从寄存器中。banked寄存器编程是在备用bank（当前未使用的bank）上完成的。在备用bank上启用已经激活的流的端口。这样做是为了不干扰已经激活的流。
4. 一旦所有值都被编程，总线启动切换到备用bank，其中所有新编程的值发挥作用。
5. 通过编程PrepareCtrl寄存器准备当前流的主从端口。

完成所有上述操作后，将流状态设置为SDW\_STREAM\_PREPARED。

总线为PREPARE状态实现以下API，需要针对每个流调用一次。从ASoC DPCM框架来看，此流状态与.prepare（）操作相关联。

int sdw\_prepare\_stream(struct sdw\_stream\_runtime \* stream);

#### SDW\_STREAM\_ENABLED

流的启用状态。在进入此状态时，数据端口被启用。在进入此状态之前执行以下操作：

1. 将在SDW\_STREAM\_PREPARED状态下计算的所有值编进备用bank（当前未使用的bank）中。它包括对已经激活的流的编程。
2. 通过对ChannelEn寄存器编程，在备用bank（当前未使用的bank）上启用当前流的所有主从端口。
3. 一旦所有值都被编程，总线启动切换到备用bank，其中所有新编程的值发挥作用，并启用与当前流相关联的端口。

在以上所有操作都成功之后，流状态被设置为SDW\_STREAM\_ENABLED

总线为ENABLE状态实现以下的API，需要每个流调用一次。根据ASoC DPCM框架来看，这个流状态与.trigger()的启动操作相关联。

int sdw\_enable\_stream(struct sdw\_stream\_runtime \* stream);

#### SDW\_STREAM\_DISABLED

流的禁用状态。离开这个状态后数据端口将被禁用。进入该状态前执行以下操作：

1. 通过编程通道（ChannelEn）寄存器，当前流的主、从端口上在备用bank(当前未使用的bank)上禁用。
2. 将总线和活动流的当前配置编程到备用bank(当前未使用的bank)。
3. 编程完所有值后，总线启动切换到备用bank，在那里所有新编程的值生效，并且与当前流相关的端口将被禁用。

在所有以上操作都成功之后，流状态被设置为SDW\_STREAM\_DISABLED。

总线实现以下API用于DISABLED状态，需要每个流调用一次。根据ASoC DPCM框架，这个流状态与.trigger()的停止操作相关联。

int sdw\_disable\_stream(struct sdw\_stream\_runtime \* stream);

#### SDW\_STREAM\_DEPREPARED

流的取消准备状态。进入该状态前执行以下操作：

1. 通过编程准备控制寄存器取消当前流的所有主、从端口的准备。
2. 将当前流的有效载荷带宽从总线的总带宽要求中减少，并通过执行bank切换等操作来计算和应用新参数。

在所有以上操作都成功之后，流状态被设置为SDW\_STREAM\_DEPREPARED。

总线实现以下API用于DEPREPARED状态，需要每个流调用一次。根据ASoC DPCM框架，这个流状态与.trigger()的停止操作相关联。

int sdw\_deprepare\_stream(struct sdw\_stream\_runtime \* stream);

#### SDW\_STREAM\_RELEASED

流的释放状态。进入该状态前执行以下操作：

1. 释放与当前流相关联的所有主、从端口的端口资源。
2. 释放与当前流相关联的所有主、从控运行时资源。
3. 释放与当前流相关联的流运行时资源。

在所有以上操作都成功之后，流状态被设置为SDW\_STREAM\_RELEASED。

总线实现以下API用于RELEASE状态，所有与流相关联的主、从控都需要调用此API。根据ASoC DPCM框架，这个流状态与.hw\_free()操作相关联。

int sdw\_stream\_remove\_master(struct sdw\_bus \* bus,

struct sdw\_stream\_runtime \* stream);

int sdw\_stream\_remove\_slave(struct sdw\_slave \* slave,

struct sdw\_stream\_runtime \* stream);

.shutdown（）ASoC DPCM操作在总线API下面调用，以释放分配为ALLOCATED状态的一部分的流。

在.shutdown()中维护流状态的数据结构将被释放。

void sdw\_release\_stream(struct sdw\_stream\_runtime \* stream);

#### 不支持

具有多个通道支持的单个端口不支持在两个流之间或跨流之间使用。例如，具有4个通道的端口不能用于处理2个独立的立体声流，即使在SoundWire理论上是可能的。

## SoundWire 错误处理

SoundWire物理层经过精心设计，总线上的错误很少发生，如果发生错误，应该限制在单个位错误范围内。这种设计的例子可以在同步机制(两次错误后同步丢失)以及用于批量寄存器访问的短CRC中找到。

