[usb枚举过程分析之hub\_events](http://blog.csdn.net/fanqipin/article/details/8480972)

分类： [usb driver](http://blog.csdn.net/fanqipin/article/category/1280061)2013-01-08 14:23 623人阅读 [评论](http://blog.csdn.net/fanqipin/article/details/8480972#comments)(1) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/fanqipin/article/details/8480972#report)

    当守护程序第一次运行或usb port上状态发生变化，守护进程被唤醒都会运行hub\_events函数，这个函数在usb系统中处理核心位置，usb的枚举过程就是由它完成,usb枚举过程流程图如图1所示；

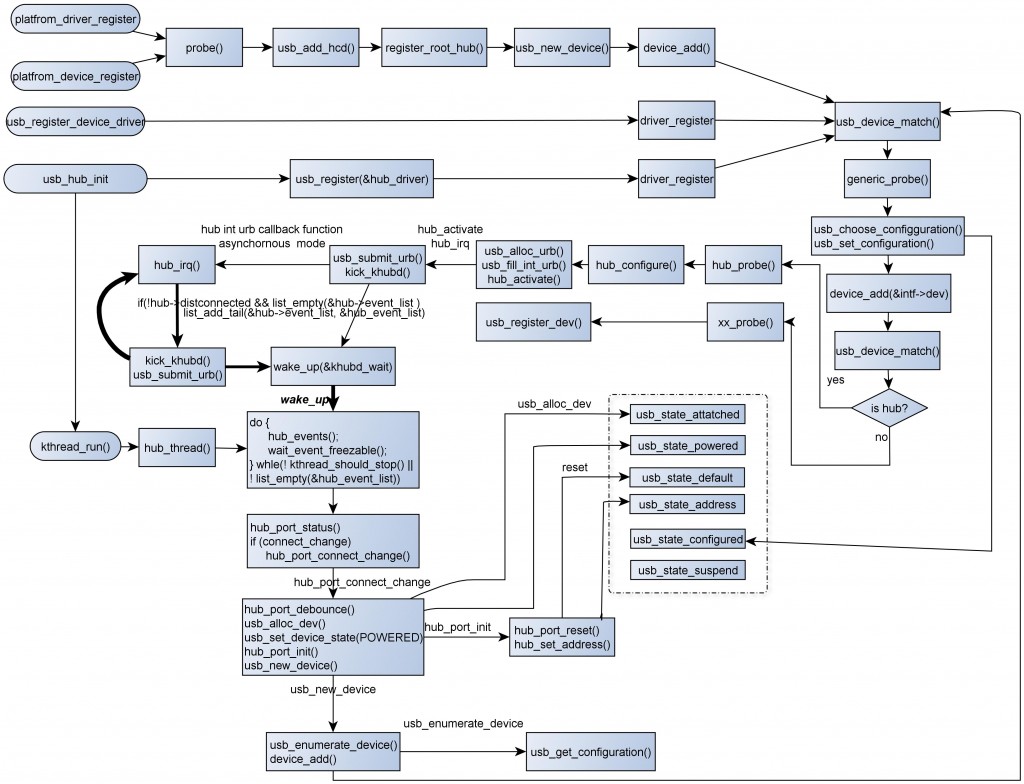


图1 usb枚举流程图

         由于hub\_events函数比较长这里分几个部分进行讲解：

**[html]** [view plaincopy](http://blog.csdn.net/fanqipin/article/details/8480972)

1. static void hub\_events(void)
2. {
3. struct list\_head \*tmp;
4. struct usb\_device \*hdev;
5. struct usb\_interface \*intf;
6. struct usb\_hub \*hub;
7. struct device \*hub\_dev;
8. u16 hubstatus;
9. u16 hubchange;
10. u16 portstatus;
11. u16 portchange;
12. int i, ret;
13. int connect\_change;
15. /\*
16. \*  We restart the list every time to avoid a deadlock with
17. \* deleting hubs downstream from this one. This should be
18. \* safe since we delete the hub from the event list.
19. \* Not the most efficient, but avoids deadlocks.
20. \*/
21. while (1) {
23. /\* Grab the first entry at the beginning of the list \*/
24. spin\_lock\_irq(&hub\_event\_lock);
25. if (list\_empty(&hub\_event\_list)) {
26. spin\_unlock\_irq(&hub\_event\_lock);
27. break;
28. }
30. tmp = hub\_event\_list.next;
31. list\_del\_init(tmp);
33. hub = list\_entry(tmp, struct usb\_hub, event\_list);
34. kref\_get(&hub-**>**kref);
35. spin\_unlock\_irq(&hub\_event\_lock);
37. hdev = hub-**>**hdev;
38. hub\_dev = hub-**>**intfdev;
39. intf = to\_usb\_interface(hub\_dev);
40. dev\_dbg(hub\_dev, "state %d ports %d chg %04x evt %04x\n",
41. hdev-**>**state, hub-**>**descriptor
42. ? hub-**>**descriptor-**>**bNbrPorts
43. : 0,
44. /\* NOTE: expects max 15 ports... \*/
45. (u16) hub-**>**change\_bits[0],
46. (u16) hub-**>**event\_bits[0]);
48. /\* Lock the device, then check to see if we were
49. \* disconnected while waiting for the lock to succeed. \*/
50. usb\_lock\_device(hdev);
51. if (unlikely(hub-**>**disconnected))
52. goto loop\_disconnected;
54. /\* If the hub has died, clean up after it \*/
55. if (hdev-**>**state == USB\_STATE\_NOTATTACHED) {
56. hub-**>**error = -ENODEV;
57. hub\_quiesce(hub, HUB\_DISCONNECT);
58. goto loop;
59. }
61. /\* Autoresume \*/
62. ret = usb\_autopm\_get\_interface(intf);
63. if (ret) {
64. dev\_dbg(hub\_dev, "Can't autoresume: %d\n", ret);
65. goto loop;
66. }
68. /\* If this is an inactive hub, do nothing \*/
69. if (hub-**>**quiescing)
70. goto loop\_autopm;
72. if (hub-**>**error) {
73. dev\_dbg (hub\_dev, "resetting for error %d\n",
74. hub-**>**error);
76. ret = usb\_reset\_device(hdev);
77. if (ret) {
78. dev\_dbg (hub\_dev,
79. "error resetting hub: %d\n", ret);
80. goto loop\_autopm;
81. }
83. hub-**>**nerrors = 0;
84. hub-**>**error = 0;
85. }

hub\_events本身也是一个死循环，只要条件满足它便会一直执行。

25-28行，判断hub\_event\_list是否为空，如果为空，则跳出循环，hub\_events运行结束，会进入休眠状态；

31-35行，从hub\_event\_list列表中取出某一项，并把它从hub\_event\_list中删除，通过list\_entry来获取event\_list所对应的hub，并增加hub使用计数；

51-85行，做了一些逻辑判断，判断hub是否连接，hub上是否有错误，如果有错误就重启hub，如果没有错误，接下来就对hub 上的每个port进行扫描，判断各个port是否发生状态变化；