错误可以通过多种机制检测到：

1. 总线冲突或奇偶校验错误：该机制依赖于独立于有效载荷和用法的低级别检测器，并且涵盖控制和音频数据。当前实现仅记录此类错误。改进可以是使整个编程序列无效并从已知位置重新启动。在控制/命令序列以外发生这样的错误时，SoundWire协议不支持音频数据的掩盖或恢复，错误的位置也会影响其可听性(PCM中最高有效位将更受影响)，并且在检测到一定数量的这样的错误之后，总线可能会被重置。请注意，由于编程错误(两个流使用相同的位插槽)或在发送/接收转换期间的电气问题引起的总线冲突无法区分，虽然在启用音频的情况下发生反复的总线冲突是总线分配问题的指示。中断机制也可以帮助识别检测到总线冲突或奇偶校验错误的从控，但它们可能不负责错误，因此单独重置它们并不是可行的恢复策略。
2. 命令状态：每个命令都与一个状态相关联，该状态仅涵盖设备之间的数据传输。应答状态表示命令已收到，并将在当前帧结束时执行。NAK表示命令有错误，并且不会被执行。在出现错误编程(发送到不存在的从控或未实现的寄存器)或电气问题的情况下，没有响应表明该命令被忽略了。一些主控实现允许多次重传命令。如果重传失败，回溯并重新启动整个编程序列可能是一种解决方案。或者，一些实现可能直接发出总线复位并重新枚举所有设备。
3. 超时：在一些情况下（如通道准备或时钟停止准备），总线驱动程序应该轮询一个寄存器字段，直到其过渡到零的未完成值。MIPI SoundWire 1.1 规范没有定义超时，但 MIPI SoundWire DisCo 文档为超时提出了建议。如果这样的配置不能完成，驱动程序将返回 -ETIMEOUT。这些超时是故障从设备的标志，并且可能无法恢复。

在全局重新配置序列期间出现的错误非常难以处理：

BankSwitch：在发出 BankSwitch 的最后一个命令期间出现错误很难回溯。在单个段设置中重新传输银行切换命令可能是可能的，但是当启用多个总线段（具有帧重新配置等副作用的命令会在不同的时间处理）时，这可能会导致同步问题。全局硬重置可能是最佳解决方案。

请注意，SoundWire 不提供检测在有效寄存器中写入非法值的机制。在某些情况下，标准甚至提到从设备可能以实现定义的方式行为。总线实现不提供此类错误的恢复机制，从设备或主驱动程序实现者负责在有效寄存器中编写有效值，并在需要时实现其他范围检查。

## SoundWire锁

本文档解释了 SoundWire总线的锁机制。总线使用以下锁以避免在共享资源上的总线操作中出现竞争条件。

1. 总线锁
2. 消息锁

### 总线锁

SoundWire 总线锁是一个互斥锁，并且是总线数据结构（sdw\_bus）的一部分，它用于每个总线实例。该锁用于序列化 SoundWire 总线实例中的以下操作。

1. 增加和删除从设备，更改从设备状态
2. 准备、启用、禁用和清除流操作。
3. 访问流数据结构。

### 消息锁

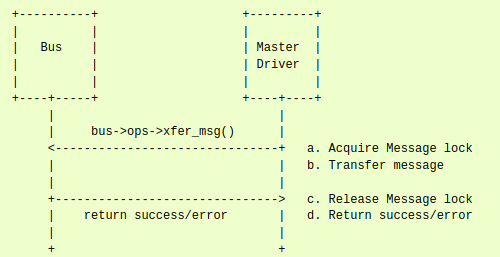
SoundWire 消息传输锁。该互斥体是总线数据结构（sdw\_bus）的一部分。该锁用于序列化 SoundWire 总线实例中的消息传输（读/写）。

下面的例子显示如何获取锁。

#### 示例1

消息传输。

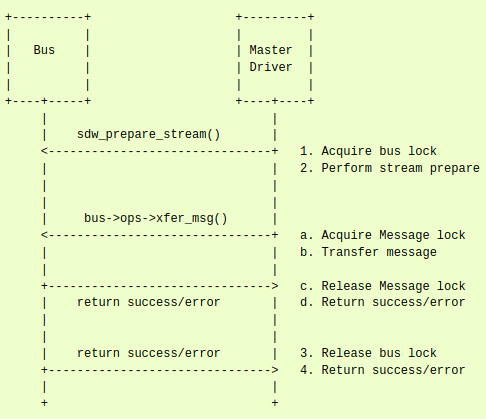
1. 对于每个消息传输：
2. 获取消息锁。
3. 将消息（读/写）传输到 从1 或在bank切换的情况下在总线上广播消息。
4. 释放消息锁。