**[plain]** [view plaincopy](http://blog.csdn.net/fanqipin/article/details/8480972)

1. for (i = 1; i <= hub->descriptor->bNbrPorts; i++) {
2. if (test\_bit(i, hub->busy\_bits))
3. continue;
4. connect\_change = test\_bit(i, hub->change\_bits);
5. if (!test\_and\_clear\_bit(i, hub->event\_bits) &&
6. !connect\_change)
7. continue;
9. ret = hub\_port\_status(hub, i,
10. &portstatus, &portchange);
11. if (ret < 0)
12. continue;
14. if (portchange & USB\_PORT\_STAT\_C\_CONNECTION) {
15. clear\_port\_feature(hdev, i,
16. USB\_PORT\_FEAT\_C\_CONNECTION);
17. connect\_change = 1;
18. }
20. if (portchange & USB\_PORT\_STAT\_C\_ENABLE) {
21. if (!connect\_change)
22. dev\_dbg (hub\_dev,
23. "port %d enable change, "
24. "status %08x\n",
25. i, portstatus);
26. clear\_port\_feature(hdev, i,
27. USB\_PORT\_FEAT\_C\_ENABLE);
29. /\*
30. \* EM interference sometimes causes badly
31. \* shielded USB devices to be shutdown by
32. \* the hub, this hack enables them again.
33. \* Works at least with mouse driver.
34. \*/
35. if (!(portstatus & USB\_PORT\_STAT\_ENABLE)
36. && !connect\_change
37. && hdev->children[i-1]) {
38. dev\_err (hub\_dev,
39. "port %i "
40. "disabled by hub (EMI?), "
41. "re-enabling...\n",
42. i);
43. connect\_change = 1;
44. }
45. }
47. if (portchange & USB\_PORT\_STAT\_C\_SUSPEND) {
48. struct usb\_device \*udev;
50. clear\_port\_feature(hdev, i,
51. USB\_PORT\_FEAT\_C\_SUSPEND);
52. udev = hdev->children[i-1];
53. if (udev) {
54. /\* TRSMRCY = 10 msec \*/
55. msleep(10);
57. usb\_lock\_device(udev);
58. ret = usb\_remote\_wakeup(hdev->
59. children[i-1]);
60. usb\_unlock\_device(udev);
61. if (ret < 0)
62. connect\_change = 1;
63. } else {
64. ret = -ENODEV;
65. hub\_port\_disable(hub, i, 1);
66. }
67. dev\_dbg (hub\_dev,
68. "resume on port %d, status %d\n",
69. i, ret);
70. }
72. if (portchange & USB\_PORT\_STAT\_C\_OVERCURRENT) {
73. u16 status = 0;
74. u16 unused;
76. dev\_dbg(hub\_dev, "over-current change on port "
77. "%d\n", i);
78. clear\_port\_feature(hdev, i,
79. USB\_PORT\_FEAT\_C\_OVER\_CURRENT);
80. msleep(100);    /\* Cool down \*/
81. hub\_power\_on(hub, true);
82. hub\_port\_status(hub, i, &status, &unused);
83. if (status & USB\_PORT\_STAT\_OVERCURRENT)
84. dev\_err(hub\_dev, "over-current "
85. "condition on port %d\n", i);
86. }
88. if (portchange & USB\_PORT\_STAT\_C\_RESET) {
89. dev\_dbg (hub\_dev,
90. "reset change on port %d\n",
91. i);
92. clear\_port\_feature(hdev, i,
93. USB\_PORT\_FEAT\_C\_RESET);
94. }
95. if ((portchange & USB\_PORT\_STAT\_C\_BH\_RESET) &&
96. hub\_is\_superspeed(hub->hdev)) {
97. dev\_dbg(hub\_dev,
98. "warm reset change on port %d\n",
99. i);
100. clear\_port\_feature(hdev, i,
101. USB\_PORT\_FEAT\_C\_BH\_PORT\_RESET);
102. }
103. if (portchange & USB\_PORT\_STAT\_C\_LINK\_STATE) {
104. clear\_port\_feature(hub->hdev, i,
105. USB\_PORT\_FEAT\_C\_PORT\_LINK\_STATE);
106. }
107. if (portchange & USB\_PORT\_STAT\_C\_CONFIG\_ERROR) {
108. dev\_warn(hub\_dev,
109. "config error on port %d\n",
110. i);
111. clear\_port\_feature(hub->hdev, i,
112. USB\_PORT\_FEAT\_C\_PORT\_CONFIG\_ERROR);
113. }
115. /\* Warm reset a USB3 protocol port if it's in
116. \* SS.Inactive state.
117. \*/
118. if (hub\_is\_superspeed(hub->hdev) &&
119. (portstatus & USB\_PORT\_STAT\_LINK\_STATE)
120. == USB\_SS\_PORT\_LS\_SS\_INACTIVE) {
121. dev\_dbg(hub\_dev, "warm reset port %d\n", i);
122. hub\_port\_warm\_reset(hub, i);
123. }
125. if (connect\_change)
126. hub\_port\_connect\_change(hub, i,
127. portstatus, portchange);
128. } /\* end for i \*/

本段程序主要是遍历hub中的所有port，对各个port的状态进行查看，判断port是否发生了状态变化，如果产生了变化则调用hub\_port\_connect\_change进行处理；

在hub结构中存在多个标志位，用来表示port一些状态：

1. 如果usb的第*i个*port处于resume或reset状态，则hub中busy\_bits中第i个位置1.如果busy\_bits中第i个位置1,则退过当前port；

2. event\_bits中第0位用来表示hub本身是否产生local power status change和是否产生过流，其它的各位用来表示hub下各个port状态是否发生变化，这些状态包括： ConnectStatusChange 连接状态发生变化，PortEnableStatusChange端口使能状态发生变化，PortSuspendStatusChange端口从挂起到恢复完成，PortOverCurrentIndicatorChange端口过流状态发生变化，PortResetStatusChange端口reset10ms置位；

3. remove\_bits用来表示hub下各个port是否有设备连接，如果置位表示没有设备连接或设备已经断开；

4. change\_bits用来表示hub下各个端口逻辑状态是否发生变化，它是在hub初始化的时候赋值的，这些状态主要有：1.原先port上没有设备，现在检测到有设备连接；2.原先port上有设备，由于控制器不正常关闭或禁止usb port等原图使得该设备处于NOATTACHED态，则需要断开设备；

2-12行，对这些标志进行判断，如果没有产生变化则跳过当前port，否则读取当前port的status，判断出产生状态变化的原因；

15-19行，表示port的连接状态发生变化，清除相应的端口状态，设置connect\_change变量为1；

21-46行，表示port的使能状态发生变化，清除相应的状态标志，如果是由于EMI电磁干扰引起的状态变化，则设置connect\_change为1；

47-70行，表示port的suspend状态发生变化，它由suspend转向resume态；

71-86行，表示port的过流状态发生变化；

88-93行，表示port reset完成；

95-123行，它们都是usb 3.0协议里才出现的，用来表示port link的状态；

如果发现port的连接状态发生变化或由于EMI导致连接使能发生变化，即connect\_change=1，则调用hub\_port\_connect\_change.

**[plain]** [view plaincopy](http://blog.csdn.net/fanqipin/article/details/8480972)

1. static void hub\_port\_connect\_change(struct usb\_hub \*hub, int port1,
2. u16 portstatus, u16 portchange)
3. {
4. struct usb\_device \*hdev = hub->hdev;
5. struct device \*hub\_dev = hub->intfdev;
6. struct usb\_hcd \*hcd = bus\_to\_hcd(hdev->bus);
7. unsigned wHubCharacteristics =
8. le16\_to\_cpu(hub->descriptor->wHubCharacteristics);
9. struct usb\_device \*udev;
10. int status, i;
12. dev\_dbg (hub\_dev,
13. "port %d, status %04x, change %04x, %s\n",
14. port1, portstatus, portchange, portspeed(hub, portstatus));
16. if (hub->has\_indicators) {
17. set\_port\_led(hub, port1, HUB\_LED\_AUTO);
18. hub->indicator[port1-1] = INDICATOR\_AUTO;
19. }
21. #ifdef  CONFIG\_USB\_OTG
22. /\* during HNP, don't repeat the debounce \*/
23. if (hdev->bus->is\_b\_host)
24. portchange &= ~(USB\_PORT\_STAT\_C\_CONNECTION |
25. USB\_PORT\_STAT\_C\_ENABLE);
26. #endif
28. /\* Try to resuscitate an existing device \*/
29. udev = hdev->children[port1-1];
30. if ((portstatus & USB\_PORT\_STAT\_CONNECTION) && udev &&
31. udev->state != USB\_STATE\_NOTATTACHED) {
32. usb\_lock\_device(udev);
33. if (portstatus & USB\_PORT\_STAT\_ENABLE) {
34. status = 0;     /\* Nothing to do \*/
36. #ifdef CONFIG\_USB\_SUSPEND
37. } else if (udev->state == USB\_STATE\_SUSPENDED &&
38. udev->persist\_enabled) {
39. /\* For a suspended device, treat this as a
40. \* remote wakeup event.
41. \*/
42. status = usb\_remote\_wakeup(udev);
43. #endif
45. } else {
46. status = -ENODEV;   /\* Don't resuscitate \*/
47. }
48. usb\_unlock\_device(udev);
50. if (status == 0) {
51. clear\_bit(port1, hub->change\_bits);
52. return;
53. }
54. }
56. /\* Disconnect any existing devices under this port \*/
57. if (udev)
58. usb\_disconnect(&hdev->children[port1-1]);
59. clear\_bit(port1, hub->change\_bits);
61. /\* We can forget about a "removed" device when there's a physical
62. \* disconnect or the connect status changes.
63. \*/
64. if (!(portstatus & USB\_PORT\_STAT\_CONNECTION) ||
65. (portchange & USB\_PORT\_STAT\_C\_CONNECTION))
66. clear\_bit(port1, hub->removed\_bits);
68. if (portchange & (USB\_PORT\_STAT\_C\_CONNECTION |
69. USB\_PORT\_STAT\_C\_ENABLE)) {
70. status = hub\_port\_debounce(hub, port1);
71. if (status < 0) {
72. if (printk\_ratelimit())
73. dev\_err(hub\_dev, "connect-debounce failed, "
74. "port %d disabled\n", port1);
75. portstatus &= ~USB\_PORT\_STAT\_CONNECTION;
76. } else {
77. portstatus = status;
78. }
79. }
81. /\* Return now if debouncing failed or nothing is connected or
82. \* the device was "removed".
83. \*/
84. if (!(portstatus & USB\_PORT\_STAT\_CONNECTION) ||
85. test\_bit(port1, hub->removed\_bits)) {
87. /\* maybe switch power back on (e.g. root hub was reset) \*/
88. if ((wHubCharacteristics & HUB\_CHAR\_LPSM) < 2
89. && !port\_is\_power\_on(hub, portstatus))
90. set\_port\_feature(hdev, port1, USB\_PORT\_FEAT\_POWER);
92. if (portstatus & USB\_PORT\_STAT\_ENABLE)
93. goto done;
94. return;
95. }
97. for (i = 0; i < SET\_CONFIG\_TRIES; i++) {
99. /\* reallocate for each attempt, since references
100. \* to the previous one can escape in various ways
101. \*/
102. udev = usb\_alloc\_dev(hdev, hdev->bus, port1);
103. if (!udev) {
104. dev\_err (hub\_dev,
105. "couldn't allocate port %d usb\_device\n",
106. port1);
107. goto done;
108. }
110. usb\_set\_device\_state(udev, USB\_STATE\_POWERED);
111. udev->bus\_mA = hub->mA\_per\_port;
112. udev->level = hdev->level + 1;
113. udev->wusb = hub\_is\_wusb(hub);
115. /\* Only USB 3.0 devices are connected to SuperSpeed hubs. \*/
116. if (hub\_is\_superspeed(hub->hdev))
117. udev->speed = USB\_SPEED\_SUPER;
118. else
119. udev->speed = USB\_SPEED\_UNKNOWN;
121. choose\_devnum(udev);
122. if (udev->devnum <= 0) {
123. status = -ENOTCONN; /\* Don't retry \*/
124. goto loop;
125. }
127. /\* reset (non-USB 3.0 devices) and get descriptor \*/
128. status = hub\_port\_init(hub, udev, port1, i);
129. if (status < 0)
130. goto loop;
132. usb\_detect\_quirks(udev);
133. if (udev->quirks & USB\_QUIRK\_DELAY\_INIT)
134. msleep(1000);
136. /\* consecutive bus-powered hubs aren't reliable; they can
137. \* violate the voltage drop budget.  if the new child has
138. \* a "powered" LED, users should notice we didn't enable it
139. \* (without reading syslog), even without per-port LEDs
140. \* on the parent.
141. \*/
142. if (udev->descriptor.bDeviceClass == USB\_CLASS\_HUB
143. && udev->bus\_mA <= 100) {
144. u16 devstat;
146. status = usb\_get\_status(udev, USB\_RECIP\_DEVICE, 0,
147. &devstat);
148. if (status < 2) {
149. dev\_dbg(&udev->dev, "get status %d ?\n", status);
150. goto loop\_disable;
151. }
152. le16\_to\_cpus(&devstat);
153. if ((devstat & (1 << USB\_DEVICE\_SELF\_POWERED)) == 0) {
154. dev\_err(&udev->dev,
155. "can't connect bus-powered hub "
156. "to this port\n");
157. if (hub->has\_indicators) {
158. hub->indicator[port1-1] =
159. INDICATOR\_AMBER\_BLINK;
160. schedule\_delayed\_work (&hub->leds, 0);
161. }
162. status = -ENOTCONN; /\* Don't retry \*/
163. goto loop\_disable;
164. }
165. }
167. /\* check for devices running slower than they could \*/
168. if (le16\_to\_cpu(udev->descriptor.bcdUSB) >= 0x0200
169. && udev->speed == USB\_SPEED\_FULL
170. && highspeed\_hubs != 0)
171. check\_highspeed (hub, udev, port1);
173. /\* Store the parent's children[] pointer.  At this point
174. \* udev becomes globally accessible, although presumably
175. \* no one will look at it until hdev is unlocked.
176. \*/
177. status = 0;
179. /\* We mustn't add new devices if the parent hub has
180. \* been disconnected; we would race with the
181. \* recursively\_mark\_NOTATTACHED() routine.
182. \*/
183. spin\_lock\_irq(&device\_state\_lock);
184. if (hdev->state == USB\_STATE\_NOTATTACHED)
185. status = -ENOTCONN;
186. else
187. hdev->children[port1-1] = udev;
188. spin\_unlock\_irq(&device\_state\_lock);
190. /\* Run it through the hoops (find a driver, etc) \*/
191. if (!status) {
192. status = usb\_new\_device(udev);
193. if (status) {
194. spin\_lock\_irq(&device\_state\_lock);
195. hdev->children[port1-1] = NULL;
196. spin\_unlock\_irq(&device\_state\_lock);
197. }
198. }
200. if (status)
201. goto loop\_disable;
203. status = hub\_power\_remaining(hub);
204. if (status)
205. dev\_dbg(hub\_dev, "%dmA power budget left\n", status);
207. return;
209. loop\_disable:
210. hub\_port\_disable(hub, port1, 1);
211. loop:
212. usb\_ep0\_reinit(udev);
213. release\_devnum(udev);
214. hub\_free\_dev(udev);
215. usb\_put\_dev(udev);
216. if ((status == -ENOTCONN) || (status == -ENOTSUPP))
217. break;
218. }
219. if (hub->hdev->parent ||
220. !hcd->driver->port\_handed\_over ||
221. !(hcd->driver->port\_handed\_over)(hcd, port1))
222. dev\_err(hub\_dev, "unable to enumerate USB device on port %d\n",
223. port1);
225. done:
226. hub\_port\_disable(hub, port1, 1);
227. if (hcd->driver->relinquish\_port && !hub->hdev->parent)
228. hcd->driver->relinquish\_port(hcd, port1);
229. }

hub\_port\_connect\_change分为两部分：第一部分主要是：确认是否有新设备插入；第二部分主要是：确认port口上有新设备插入后，分配设备资源，并进行枚举操作；  
16-19行， 如果当前的hub支持indicator指示灯，则设备指示灯为HUB\_LED\_AUTO;

29-54行，试着去唤醒hub上一些处于suspend的设备，对于两类型设备它不会去执行枚举操作，一是usb设备本身已经使能，二是usb 设备处于suspend状态，并可以唤醒，对于这两类设备，它不需要去申请usb设备结构，所以在清除相应port的change\_bits后，返回；

57-58行，hub port连接状态变化可以是从有到无，也可以是从无到有，对于后者这里不可能有udev，它以属于前者，把一些已经处于NOATTACHED的设备从hub中断开；

59行，清除当前port的change\_bits标志；

68-95行，对于一些已经连接的设备，进行一些类似防抖动处理，处理机制：每隔25ms去读取port的status和change状态，查看port连接状态是否稳定，如果port处于稳定状态大于100ms，则认为当前port上的设备已经稳定，这种处理机制最长时间为1.5s;如果port处于稳定状态的时间小于100ms，则认为连接是不稳定的，则跳出当前port，去执行下一个port;

对于一个新插入的设备，并已经确定它的存在，则接下来会为这个usb设备进行枚举；

102-108行，通过usb\_alloc\_dev为新的USB设备申请资源，并进行一些初始化，将usb设置为USB\_STATE\_ATTACHED，表示设备已经连接；

110行，设置usb设备为USB\_STATE\_POWERED态；

111-113行，设置usb设备可以从port上获取的电流量，及usb设备的拓扑层级；

116-119行，如果hub支持超速，则设置usb设备的速度为超速，否则设置usb设备速度 为unknow;

121-125行，一条usb总线下总共可以有128个设备，usb总线通过位图的形式来管理usb设备的地址，choose\_devnum是从usb 总线中找到一个usb地址，usb地址在1和128之间；

128行，通过hub\_port\_init对usb设备进行reset，并设置usb设备的地址，获取usb设备的设备描述符，这个函数相对比较重要，所以对它进行了进一步分析：