#### 示例2

准备操作。

1. 获取与主机1关联的总线实例的锁。
2. 对于准备操作中的每个消息传输。
3. 获取消息锁。
4. 将消息（读/写）传输到 从1 或在bank切换的情况下在总线上广播消息。
5. 释放消息锁。
6. 释放与主机1关联的总线实例的锁。



# FPGA子系统

## 介绍

FPGA 子系统支持在 Linux 下动态重新编程 FPGA。FPGA 子系统的一些核心意图是：

FPGA 子系统不受供应商限制。

FPGA 子系统将上层（用户空间接口和枚举）与具有编程特定 FPGA 知识的下层分开。

代码不应在上下层之间共享。这应该是理所当然的。如果似乎有必要这样做，那么可能会添加一些框架功能，这将使其他用户受益。编写 Linux-fpga 邮件列表和维护者，并寻求扩展框架以进行广泛重用的解决方案。

一般而言，在添加代码时，请考虑未来。计划重用。

内核中的框架分为:

### FPGA管理器

如果您正在添加新的 FPGA 或新的 FPGA 编程方法，则适用于此子系统。低级 FPGA 管理器驱动程序包含如何编程特定设备的知识。该子系统包括 fpga-mgr.c 中的框架和向其注册的低级驱动程序。

### FPGA桥

FPGA 桥可防止在编程期间从 FPGA 或 FPGA 的某个区域发送杂波信号。在编程开始之前禁用它们，之后重新启用。FPGA 桥可能是实际的硬件，用于将总线连接到 CPU，或者是 FPGA 结构中围绕FPGA部分重新配置区域的软“冻结”桥。该子系统包括 fpga-bridge.C 和向其注册的低级驱动程序。

### FPGA区域

如果您要向 FPGA 框架添加新接口，请将其添加到 FPGA 区域顶部，以允许最大程度地重用您的接口。

FPGA 区域框架（fpga-region.c）将管理器和桥关联为可重新配置的区域。该区域可以指整个 FPGA 全面重新配置区域，也可以指部分重新配置区域。  
设备树 FPGA 区域支持（of-fpga-region.c）在应用设备树叠加时处理对 FPGA的重新编程。

## FPGA管理器

### 概述

FPGA 管理器核心提供了一组功能，用于使用映像编程 FPGA。API 是跨制造商的。所有制造商的特性被隐藏到低级别驱动程序中，该驱动程序在核心中注册一组操作。FPGA 映像数据本身是制造商特定的，但对于我们的目的而言，它只是二进制数据。FPGA 管理器核心不会解析它。

要编写程序的 FPGA 映像可以是散点收集列表、单个连续缓冲区或固件文件。因为应该避免为缓冲区分配连续的内核内存，因此鼓励用户使用散点收集列表。

编程映像的详细信息以结构体（struct fpga\_image\_info）呈现。该结构包含诸如 FPGA 映像的指针以及映像特定的详细信息，例如映像是为完全重新配置还是部分重新配置而构建的。

### 如何支持新的 FPGA 设备

要添加另一个 FPGA 管理器，请编写一个实现一组操作的驱动程序。探测函数调用 fpga\_mgr\_register()，例如:

static const struct fpga\_manager\_ops socfpga\_fpga\_ops = {

.write\_init = socfpga\_fpga\_ops\_configure\_init,

.write = socfpga\_fpga\_ops\_configure\_write,

.write\_complete = socfpga\_fpga\_ops\_configure\_complete,

.state = socfpga\_fpga\_ops\_state,

};

static int socfpga\_fpga\_probe(struct platform\_device \*pdev)

{

struct device \*dev = &pdev->dev;

struct socfpga\_fpga\_priv \*priv;

struct fpga\_manager \*mgr;

int ret;

priv = devm\_kzalloc(dev, sizeof(\*priv), GFP\_KERNEL);

if (!priv)

return -ENOMEM;

/\*

\* do ioremaps, get interrupts, etc. and save

\* them in priv

\*/

mgr = fpga\_mgr\_create(dev, "Altera SOCFPGA FPGA Manager",

&socfpga\_fpga\_ops, priv);

if (!mgr)

return -ENOMEM;

platform\_set\_drvdata(pdev, mgr);

ret = fpga\_mgr\_register(mgr);

if (ret)

fpga\_mgr\_free(mgr);

return ret;

}

static int socfpga\_fpga\_remove(struct platform\_device \*pdev)

{

struct fpga\_manager \*mgr = platform\_get\_drvdata(pdev);

fpga\_mgr\_unregister(mgr);

return 0;

}

ops 将实现为该特定 FPGA 执行编程序列所需的任何设备特定寄存器写入。这些 ops 成功返回 0，否则返回负错误代码。

编程序列为::

.write\_init

.write 或 .write\_sg (可能会被调用一次或多次)