**[html]** [view plaincopy](http://blog.csdn.net/fanqipin/article/details/8480972)

1. static int
2. hub\_port\_init (struct usb\_hub \*hub, struct usb\_device \*udev, int port1,
3. int retry\_counter)
4. {
5. static DEFINE\_MUTEX(usb\_address0\_mutex);
7. struct usb\_device   \*hdev = hub-**>**hdev;
8. struct usb\_hcd      \*hcd = bus\_to\_hcd(hdev-**>**bus);
9. int         i, j, retval;
10. unsigned        delay = HUB\_SHORT\_RESET\_TIME;
11. enum usb\_device\_speed   oldspeed = udev-**>**speed;
12. char            \*speed, \*type;
13. int         devnum = udev-**>**devnum;
15. /\* root hub ports have a slightly longer reset period
16. \* (from USB 2.0 spec, section 7.1.7.5)
17. \*/
18. if (!hdev-**>**parent) {
19. delay = HUB\_ROOT\_RESET\_TIME;
20. if (port1 == hdev-**>**bus-**>**otg\_port)
21. hdev-**>**bus-**>**b\_hnp\_enable = 0;
22. }
24. /\* Some low speed devices have problems with the quick delay, so \*/
25. /\*  be a bit pessimistic with those devices. RHbug #23670 \*/
26. if (oldspeed == USB\_SPEED\_LOW)
27. delay = HUB\_LONG\_RESET\_TIME;
29. mutex\_lock(&usb\_address0\_mutex);
31. /\* Reset the device; full speed may morph to high speed \*/
32. /\* FIXME a USB 2.0 device may morph into SuperSpeed on reset. \*/
33. retval = hub\_port\_reset(hub, port1, udev, delay);
34. if (retval **<** **0**)      /\* error or disconnect \*/
35. goto fail;
36. /\* success, speed is known \*/
38. retval = -ENODEV;
40. if (oldspeed != USB\_SPEED\_UNKNOWN && oldspeed != udev-**>**speed) {
41. dev\_dbg(&udev-**>**dev, "device reset changed speed!\n");
42. goto fail;
43. }
44. oldspeed = udev-**>**speed;
46. /\* USB 2.0 section 5.5.3 talks about ep0 maxpacket ...
47. \* it's fixed size except for full speed devices.
48. \* For Wireless USB devices, ep0 max packet is always 512 (tho
49. \* reported as 0xff in the device descriptor). WUSB1.0[4.8.1].
50. \*/
51. switch (udev-**>**speed) {
52. case USB\_SPEED\_SUPER:
53. case USB\_SPEED\_WIRELESS:    /\* fixed at 512 \*/
54. udev-**>**ep0.desc.wMaxPacketSize = cpu\_to\_le16(512);
55. break;
56. case USB\_SPEED\_HIGH:        /\* fixed at 64 \*/
57. udev-**>**ep0.desc.wMaxPacketSize = cpu\_to\_le16(64);
58. break;
59. case USB\_SPEED\_FULL:        /\* 8, 16, 32, or 64 \*/
60. /\* to determine the ep0 maxpacket size, try to read
61. \* the device descriptor to get bMaxPacketSize0 and
62. \* then correct our initial guess.
63. \*/
64. udev-**>**ep0.desc.wMaxPacketSize = cpu\_to\_le16(64);
65. break;
66. case USB\_SPEED\_LOW:     /\* fixed at 8 \*/
67. udev-**>**ep0.desc.wMaxPacketSize = cpu\_to\_le16(8);
68. break;
69. default:
70. goto fail;
71. }
73. type = "";
74. switch (udev-**>**speed) {
75. case USB\_SPEED\_LOW: speed = "low";  break;
76. case USB\_SPEED\_FULL:    speed = "full"; break;
77. case USB\_SPEED\_HIGH:    speed = "high"; break;
78. case USB\_SPEED\_SUPER:
79. speed = "super";
80. break;
81. case USB\_SPEED\_WIRELESS:
82. speed = "variable";
83. type = "Wireless ";
84. break;
85. default:        speed = "?";    break;
86. }
87. if (udev-**>**speed != USB\_SPEED\_SUPER)
88. dev\_info(&udev-**>**dev,
89. "%s %s speed %sUSB device number %d using %s\n",
90. (udev-**>**config) ? "reset" : "new", speed, type,
91. devnum, udev-**>**bus-**>**controller-**>**driver-**>**name);
93. /\* Set up TT records, if needed  \*/
94. if (hdev-**>**tt) {
95. udev-**>**tt = hdev-**>**tt;
96. udev-**>**ttport = hdev-**>**ttport;
97. } else if (udev-**>**speed != USB\_SPEED\_HIGH
98. && hdev-**>**speed == USB\_SPEED\_HIGH) {
99. if (!hub-**>**tt.hub) {
100. dev\_err(&udev-**>**dev, "parent hub has no TT\n");
101. retval = -EINVAL;
102. goto fail;
103. }
104. udev-**>**tt = &hub-**>**tt;
105. udev-**>**ttport = port1;
106. }
108. /\* Why interleave GET\_DESCRIPTOR and SET\_ADDRESS this way?
109. \* Because device hardware and firmware is sometimes buggy in
110. \* this area, and this is how Linux has done it for ages.
111. \* Change it cautiously.
112. \*
113. \* NOTE:  If USE\_NEW\_SCHEME() is true we will start by issuing
114. \* a 64-byte GET\_DESCRIPTOR request.  This is what Windows does,
115. \* so it may help with some non-standards-compliant devices.
116. \* Otherwise we start with SET\_ADDRESS and then try to read the
117. \* first 8 bytes of the device descriptor to get the ep0 maxpacket
118. \* value.
119. \*/
120. for (i = 0; i **<** **GET\_DESCRIPTOR\_TRIES**; (++i, msleep(100))) {
121. if (USE\_NEW\_SCHEME(retry\_counter) && !(hcd-**>**driver-**>**flags & HCD\_USB3)) {
122. struct usb\_device\_descriptor \*buf;
123. int r = 0;
125. #define GET\_DESCRIPTOR\_BUFSIZE  64
126. buf = kmalloc(GET\_DESCRIPTOR\_BUFSIZE, GFP\_NOIO);
127. if (!buf) {
128. retval = -ENOMEM;
129. continue;
130. }
132. /\* Retry on all errors; some devices are flakey.
133. \* 255 is for WUSB devices, we actually need to use
134. \* 512 (WUSB1.0[4.8.1]).
135. \*/
136. for (j = 0; j **<** **3**; ++j) {
137. buf-**>**bMaxPacketSize0 = 0;
138. r = usb\_control\_msg(udev, usb\_rcvaddr0pipe(),
139. USB\_REQ\_GET\_DESCRIPTOR, USB\_DIR\_IN,
140. USB\_DT\_DEVICE **<<** **8**, 0,
141. buf, GET\_DESCRIPTOR\_BUFSIZE,
142. initial\_descriptor\_timeout);
143. switch (buf-**>**bMaxPacketSize0) {
144. case 8: case 16: case 32: case 64: case 255:
145. if (buf-**>**bDescriptorType ==
146. USB\_DT\_DEVICE) {
147. r = 0;
148. break;
149. }
150. /\* FALL THROUGH \*/
151. default:
152. if (r == 0)
153. r = -EPROTO;
154. break;
155. }
156. if (r == 0)
157. break;
158. }
159. udev-**>**descriptor.bMaxPacketSize0 =
160. buf-**>**bMaxPacketSize0;
161. kfree(buf);
163. retval = hub\_port\_reset(hub, port1, udev, delay);
164. if (retval **<** **0**)      /\* error or disconnect \*/
165. goto fail;
166. if (oldspeed != udev-**>**speed) {
167. dev\_dbg(&udev-**>**dev,
168. "device reset changed speed!\n");
169. retval = -ENODEV;
170. goto fail;
171. }
172. if (r) {
173. dev\_err(&udev-**>**dev,
174. "device descriptor read/64, error %d\n",
175. r);
176. retval = -EMSGSIZE;
177. continue;
178. }
179. #undef GET\_DESCRIPTOR\_BUFSIZE
180. }
182. /\*
183. \* If device is WUSB, we already assigned an
184. \* unauthorized address in the Connect Ack sequence;
185. \* authorization will assign the final address.
186. \*/
187. if (udev-**>**wusb == 0) {
188. for (j = 0; j **<** **SET\_ADDRESS\_TRIES**; ++j) {
189. retval = hub\_set\_address(udev, devnum);
190. if (retval **>**= 0)
191. break;
192. msleep(200);
193. }
194. if (retval **<** **0**) {
195. dev\_err(&udev-**>**dev,
196. "device not accepting address %d, error %d\n",
197. devnum, retval);
198. goto fail;
199. }
200. if (udev-**>**speed == USB\_SPEED\_SUPER) {
201. devnum = udev-**>**devnum;
202. dev\_info(&udev-**>**dev,
203. "%s SuperSpeed USB device number %d using %s\n",
204. (udev-**>**config) ? "reset" : "new",
205. devnum, udev-**>**bus-**>**controller-**>**driver-**>**name);
206. }
208. /\* cope with hardware quirkiness:
209. \*  - let SET\_ADDRESS settle, some device hardware wants it
210. \*  - read ep0 maxpacket even for high and low speed,
211. \*/
212. msleep(10);
213. if (USE\_NEW\_SCHEME(retry\_counter) && !(hcd-**>**driver-**>**flags & HCD\_USB3))
214. break;
215. }
217. retval = usb\_get\_device\_descriptor(udev, 8);
218. if (retval **<** **8**) {
219. dev\_err(&udev-**>**dev,
220. "device descriptor read/8, error %d\n",
221. retval);
222. if (retval **>**= 0)
223. retval = -EMSGSIZE;
224. } else {
225. retval = 0;
226. break;
227. }
228. }
229. if (retval)
230. goto fail;
232. if (udev-**>**descriptor.bMaxPacketSize0 == 0xff ||
233. udev-**>**speed == USB\_SPEED\_SUPER)
234. i = 512;
235. else
236. i = udev-**>**descriptor.bMaxPacketSize0;
237. if (le16\_to\_cpu(udev-**>**ep0.desc.wMaxPacketSize) != i) {
238. if (udev-**>**speed == USB\_SPEED\_LOW ||
239. !(i == 8 || i == 16 || i == 32 || i == 64)) {
240. dev\_err(&udev-**>**dev, "Invalid ep0 maxpacket: %d\n", i);
241. retval = -EMSGSIZE;
242. goto fail;
243. }
244. if (udev-**>**speed == USB\_SPEED\_FULL)
245. dev\_dbg(&udev-**>**dev, "ep0 maxpacket = %d\n", i);
246. else
247. dev\_warn(&udev-**>**dev, "Using ep0 maxpacket: %d\n", i);
248. udev-**>**ep0.desc.wMaxPacketSize = cpu\_to\_le16(i);
249. usb\_ep0\_reinit(udev);
250. }
252. retval = usb\_get\_device\_descriptor(udev, USB\_DT\_DEVICE\_SIZE);
253. if (retval **<** (signed)sizeof(udev-**>**descriptor)) {
254. dev\_err(&udev-**>**dev, "device descriptor read/all, error %d\n",
255. retval);
256. if (retval **>**= 0)
257. retval = -ENOMSG;
258. goto fail;
259. }
261. retval = 0;
262. /\* notify HCD that we have a device connected and addressed \*/
263. if (hcd-**>**driver-**>**update\_device)
264. hcd-**>**driver-**>**update\_device(hcd, udev);
265. fail:
266. if (retval) {
267. hub\_port\_disable(hub, port1, 0);
268. update\_devnum(udev, devnum);    /\* for disconnect processing \*/
269. }
270. mutex\_unlock(&usb\_address0\_mutex);
271. return retval;
272. }

hub\_port\_init首先对port进行reset,复位成功后设备usb设备的速度和端口0最大发送数据长度，设置usb设备的地址，并获取usb设备的设备描述符；

18-27行，设置用于读取port状态的时间间隔，root hub需要50ms,普通的hub需要10ms，对于低速的usb设备需要200ms；  
33行，通过hub\_port\_reset来reset设备，reset机制为：通过set\_port\_feature来传输USB\_PORT\_FEAT\_RESET指令，设置成功后循环延时由之前确定的时间间隔后去读取port的status和change状态，要确保usb设备在reset后还能正常存在，如reset还能正常，则通过port的status状态位来确定usb设备的速度，循环延时总时间为500ms,而usb设备reset次数为5次，

51-72行，根据usb设备的速度来初始化endpont0的最大发送数据长度，对于无线usb其maxpacketsize为512字节，对于高速和全速为64字节，而低速为8个字节；

94-106行，如果hub为高速，而usb设备为低速或全速，则在它们之间需要有一个速度转换设备tt，这几行代码就是 用来设置TT的；

120-228行，通过获取usb设备的设备描述符来得到usb设备的endpoint0的最大发送数据长度，设备描述符实际 上有18个字节，而endpoint0的maxpacketsize在设备描述符里的第8个字节位，所以只要获取设备描述符的前8个字节数据就可以了，但有些USB设备它并不支持只获取8个字节这样不完整的usb请求，为解决这种问题usb驱动里采用了两种策略去获取maxpacketsize，一种是windows作法，它直接向usb设备请求64字节数据，对于一些maxpacketsize为32或64的，它可以直接一次性把18个字节的设备描述符传输回来，而对于像low speed的设备它的maxpacketsize只有8个字节，就需要进行多次发送; 而linux作法是先设置usb设备的地址，然后再发送8个字节数据请求，从返回的8个字节里获取endpoint0的maxpacketsize;

232-236行，根据usb设备描述符确定endpoint0的maxpacketsize,对于无线usb设备和超速usb设备，maxpacketsize为512，而其它类型就为从设备侧获取的maxpacketsize;

237-250行，如果初始化时设置的maxpacketsize和设备描述符里的maxpacketsize不一样，则将endpint0 maxpacketsize设置成和设备描述符里的一样；

252-259行,根据maxpacketsize去获取完整的设备描述符；

在取得usb的设备描述符后，返回到hub\_port\_connect\_change函数，接下来根据设备描述符对usb设备合法性进行进一步判断，如果合法则去获取这个设备的配置，接口，端口描述符，并将usb设备注册到系统中，为它找到合适的设备驱动；

142-165行，如果当前的usb设备是一个hub，它由hub供电，而它从上一级hub上分配到的电流少于100mA，这种情况 是不允许的，它会通过点亮AMBER指示灯来显示出现这种错误，指示灯由延时任务队列完成；

168-171行，如果usb设备支持高速运行，而现在却工作在全速，它就会通过点亮绿色指示灯来指示这种错误；

187行，如果通过这些判断后，usb设备还是正常的， 则将它放到用于表示hub每个port的指针数据hub->children里，并通过usb\_new\_device去获取usb设备的配置信息，最后将它注册到系统中；

**[plain]** [view plaincopy](http://blog.csdn.net/fanqipin/article/details/8480972)

1. int usb\_new\_device(struct usb\_device \*udev)
2. {
3. int err;
5. if (udev->parent) {
6. /\* Initialize non-root-hub device wakeup to disabled;
7. \* device (un)configuration controls wakeup capable
8. \* sysfs power/wakeup controls wakeup enabled/disabled
9. \*/
10. device\_init\_wakeup(&udev->dev, 0);
11. }
13. /\* Tell the runtime-PM framework the device is active \*/
14. pm\_runtime\_set\_active(&udev->dev);
15. pm\_runtime\_get\_noresume(&udev->dev);
16. pm\_runtime\_use\_autosuspend(&udev->dev);
17. pm\_runtime\_enable(&udev->dev);
19. /\* By default, forbid autosuspend for all devices.  It will be
20. \* allowed for hubs during binding.
21. \*/
22. usb\_disable\_autosuspend(udev);
24. err = usb\_enumerate\_device(udev);   /\* Read descriptors \*/
25. if (err < 0)
26. goto fail;
27. dev\_dbg(&udev->dev, "udev %d, busnum %d, minor = %d\n",
28. udev->devnum, udev->bus->busnum,
29. (((udev->bus->busnum-1) \* 128) + (udev->devnum-1)));
30. /\* export the usbdev device-node for libusb \*/
31. udev->dev.devt = MKDEV(USB\_DEVICE\_MAJOR,
32. (((udev->bus->busnum-1) \* 128) + (udev->devnum-1)));
34. /\* Tell the world! \*/
35. announce\_device(udev);
37. device\_enable\_async\_suspend(&udev->dev);
38. /\* Register the device.  The device driver is responsible
39. \* for configuring the device and invoking the add-device
40. \* notifier chain (used by usbfs and possibly others).
41. \*/
42. err = device\_add(&udev->dev);
43. if (err) {
44. dev\_err(&udev->dev, "can't device\_add, error %d\n", err);
45. goto fail;
46. }
48. (void) usb\_create\_ep\_devs(&udev->dev, &udev->ep0, udev);
49. usb\_mark\_last\_busy(udev);
50. pm\_runtime\_put\_sync\_autosuspend(&udev->dev);
51. return err;
53. fail:
54. usb\_set\_device\_state(udev, USB\_STATE\_NOTATTACHED);
55. pm\_runtime\_disable(&udev->dev);
56. pm\_runtime\_set\_suspended(&udev->dev);
57. return err;
58. }

usb\_new\_device主是要获取usb设备的配置信息，然后将usb设备添加到系统中，这里usb\_enumerate\_device相对比较得要，就是它完成了配置信息获取及分配；

**[plain]** [view plaincopy](http://blog.csdn.net/fanqipin/article/details/8480972)

1. static int usb\_enumerate\_device(struct usb\_device \*udev)
2. {
3. int err;
5. if (udev->config == NULL) {
6. err = usb\_get\_configuration(udev);
7. if (err < 0) {
8. dev\_err(&udev->dev, "can't read configurations, error %d\n",
9. err);
10. goto fail;
11. }
12. }
13. if (udev->wusb == 1 && udev->authorized == 0) {
14. udev->product = kstrdup("n/a (unauthorized)", GFP\_KERNEL);
15. udev->manufacturer = kstrdup("n/a (unauthorized)", GFP\_KERNEL);
16. udev->serial = kstrdup("n/a (unauthorized)", GFP\_KERNEL);
17. }
18. else {
19. /\* read the standard strings and cache them if present \*/
20. udev->product = usb\_cache\_string(udev, udev->descriptor.iProduct);
21. udev->manufacturer = usb\_cache\_string(udev,
22. udev->descriptor.iManufacturer);
23. udev->serial = usb\_cache\_string(udev, udev->descriptor.iSerialNumber);
24. }
25. err = usb\_enumerate\_device\_otg(udev);
26. fail:
27. return err;
28. }