.write\_complete

.write\_init 函数将准备 FPGA 接收映像数据。传递到 .write\_init 的缓冲区至少为 info->header\_size 字节长。如果整个比特流不可用，则核心代码会在开始之前至少缓存这么多。

.write 函数将缓冲区写入 FPGA。缓冲区可以包含整个 FPGA 映像或是 FPGA 映像的较小块。在后一种情况下，此函数将被多次调用以用于连续块的后续块。此接口适用 于PIO 的驱动程序。

write\_sg 版本的行为与 .write 相同，只是输入为 sg\_table 散点收集列表。此接口适用 于 DMA 的驱动程序。

在写入所有映像后调用 .write\_complete 函数，使 FPGA 进入工作模式  
ops包括一个 .state 函数，该函数将确定 FPGA 所处的状态并返回类型为 enum fpga\_mgr\_states 的代码。它不会导致状态发生变化。

### 如何将映像缓冲区编写到受支持的 FPGA

一些示例代码：

#include <linux/fpga/fpga-mgr.h>

struct fpga\_manager \*mgr;

struct fpga\_image\_info \*info;

int ret;

/\*

\* Get a reference to FPGA manager. The manager is not locked, so you can

\* hold onto this reference without it preventing programming.

\*

\* This example uses the device node of the manager. Alternatively, use

\* fpga\_mgr\_get(dev) instead if you have the device.

\*/

mgr = of\_fpga\_mgr\_get(mgr\_node);

/\* struct with information about the FPGA image to program. \*/

info = fpga\_image\_info\_alloc(dev);

/\* flags indicates whether to do full or partial reconfiguration \*/

info->flags = FPGA\_MGR\_PARTIAL\_RECONFIG;

/\*

\* At this point, indicate where the image is. This is pseudo-code; you're

\* going to use one of these three.

\*/

if (image is in a scatter gather table) {

info->sgt = [your scatter gather table]

} else if (image is in a buffer) {

info->buf = [your image buffer]

info->count = [image buffer size]

} else if (image is in a firmware file) {

info->firmware\_name = devm\_kstrdup(dev, firmware\_name, GFP\_KERNEL);

}

/\* Get exclusive control of FPGA manager \*/

ret = fpga\_mgr\_lock(mgr);

/\* Load the buffer to the FPGA \*/

ret = fpga\_mgr\_buf\_load(mgr, &info, buf, count);

/\* Release the FPGA manager \*/

fpga\_mgr\_unlock(mgr);

fpga\_mgr\_put(mgr);

/\* Deallocate the image info if you're done with it \*/

fpga\_image\_info\_free(info);

### 实现新的FPGA管理器驱动程序的 API

#### struct fpga\_manager

fpga管理器结构

**定义**

struct fpga\_manager {

const char \*name;

struct device dev;

struct mutex ref\_mutex;

enum fpga\_mgr\_states state;

struct fpga\_compat\_id \*compat\_id;

const struct fpga\_manager\_ops \*mops;

void \*priv;

};

**成员**

name

低级别 FPGA 管理器名称

dev

FPGA 管理器设备

ref\_mutex

只允许对FPGA管理器进行一次引用

state

FPGA 管理器状态

compat\_id

用于兼容性检查的 FPGA 管理器 ID

mops

指向 FPGA 管理器操作的结构体指针

priv

底层驱动程序的私有数据

#### struct **fpga\_manager\_ops**

底层 FPGA 管理器驱动程序的操作

**定义**

struct fpga\_manager\_ops {

size\_t initial\_header\_size;

enum fpga\_mgr\_states (\*state)(struct fpga\_manager \*mgr);

u64 (\*status)(struct fpga\_manager \*mgr);

int (\*write\_init)(struct fpga\_manager \*mgr,struct fpga\_image\_info \*info, const char \*buf, size\_t count);

int (\*write)(struct fpga\_manager \*mgr, const char \*buf, size\_t count);

int (\*write\_sg)(struct fpga\_manager \*mgr, struct sg\_table \*sgt);

int (\*write\_complete)(struct fpga\_manager \*mgr, struct fpga\_image\_info \*info);

void (\*fpga\_remove)(struct fpga\_manager \*mgr);

const struct attribute\_group \*\*groups;

};

**成员**

initial\_header\_size

应该传递给 write\_init 的最大字节数

state

返回 FPGA 状态的枚举值

status

返回 FPGA 的状态，包括重配置错误代码

write\_init

准备 FPGA 接收配置数据

write

向 FPGA 写入 count 字节的配置数据

write\_sg

向 FPGA 写入配置数据的散列列表

write\_complete

在写入完成后将 FPGA 设置为操作状态

fpga\_remove

可选：在驱动程序删除期间将 FPGA 设置为特定状态

groups

可选属性组

**说明**

fpga\_manager\_ops 是特定 fpga 管理器驱动程序实现的底层函数。在调用它们之前，可选项会被测试为 NULL，所以省略它们是可以的。

#### struct [fpga\_manager](https://www.kernel.org/doc/html/v4.19/driver-api/fpga/fpga-mgr.html" \l "c.fpga_manager" \o "fpga_manager) \* fpga\_mgr\_create(struct [device](https://www.kernel.org/doc/html/v4.19/driver-api/infrastructure.html" \l "c.device" \o "device) \* dev, const char \* name, const struct [fpga\_manager\_ops](https://www.kernel.org/doc/html/v4.19/driver-api/fpga/fpga-mgr.html" \l "c.fpga_manager_ops" \o "fpga_manager_ops) \* mops, void \* priv)

创建和初始化 FPGA 管理器结构

**参数**

struct device \* dev

pdev 中的 fpga 管理器设备

const char \* name

fpga 管理器名称

const struct fpga\_manager\_ops \* mops

指向 FPGA 管理器操作的结构体指针

void \* priv

ffpga 管理器私有数据

**返回**

指向 struct fpga\_manager 的指针或 NULL

#### void fpga\_mgr\_free(struct fpga\_manager \* mgr)

释放 FPGA 管理器

**参数**

struct fpga\_manager \* mgr

由 fpga\_mgr\_create 创建的 fpga 管理器结构

#### int fpga\_mgr\_register(struct fpga\_manager \* mgr)

注册 FPGA 管理器

**参数**

struct fpga\_manager \* mgr

由 fpga\_mgr\_create 创建的 fpga 管理器结构

**返回**

成功返回 0，否则返回负错误代码

#### void fpga\_mgr\_unregister(struct [fpga\_manager](https://www.kernel.org/doc/html/latest/driver-api/fpga/fpga-mgr.html" \l "c.fpga_manager" \o "fpga_manager) \*mgr)

注销并释放FPGA管理器

**参数**

struct fpga\_manager \*mgr

fpga 管理器结构

### 编程 FPGA 的 API

FPGA 管理器标志

fpga\_image\_info->flags 字段中使用的标志

FPGA\_MGR\_PARTIAL\_RECONFIG:如果支持，则进行部分重配

FPGA\_MGR\_EXTERNAL\_CONFIG: FPGA 在 Linux 引导之前已被配置

FPGA\_MGR\_ENCRYPTED\_BITSTREAM:指示位流已加密

FPGA\_MGR\_BITSTREAM\_LSB\_FIRST:  SPI位流顺序是 LSB 在前

FPGA\_MGR\_COMPRESSED\_BITSTREAM:FPGA 位流已压缩

#### struct fpga\_image\_info

特定于 FPGA 映像的信息

**定义**

struct fpga\_image\_info {

u32 flags;

u32 enable\_timeout\_us;

u32 disable\_timeout\_us;

u32 config\_complete\_timeout\_us;

char \*firmware\_name;

struct sg\_table \*sgt;

const char \*buf;

size\_t count;

int region\_id;

struct device \*dev;

#ifdef CONFIG\_OF;

struct device\_node \*overlay;

#endif;

};