5-12行，通过usb\_get\_configuration获取和分配usb设备的配置，接口，端口信息；

**[plain]** [view plaincopy](http://blog.csdn.net/fanqipin/article/details/8480972)

1. int usb\_get\_configuration(struct usb\_device \*dev)
2. {
3. struct device \*ddev = &dev->dev;
4. int ncfg = dev->descriptor.bNumConfigurations;
5. int result = 0;
6. unsigned int cfgno, length;
7. unsigned char \*bigbuffer;
8. struct usb\_config\_descriptor \*desc;
10. cfgno = 0;
11. if (dev->authorized == 0)    /\* Not really an error \*/
12. goto out\_not\_authorized;
13. result = -ENOMEM;
14. if (ncfg > USB\_MAXCONFIG) {
15. dev\_warn(ddev, "too many configurations: %d, "
16. "using maximum allowed: %d\n", ncfg, USB\_MAXCONFIG);
17. dev->descriptor.bNumConfigurations = ncfg = USB\_MAXCONFIG;
18. }
20. if (ncfg < 1) {
21. dev\_err(ddev, "no configurations\n");
22. return -EINVAL;
23. }
25. length = ncfg \* sizeof(struct usb\_host\_config);
26. dev->config = kzalloc(length, GFP\_KERNEL);
27. if (!dev->config)
28. goto err2;
30. length = ncfg \* sizeof(char \*);
31. dev->rawdescriptors = kzalloc(length, GFP\_KERNEL);
32. if (!dev->rawdescriptors)
33. goto err2;
35. desc = kmalloc(USB\_DT\_CONFIG\_SIZE, GFP\_KERNEL);
36. if (!desc)
37. goto err2;
39. result = 0;
40. for (; cfgno < ncfg; cfgno++) {
41. /\* We grab just the first descriptor so we know how long
42. \* the whole configuration is \*/
43. result = usb\_get\_descriptor(dev, USB\_DT\_CONFIG, cfgno,
44. desc, USB\_DT\_CONFIG\_SIZE);
45. if (result < 0) {
46. dev\_err(ddev, "unable to read config index %d "
47. "descriptor/%s: %d\n", cfgno, "start", result);
48. dev\_err(ddev, "chopping to %d config(s)\n", cfgno);
49. dev->descriptor.bNumConfigurations = cfgno;
50. break;
51. } else if (result < 4) {
52. dev\_err(ddev, "config index %d descriptor too short "
53. "(expected %i, got %i)\n", cfgno,
54. USB\_DT\_CONFIG\_SIZE, result);
55. result = -EINVAL;
56. goto err;
57. }
58. length = max((int) le16\_to\_cpu(desc->wTotalLength),
59. USB\_DT\_CONFIG\_SIZE);
61. /\* Now that we know the length, get the whole thing \*/
62. bigbuffer = kmalloc(length, GFP\_KERNEL);
63. if (!bigbuffer) {
64. result = -ENOMEM;
65. goto err;
66. }
67. result = usb\_get\_descriptor(dev, USB\_DT\_CONFIG, cfgno,
68. bigbuffer, length);
69. if (result < 0) {
70. dev\_err(ddev, "unable to read config index %d "
71. "descriptor/%s\n", cfgno, "all");
72. kfree(bigbuffer);
73. goto err;
74. }
75. if (result < length) {
76. dev\_warn(ddev, "config index %d descriptor too short "
77. "(expected %i, got %i)\n", cfgno, length, result);
78. length = result;
79. }
81. dev->rawdescriptors[cfgno] = bigbuffer;
83. result = usb\_parse\_configuration(dev, cfgno,
84. &dev->config[cfgno], bigbuffer, length);
85. if (result < 0) {
86. ++cfgno;
87. goto err;
88. }
89. }
90. result = 0;
92. err:
93. kfree(desc);
94. out\_not\_authorized:
95. dev->descriptor.bNumConfigurations = cfgno;
96. err2:
97. if (result == -ENOMEM)
98. dev\_err(ddev, "out of memory\n");
99. return result;
100. }

这个函数主要分成二部分，第一部分是向usb device侧获取设备的配置，接口，端口信息，这部分由第1-79行完成，首先，它为这些信息申请存放空间，然后像之前获取设备描述符一样，先发送一个9 个字节的请求，用来获取配置，接口，端口等描述符的总长度，最后根据得到的总长度去得到完成 的设备配置，接口，端口信息；第二部分是把获取到的这个信息按照类别分别进行分类，并存到相应的结构中；  
11-12行，如果usb设备没有得到授权，则直接退出；

14-18行，如果usb设备的配置个数大于最大允许配置数8，则发出警告，并把设备的配置个数改成最大允许配置数8  
20-23行，如果usb设备没有配置，则不允许，直接返回错误信息；

25-33行，根据配置数，申请usb配置结构usb\_host\_config,并为用于存放配置结构的指针数据申请空间；

35-37行，申请用于暂时存放配置描述符的结构空间；

40-89行，依次获取usb设备的配置信息；

43-57行，由于接口，端口，产家专有等描述符信息是直接跟在配置描述符后面的，配置描述符进里有一项wTotalLength，专门用来表示配置，接口，端口，产品和产家专有等描述符的总长度，所以它先获取配置描述符信息，从而得到总长度；wTotallLength在配置描述符里的第3-4个字节位置，所以在获取配置描述时，只要得到的数据大于等于4个字节就可以，但如果少于4字节，就表示出错；

58-59行，得到wTotallLength;

62-66行，根据得到的wTotallLength，申请用于存放这些信息的内存空间；

67-74行，根据得到的wTotallLength，去获取完整的配置，接口，端口等描述符信息,把这些信息存放到bigbuffer里；

81行，把用于存放配置等信息的地址保存在之前申请的rawdescriptors里；

83-88行，通过usb\_parse\_configuration函数，把获得的配置等信息按照类型进行分类 ；

这里的usb\_get\_descriptor，只是向设备发送一个获取配置信息的请求，相对比较简单，所以就不深入讲解，而usb\_parse\_configuration函数却比较复杂，这里会对它进一步讲解；