**成员**

flags

如上所定义的布尔标志

enable\_timeout\_us

启用通过桥的流量的最大时间（微秒）

disable\_timeout\_us

禁用通过桥的流量的最大时间（微秒）

config\_complete\_timeout\_us

FPGA 切换到写入完成操作中的操作状态所需的最大时间

firmware\_name

FPGA 映像固件文件的名称

sgt

包含 FPGA 映像的分散/聚集表

buf

包含 FPGA 映像的连续缓冲区

count

缓冲区的大小

region\_id

目标区域的 ID

dev

拥有此设备的设备

overlay

设备树覆盖

#### enum fpga\_mgr\_states

FPGA 框架状态

**常量**

FPGA\_MGR\_STATE\_UNKNOWN

无法确定状态

FPGA\_MGR\_STATE\_POWER\_OFF

FPGA 电源已关闭

FPGA\_MGR\_STATE\_POWER\_UP

FPGA 报告电源已打开

FPGA\_MGR\_STATE\_RESET

FPGA 复位状态

FPGA\_MGR\_STATE\_FIRMWARE\_REQ

正在进行固件请求

FPGA\_MGR\_STATE\_FIRMWARE\_REQ\_ERR

固件请求失败

FPGA\_MGR\_STATE\_WRITE\_INIT

为编程准备 FPGA

FPGA\_MGR\_STATE\_WRITE\_INIT\_ERR

WRITE\_INIT 阶段出错

FPGA\_MGR\_STATE\_WRITE

向 FPGA 写入映像

FPGA\_MGR\_STATE\_WRITE\_ERR

写入 FPGA 时出错

FPGA\_MGR\_STATE\_WRITE\_COMPLETE

完成编程步骤

FPGA\_MGR\_STATE\_WRITE\_COMPLETE\_ERR

在 WRITE\_COMPLETE 期间发生错误

FPGA\_MGR\_STATE\_OPERATING

FPGA 已编程并正在运行

#### struct fpga\_image\_info \* fpga\_image\_info\_alloc(struct device \* dev)

分配 FPGA 映像信息结构

**参数**

struct device \* dev

拥有此设备的设备

**返回**

struct fpga\_image\_info或NULL

#### void fpga\_image\_info\_free(struct fpga\_image\_info \* info)

释放 FPGA 映像信息结构

**参数**

struct fpga\_image\_info \* info

要释放的 FPGA 映像信息结构

#### struct fpga\_manager \* of\_fpga\_mgr\_get(struct device\_node \* node)

给定一个设备节点，获取FPGA管理器结构的引用

**参数**

struct device\_node \* node

设备节点

**返回**

FPGA管理器结构或包含错误代码的 IS\_ERR() 条件

#### struct fpga\_manager \* fpga\_mgr\_get(struct device \* dev)

给定一个设备，获得 fpga 管理器的引用

**参数**

struct device \* dev

fpga 管理器注册的父设备

**返回**

fpga 管理器结构或包含错误代码的 IS\_ERR() 条件

#### void fpga\_mgr\_put(struct fpga\_manager \* mgr)

释放对 fpga 管理器的引用

**参数**

struct fpga\_manager \* mgr

fpga 管理器结构

#### int fpga\_mgr\_lock(struct fpga\_manager \* mgr)

锁定 FPGA 管理器以进行独占使用

**参数**

struct fpga\_manager \* mgr

fpga管理器

**说明**

给定一个指向 FPGA Manager 的指针（从 fpga\_mgr\_get() 或 of\_fpga\_mgr\_put() 获得），尝试获取互斥体。用户应调用 fpga\_mgr\_lock()，并验证它返回 0 后才尝试编程 FPGA。同样，用户在编程 FPGA 完成后应调用 fpga\_mgr\_unlock。

**返回**

成功为 0，否则为 -EBUSY

#### void fpga\_mgr\_unlock(struct fpga\_manager \* mgr)

完成编程后，请解锁FPGA管理器

**参数**

struct fpga\_manager \* mgr

fpga管理器

#### enum fpga\_mgr\_states

fpga框架状态

**常量**

FPGA\_MGR\_STATE\_UNKNOWN

无法确定状态

FPGA\_MGR\_STATE\_POWER\_OFF

FPGA电源已关闭

FPGA\_MGR\_STATE\_POWER\_UP

FPGA报告电源已上电

FPGA\_MGR\_STATE\_RESET

FPGA重置状态

FPGA\_MGR\_STATE\_FIRMWARE\_REQ

正在进行固件请求

FPGA\_MGR\_STATE\_FIRMWARE\_REQ\_ERR

固件请求失败

FPGA\_MGR\_STATE\_WRITE\_INIT

准备FPGA进行编程

FPGA\_MGR\_STATE\_WRITE\_INIT\_ERR

WRITE\_INIT阶段出错

FPGA\_MGR\_STATE\_WRITE

向FPGA写入图像

FPGA\_MGR\_STATE\_WRITE\_ERR

在写入FPGA时出错

FPGA\_MGR\_STATE\_WRITE\_COMPLETE

进行后期编程步骤

FPGA\_MGR\_STATE\_WRITE\_COMPLETE\_ERR

WRITE\_COMPLETE时出错

FPGA\_MGR\_STATE\_OPERATING

已经编程并正在运行

注：使用fpga\_region\_program\_fpga()代替fpga\_mgr\_load()

#### int fpga\_mgr\_load(struct fpga\_manager \* mgr, struct fpga\_image\_info \* info)

从分散/聚合表、缓冲区或固件加载fpga

**参数**

struct fpga\_manager \* mgr

fpga管理器

struct fpga\_image\_info \* info

fpga镜像信息

**说明**

从在info中指示的映像中加载FPGA。如果成功，FPGA处于操作模式。

**返回**

成功返回 0，否则返回负错误代码

## FPGA桥

### 实现新FPGA桥接的API

#### struct fpga\_bridge

FPGA bridge structure

**定义**

struct fpga\_bridge {

const char \*name;

struct device dev;

struct mutex mutex;

const struct fpga\_bridge\_ops \*br\_ops;

struct fpga\_image\_info \*info;

struct list\_head node;

void \*priv;

};

**成员**

name

低级FPGA桥接的名称

dev

FPGA桥接设备

mutex

强制桥接的独占参考

br\_ops

指向FPGA桥接操作的结构体的指针

info

FPGA镜像特定信息

node

FPGA桥接列表节点

priv

低级驱动程序私有数据

#### struct fpga\_bridge\_ops

用于底层FPGA桥接驱动程序的操作

**定义**

struct fpga\_bridge\_ops {

int (\*enable\_show)(struct fpga\_bridge \*bridge);

int (\*enable\_set)(struct fpga\_bridge \*bridge, bool enable);

void (\*fpga\_bridge\_remove)(struct fpga\_bridge \*bridge);

const struct attribute\_group \*\*groups;

};

**成员**

enable\_show

返回FPGA桥的状态

enable\_set

将FPGA桥设置为启用或禁用

fpga\_bridge\_remove

在驱动程序删除期间将FPGA设置为特定状态

groups

可选属性组

#### struct fpga\_bridge \* fpga\_bridge\_create(struct device \* dev, const char \* name, const struct fpga\_bridge\_ops \* br\_ops, void \* priv)

创建并初始化fpga\_bridge

**参数**

struct device \* dev

从pdev获取的FPGA桥接设备

const char \* name

FPGA桥名称

const struct fpga\_bridge\_ops \* br\_ops

指向FPGA桥接操作的结构体的指针

void \* priv

FPGA桥接私有数据  
**返回**

struct fpga\_bridge 或 NULL

#### void fpga\_bridge\_free(struct fpga\_bridge \* bridge)

释放fpga桥接及其ID

**参数**

struct fpga\_bridge \* bridge

fpga\_bridge\_create创建的FPGA桥接结构体

#### int fpga\_bridge\_register(struct fpga\_bridge \* bridge)

注册FPGA桥

**参数**

struct fpga\_bridge \* bridge

由fpga\_bridge\_create创建的FPGA桥结构体

**返回**

成功返回 0，否则返回错误代码

#### void fpga\_bridge\_unregister(struct [fpga\_bridge](https://www.kernel.org/doc/html/latest/driver-api/fpga/fpga-bridge.html" \l "c.fpga_bridge" \o "fpga_bridge) \*bridge)

注销并释放一个fpga桥

**参数**

struct fpga\_bridge \*bridge

由fpga\_bridge\_create创建的FPGA桥结构体

### 用于控制 FPGA 桥接器的 API

您可能不需要直接使用这些。 FPGA区域应处理此事。

#### struct fpga\_bridge \* of\_fpga\_bridge\_get(struct device\_node \* np, struct fpga\_image\_info \* info)

获取对fpga桥的独占引用

**参数**

struct device\_node \* np

FPGA桥节点指针

struct fpga\_image\_info \* info

FPGA镜像特定信息

**说明**

如果成功，则返回fpga\_bridge结构。如果有人已经引用了桥接，则返回-EBUSY。如果np不是FPGA Bridge，则返回-ENODEV。

#### struct fpga\_bridge \* fpga\_bridge\_get(struct device \* dev, struct fpga\_image\_info \* info)

获取对fpga桥的独占引用

**参数**

struct device \* dev

用于注册fpga桥接的父设备

struct fpga\_image\_info \* info

fpga管理器信息

**说明**

给定设备，获取对fpga桥接的独占引用。

**返回**

fpga桥结构或包含错误代码的IS\_ERR()条件。

#### void fpga\_bridge\_put(struct fpga\_bridge \* bridge)

释放对桥接的引用

**参数**

struct fpga\_bridge \* bridge

FPGA桥接

#### int fpga\_bridge\_get\_to\_list(struct device \* dev, struct fpga\_image\_info \* info, struct list\_head \* bridge\_list)

给定设备，获取桥接，将其添加到列表中

**参数**

struct device \* dev

FPGA桥接设备

struct fpga\_image\_info \* info

FPGA镜像特定信息

struct list\_head \* bridge\_list

FPGA桥接列表

**说明**

获取对桥接的独占引用并将其添加到列表中。

返回0表示成功，从fpga\_bridge\_get()返回的错误代码表示出错。

#### int of\_fpga\_bridge\_get\_to\_list(struct device\_node \* np, struct fpga\_image\_info \* info, struct list\_head \* bridge\_list)

获取桥接并将其添加到列表中

**参数**

struct device\_node \* np

FPGA桥节点指针

struct fpga\_image\_info \* info

FPGA镜像特定信息

struct list\_head \* bridge\_list

FPGA桥接列表

**说明**

获取对桥接的独占引用并将其添加到列表中。

返回0表示成功，从of\_fpga\_bridge\_get()返回的错误代码表示出错。

#### int fpga\_bridge\_enable(struct fpga\_bridge \* bridge)

启用桥上的事务

**参数**

struct fpga\_bridge \* bridge

FPGA桥

**返回**

成功返回 0，否则返回错误代码

#### int fpga\_bridge\_disable(struct fpga\_bridge \* bridge)

禁用桥上的事务

**参数**

struct fpga\_bridge \* bridge

FPGA桥

**返回**

成功返回 0，否则返回错误代码

## FPGA区域

### 概述

本文档旨在简要概述FPGA区域API的使用。在设备树绑定文档中可以找到更具概念性的区域视图[1].