**[plain]** [view plaincopy](http://blog.csdn.net/fanqipin/article/details/8480972)

1. static int usb\_parse\_configuration(struct usb\_device \*dev, int cfgidx,
2. struct usb\_host\_config \*config, unsigned char \*buffer, int size)
3. {
4. struct device \*ddev = &dev->dev;
5. unsigned char \*buffer0 = buffer;
6. int cfgno;
7. int nintf, nintf\_orig;
8. int i, j, n;
9. struct usb\_interface\_cache \*intfc;
10. unsigned char \*buffer2;
11. int size2;
12. struct usb\_descriptor\_header \*header;
13. int len, retval;
14. u8 inums[USB\_MAXINTERFACES], nalts[USB\_MAXINTERFACES];
15. unsigned iad\_num = 0;
17. memcpy(&config->desc, buffer, USB\_DT\_CONFIG\_SIZE);
18. if (config->desc.bDescriptorType != USB\_DT\_CONFIG ||
19. config->desc.bLength < USB\_DT\_CONFIG\_SIZE) {
20. dev\_err(ddev, "invalid descriptor for config index %d: "
21. "type = 0x%X, length = %d\n", cfgidx,
22. config->desc.bDescriptorType, config->desc.bLength);
23. return -EINVAL;
24. }
25. cfgno = config->desc.bConfigurationValue;
27. buffer += config->desc.bLength;
28. size -= config->desc.bLength;
30. nintf = nintf\_orig = config->desc.bNumInterfaces;
31. if (nintf > USB\_MAXINTERFACES) {
32. dev\_warn(ddev, "config %d has too many interfaces: %d, "
33. "using maximum allowed: %d\n",
34. cfgno, nintf, USB\_MAXINTERFACES);
35. nintf = USB\_MAXINTERFACES;
36. }
38. /\* Go through the descriptors, checking their length and counting the
39. \* number of altsettings for each interface \*/
40. n = 0;
41. for ((buffer2 = buffer, size2 = size);
42. size2 > 0;
43. (buffer2 += header->bLength, size2 -= header->bLength)) {
45. if (size2 < sizeof(struct usb\_descriptor\_header)) {
46. dev\_warn(ddev, "config %d descriptor has %d excess "
47. "byte%s, ignoring\n",
48. cfgno, size2, plural(size2));
49. break;
50. }
52. header = (struct usb\_descriptor\_header \*) buffer2;
53. if ((header->bLength > size2) || (header->bLength < 2)) {
54. dev\_warn(ddev, "config %d has an invalid descriptor "
55. "of length %d, skipping remainder of the config\n",
56. cfgno, header->bLength);
57. break;
58. }
60. if (header->bDescriptorType == USB\_DT\_INTERFACE) {
61. struct usb\_interface\_descriptor \*d;
62. int inum;
64. d = (struct usb\_interface\_descriptor \*) header;
65. if (d->bLength < USB\_DT\_INTERFACE\_SIZE) {
66. dev\_warn(ddev, "config %d has an invalid "
67. "interface descriptor of length %d, "
68. "skipping\n", cfgno, d->bLength);
69. continue;
70. }
72. inum = d->bInterfaceNumber;
74. if ((dev->quirks & USB\_QUIRK\_HONOR\_BNUMINTERFACES) &&
75. n >= nintf\_orig) {
76. dev\_warn(ddev, "config %d has more interface "
77. "descriptors, than it declares in "
78. "bNumInterfaces, ignoring interface "
79. "number: %d\n", cfgno, inum);
80. continue;
81. }
83. if (inum >= nintf\_orig)
84. dev\_warn(ddev, "config %d has an invalid "
85. "interface number: %d but max is %d\n",
86. cfgno, inum, nintf\_orig - 1);
88. /\* Have we already encountered this interface?
89. \* Count its altsettings \*/
90. for (i = 0; i < n; ++i) {
91. if (inums[i] == inum)
92. break;
93. }
94. if (i < n) {
95. if (nalts[i] < 255)
96. ++nalts[i];
97. } else if (n < USB\_MAXINTERFACES) {
98. inums[n] = inum;
99. nalts[n] = 1;
100. ++n;
101. }
103. } else if (header->bDescriptorType ==
104. USB\_DT\_INTERFACE\_ASSOCIATION) {
105. if (iad\_num == USB\_MAXIADS) {
106. dev\_warn(ddev, "found more Interface "
107. "Association Descriptors "
108. "than allocated for in "
109. "configuration %d\n", cfgno);
110. } else {
111. config->intf\_assoc[iad\_num] =
112. (struct usb\_interface\_assoc\_descriptor
113. \*)header;
114. iad\_num++;
115. }
117. } else if (header->bDescriptorType == USB\_DT\_DEVICE ||
118. header->bDescriptorType == USB\_DT\_CONFIG)
119. dev\_warn(ddev, "config %d contains an unexpected "
120. "descriptor of type 0x%X, skipping\n",
121. cfgno, header->bDescriptorType);
123. }   /\* for ((buffer2 = buffer, size2 = size); ...) \*/
124. size = buffer2 - buffer;
125. config->desc.wTotalLength = cpu\_to\_le16(buffer2 - buffer0);
127. if (n != nintf)
128. dev\_warn(ddev, "config %d has %d interface%s, different from "
129. "the descriptor's value: %d\n",
130. cfgno, n, plural(n), nintf\_orig);
131. else if (n == 0)
132. dev\_warn(ddev, "config %d has no interfaces?\n", cfgno);
133. config->desc.bNumInterfaces = nintf = n;
135. /\* Check for missing interface numbers \*/
136. for (i = 0; i < nintf; ++i) {
137. for (j = 0; j < nintf; ++j) {
138. if (inums[j] == i)
139. break;
140. }
141. if (j >= nintf)
142. dev\_warn(ddev, "config %d has no interface number "
143. "%d\n", cfgno, i);
144. }
146. /\* Allocate the usb\_interface\_caches and altsetting arrays \*/
147. for (i = 0; i < nintf; ++i) {
148. j = nalts[i];
149. if (j > USB\_MAXALTSETTING) {
150. dev\_warn(ddev, "too many alternate settings for "
151. "config %d interface %d: %d, "
152. "using maximum allowed: %d\n",
153. cfgno, inums[i], j, USB\_MAXALTSETTING);
154. nalts[i] = j = USB\_MAXALTSETTING;
155. }
157. len = sizeof(\*intfc) + sizeof(struct usb\_host\_interface) \* j;
158. config->intf\_cache[i] = intfc = kzalloc(len, GFP\_KERNEL);
159. if (!intfc)
160. return -ENOMEM;
161. kref\_init(&intfc->ref);
162. }
164. /\* FIXME: parse the BOS descriptor \*/
166. /\* Skip over any Class Specific or Vendor Specific descriptors;
167. \* find the first interface descriptor \*/
168. config->extra = buffer;
169. i = find\_next\_descriptor(buffer, size, USB\_DT\_INTERFACE,
170. USB\_DT\_INTERFACE, &n);
171. config->extralen = i;
172. if (n > 0)
173. dev\_dbg(ddev, "skipped %d descriptor%s after %s\n",
174. n, plural(n), "configuration");
175. buffer += i;
176. size -= i;
178. /\* Parse all the interface/altsetting descriptors \*/
179. while (size > 0) {
180. retval = usb\_parse\_interface(ddev, cfgno, config,
181. buffer, size, inums, nalts);
182. if (retval < 0)
183. return retval;
185. buffer += retval;
186. size -= retval;
187. }
189. /\* Check for missing altsettings \*/
190. for (i = 0; i < nintf; ++i) {
191. intfc = config->intf\_cache[i];
192. for (j = 0; j < intfc->num\_altsetting; ++j) {
193. for (n = 0; n < intfc->num\_altsetting; ++n) {
194. if (intfc->altsetting[n].desc.
195. bAlternateSetting == j)
196. break;
197. }
198. if (n >= intfc->num\_altsetting)
199. dev\_warn(ddev, "config %d interface %d has no "
200. "altsetting %d\n", cfgno, inums[i], j);
201. }
202. }
204. return 0;
205. }

获取得到的信息里它包含了多种信息：配置，相关联接口，接口，端口等；这个函数的作用就是将这些信息把它补全到之前申请的usb\_host\_config结构里，把各类信息放到相应的结构中；

获取得到的信息最前面就是配置信息，所以17行，根据配置信息长度把配置描述符信息保存到usb\_host\_config中的desc中；

18-24行，如果获得的信息最前面不是配置描述符信息或描述符信息长度少于标准数据长度，则出错，直接返回；

25行，从配置描述符里得到当前配置值；

27-28行，更新buffer地址和size大小，从而可以获得接下来的信息；

30-36行，取得当前配置里接口个数，如果接口个数大于允许的最大接口数据，发出警告，并重新设置设备接口数；

40-123行，遍历当前配置信息，统计得到当前配置下每个接口对应的接口设置数；

其中n用来表示接口数，inums和nalts数组用来分别存放接口号及该接口号对应的接口设置数；

45-58行，根据usb的描述符头里的长度对buffer里的数据进行判断，判断是否满足长度需求，如果不满足，则跳出当前统计循环；

USB3.0协议引进了一个interrface association descriptor概念，它用来描述一个复合型接口，复合型接口是指由2个或以上接口组成一个功能的接口。如果存在这类描述符，则它是紧跟着配置描述符的；

60-86行，如果配置描述符后紧跟着的是接口描述符，则根据接口描述符对长度和接口号进行判断，判断这个接口描述符是否合法;

90-93行，遍历inums数组，判断接口描述符中接口号是否已经存在inums数组里；

94-101行，如果在inums里找到和当前接口号相同项，并且这个接口对应的接口设置数量没有超过255，则增加接口设置数。如果不能在inums里找到与当前接口与相同项，表示这是一个新的接口，如果接口数n没有超过最大接口数32，则将当前接口号保存到inums数组里，并设置对应的接口设置数为1；

103-115行，如果当前描述符是interface association descriptor，则把它保存到配置结构intf\_assoc里；

117-123行，在当前配置描述下，如果还有其它配置或设备描述符，则发出警告；

124-125行，重新获取配置等描述符信息长度；

127-133行，对统计出来的接口数进行判断，如果n与配置描述中的接口数不相等或n等于0，则发生警告；

136-144行，对inums里的接口号进行检查，查看对就的接口号是否存在；

147-162行，为当前接口和接口下的接口设置申请内存空间，并把它存放在配置里的intf\_cache结构里，intf\_cache本身 是一个指针数组，它用来存放接口和接口设置这个内存空间的地址；

169行，通过find\_next\_descriptor函数从buffer里查找类型为接口的描述符，返回值i表示buffer与接口描述符描述符之间数据长度，n用来表示buffer与接口描述符之间需要跳过的其它描述符个数。buffer最开始的地方有可能不是接口描述符，所以要去寻找第一次出现接口描述符类型的位置，这个位置可以由返回值来确定，即接口描述符开始地址=buffer+返回值i；