出于本API文档的目的，我们假设一个区域将FPGA管理器和桥接器（或多个桥接器）与FPGA的可重新编程区域或整个FPGA相关联。 API提供了一种注册区域并编程区域的方法。

目前内核中仅在fpga-region.c上面的唯一层[1]对设备树支持（of-fpga-region.c）进行描述。T支持层使用区域对FPGA进行编程，然后使用DT进行枚举处理。通用区域代码旨在由其他具有其他方法在编程后完成枚举的方案使用。

可以设置fpga-region以了解以下内容：

使用哪个 FPGA 管理器进行编程

在编程之前禁用哪些桥接，然后在编程之后启用。

传递到结构体fpga\_image\_info中的附加信息包括：

指向图像的指针作为分散收集缓冲区、连续缓冲区或固件文件的名称

指示细节的标志，包括图像是否用于部分重构等具体信息。

### 如何使用区域对 FPGA 进行编程

首先，分配信息结构:

info = fpga\_image\_info\_alloc(dev);

if (!info)

return -ENOMEM;

根据需要设置标志，即：

info->flags |= FPGA\_MGR\_PARTIAL\_RECONFIG;

指向FPGA映像，例如:

info->sgt = &sgt;

向区域添加信息并进行编程:

region->info = info;

ret = fpga\_region\_program\_fpga(region);

fpga\_region\_program\_fpga（）操作传递给 fpga\_image\_info（region->info）的信息。此函数将尝试:

锁定区域的互斥锁；

锁定区域的FPGA管理器；

如果已指定一种方法，则构建FPGA桥接器列表；

禁用桥接器；

程序FPGA；

重新启用桥接器；

释放锁。

然后，您将需要枚举FPGA中出现的任何硬件。

### 如何添加新的 FPGA 区域

可以在[2]的探测功能中查看用法。

1. ../devicetree/bindings/fpga/fpga-region.txt
2. ../../drivers/fpga/of-fpga-region.c

### 编程FPGA的API

#### int fpga\_region\_program\_fpga(struct fpga\_region \* region)

编程FPGA

**参数**

struct fpga\_region \* region

FPGA区域

**说明**

使用FPGA映像信息（region - >info）编程FPGA。如果该区域具有get\_bridges函数，则在编程成功后会保持桥接器的独占引用。这旨在防止重新编程该区域，直到调用程序认为安全为止。调用程序需要在尝试重新编程该区域之前调用fpga\_bridges\_put（）。返回0表示成功，否则为负错误代码。

### 添加新的FPGA区域的API

#### struct fpga\_region

FPGA区域结构

**定义**

struct fpga\_region {

struct device dev;

struct mutex mutex;

struct list\_head bridge\_list;

struct fpga\_manager \*mgr;

struct fpga\_image\_info \*info;

struct fpga\_compat\_id \*compat\_id;

void \*priv;

int (\*get\_bridges)(struct fpga\_region \*region);

};

**成员**

dev

FPGA区域设备

mutex

强制引用区域

bridge\_list

在区域中指定的FPGA桥接器列表

mgr

FPGA管理器

info

FPGA映像信息

compat\_id

于兼容性检查的FPGA区域标识符

priv

私有数据

get\_bridges

获取桥接器列表的可选函数

#### struct fpga\_region \* fpga\_region\_create(struct device \* dev, struct fpga\_manager \* mgr, int (\*get\_bridges) (struct fpga\_region \*)

分配并初始化fpga\_region

**参数**

struct device \* dev

设备父项

struct fpga\_manager \* mgr

管理此区域的管理器

int (\*)(struct fpga\_region \*) get\_bridges

获取桥接器列表的可选函数

**返回**

struct fpga\_region 或 NULL

#### void fpga\_region\_free(struct fpga\_region \* region)

释放结构体fpga\_region

**参数**

struct fpga\_region \* region

regionfpga\_region\_create创建的FPGA区域

#### int fpga\_region\_register(struct fpga\_region \* region)

注册FPGA区域

**参数**

struct fpga\_region \* region

regionfpga\_region\_create创建的FPGA区域

**返回**

0 或 -errno

#### void fpga\_region\_unregister(struct [fpga\_region](https://www.kernel.org/doc/html/latest/driver-api/fpga/fpga-region.html" \l "c.fpga_region" \o "fpga_region) \*region)

注销并释放FPGA区域

**参数**

struct fpga\_region \*region

FPGA区域