175-176行，如果在接口描述符前面还存在其它数据，则更新buffer和size,以确保接下来buffer就为接口描述符首地址；

179-187行，接下来由usb\_parse\_interface函数对buffer里的接口和端口等信息进行分解；

**[plain]** [view plaincopy](http://blog.csdn.net/fanqipin/article/details/8480972)

1. static int usb\_parse\_interface(struct device \*ddev, int cfgno,
2. struct usb\_host\_config \*config, unsigned char \*buffer, int size,
3. u8 inums[], u8 nalts[])
4. {
5. unsigned char \*buffer0 = buffer;
6. struct usb\_interface\_descriptor \*d;
7. int inum, asnum;
8. struct usb\_interface\_cache \*intfc;
9. struct usb\_host\_interface \*alt;
10. int i, n;
11. int len, retval;
12. int num\_ep, num\_ep\_orig;
14. d = (struct usb\_interface\_descriptor \*) buffer;
15. buffer += d->bLength;
16. size -= d->bLength;
18. if (d->bLength < USB\_DT\_INTERFACE\_SIZE)
19. goto skip\_to\_next\_interface\_descriptor;
21. /\* Which interface entry is this? \*/
22. intfc = NULL;
23. inum = d->bInterfaceNumber;
24. for (i = 0; i < config->desc.bNumInterfaces; ++i) {
25. if (inums[i] == inum) {
26. intfc = config->intf\_cache[i];
27. break;
28. }
29. }
30. if (!intfc || intfc->num\_altsetting >= nalts[i])
31. goto skip\_to\_next\_interface\_descriptor;
33. /\* Check for duplicate altsetting entries \*/
34. asnum = d->bAlternateSetting;
35. for ((i = 0, alt = &intfc->altsetting[0]);
36. i < intfc->num\_altsetting;
37. (++i, ++alt)) {
38. if (alt->desc.bAlternateSetting == asnum) {
39. dev\_warn(ddev, "Duplicate descriptor for config %d "
40. "interface %d altsetting %d, skipping\n",
41. cfgno, inum, asnum);
42. goto skip\_to\_next\_interface\_descriptor;
43. }
44. }
46. ++intfc->num\_altsetting;
47. memcpy(&alt->desc, d, USB\_DT\_INTERFACE\_SIZE);
49. /\* Skip over any Class Specific or Vendor Specific descriptors;
50. \* find the first endpoint or interface descriptor \*/
51. alt->extra = buffer;
52. i = find\_next\_descriptor(buffer, size, USB\_DT\_ENDPOINT,
53. USB\_DT\_INTERFACE, &n);
54. alt->extralen = i;
55. if (n > 0)
56. dev\_dbg(ddev, "skipped %d descriptor%s after %s\n",
57. n, plural(n), "interface");
58. buffer += i;
59. size -= i;
61. /\* Allocate space for the right(?) number of endpoints \*/
62. num\_ep = num\_ep\_orig = alt->desc.bNumEndpoints;
63. alt->desc.bNumEndpoints = 0;     /\* Use as a counter \*/
64. if (num\_ep > USB\_MAXENDPOINTS) {
65. dev\_warn(ddev, "too many endpoints for config %d interface %d "
66. "altsetting %d: %d, using maximum allowed: %d\n",
67. cfgno, inum, asnum, num\_ep, USB\_MAXENDPOINTS);
68. num\_ep = USB\_MAXENDPOINTS;
69. }
71. if (num\_ep > 0) {
72. /\* Can't allocate 0 bytes \*/
73. len = sizeof(struct usb\_host\_endpoint) \* num\_ep;
74. alt->endpoint = kzalloc(len, GFP\_KERNEL);
75. if (!alt->endpoint)
76. return -ENOMEM;
77. }
79. /\* Parse all the endpoint descriptors \*/
80. n = 0;
81. while (size > 0) {
82. if (((struct usb\_descriptor\_header \*) buffer)->bDescriptorType
83. == USB\_DT\_INTERFACE)
84. break;
85. retval = usb\_parse\_endpoint(ddev, cfgno, inum, asnum, alt,
86. num\_ep, buffer, size);
87. if (retval < 0)
88. return retval;
89. ++n;
91. buffer += retval;
92. size -= retval;
93. }
95. if (n != num\_ep\_orig)
96. dev\_warn(ddev, "config %d interface %d altsetting %d has %d "
97. "endpoint descriptor%s, different from the interface "
98. "descriptor's value: %d\n",
99. cfgno, inum, asnum, n, plural(n), num\_ep\_orig);
100. return buffer - buffer0;
102. skip\_to\_next\_interface\_descriptor:
103. i = find\_next\_descriptor(buffer, size, USB\_DT\_INTERFACE,
104. USB\_DT\_INTERFACE, NULL);
105. return buffer - buffer0 + i;
106. }

14-19行，更新buffer和size，并判断接口描述符的长度是否正常，如果不正常则查找下一个接口描述符；  
23-29行，根据当前接口描述符中的接口号来得到用于存放接口信息的intf\_cache;

34-44行，把接口描述符中的接口设置值与存放在intf\_cache中的接口设置值进行比较，如果找到相同的设置值，则表示这个接口设置已经保存在intf\_cache里了，则跳过当前接口设置，直接去查找下一个接口；如果找不到相同的，则表示这是一个新的接口设置，则通过for 循环中++alt来确定接口设置存放位置；

46-47行，更新接口设置值，并将接口描述符保存到相应接口下；

52-59行，在剩下的buffer里找到endpoint或interface描述符首次出现位置，并根据返回值来更新buffer和size;

62-77行，得到当前接口设置下有多少个endpoint端口，并为这些端口申请空间；

81-93行，如果在接口描述符后面紧跟着端口描述符，则通过usb\_parse\_endpoint对端口描述符进行分解，并将它保存在之前申请的endpoint里；

**[plain]** [view plaincopy](http://blog.csdn.net/fanqipin/article/details/8480972)

1. static int usb\_parse\_endpoint(struct device \*ddev, int cfgno, int inum,
2. int asnum, struct usb\_host\_interface \*ifp, int num\_ep,
3. unsigned char \*buffer, int size)
4. {
5. unsigned char \*buffer0 = buffer;
6. struct usb\_endpoint\_descriptor \*d;
7. struct usb\_host\_endpoint \*endpoint;
8. int n, i, j, retval;
10. d = (struct usb\_endpoint\_descriptor \*) buffer;
11. buffer += d->bLength;
12. size -= d->bLength;
14. if (d->bLength >= USB\_DT\_ENDPOINT\_AUDIO\_SIZE)
15. n = USB\_DT\_ENDPOINT\_AUDIO\_SIZE;
16. else if (d->bLength >= USB\_DT\_ENDPOINT\_SIZE)
17. n = USB\_DT\_ENDPOINT\_SIZE;
18. else {
19. dev\_warn(ddev, "config %d interface %d altsetting %d has an "
20. "invalid endpoint descriptor of length %d, skipping\n",
21. cfgno, inum, asnum, d->bLength);
22. goto skip\_to\_next\_endpoint\_or\_interface\_descriptor;
23. }
25. i = d->bEndpointAddress & ~USB\_ENDPOINT\_DIR\_MASK;
26. if (i >= 16 || i == 0) {
27. dev\_warn(ddev, "config %d interface %d altsetting %d has an "
28. "invalid endpoint with address 0x%X, skipping\n",
29. cfgno, inum, asnum, d->bEndpointAddress);
30. goto skip\_to\_next\_endpoint\_or\_interface\_descriptor;
31. }
33. /\* Only store as many endpoints as we have room for \*/
34. if (ifp->desc.bNumEndpoints >= num\_ep)
35. goto skip\_to\_next\_endpoint\_or\_interface\_descriptor;
37. endpoint = &ifp->endpoint[ifp->desc.bNumEndpoints];
38. ++ifp->desc.bNumEndpoints;
40. memcpy(&endpoint->desc, d, n);
41. INIT\_LIST\_HEAD(&endpoint->urb\_list);
43. /\* Fix up bInterval values outside the legal range. Use 32 ms if no
44. \* proper value can be guessed. \*/
45. i = 0;      /\* i = min, j = max, n = default \*/
46. j = 255;
47. if (usb\_endpoint\_xfer\_int(d)) {
48. i = 1;
49. switch (to\_usb\_device(ddev)->speed) {
50. case USB\_SPEED\_SUPER:
51. case USB\_SPEED\_HIGH:
52. /\* Many device manufacturers are using full-speed
53. \* bInterval values in high-speed interrupt endpoint
54. \* descriptors. Try to fix those and fall back to a
55. \* 32 ms default value otherwise. \*/
56. n = fls(d->bInterval\*8);
57. if (n == 0)
58. n = 9;  /\* 32 ms = 2^(9-1) uframes \*/
59. j = 16;
60. break;
61. default:        /\* USB\_SPEED\_FULL or \_LOW \*/
62. /\* For low-speed, 10 ms is the official minimum.
63. \* But some "overclocked" devices might want faster
64. \* polling so we'll allow it. \*/
65. n = 32;
66. break;
67. }
68. } else if (usb\_endpoint\_xfer\_isoc(d)) {
69. i = 1;
70. j = 16;
71. switch (to\_usb\_device(ddev)->speed) {
72. case USB\_SPEED\_HIGH:
73. n = 9;      /\* 32 ms = 2^(9-1) uframes \*/
74. break;
75. default:        /\* USB\_SPEED\_FULL \*/
76. n = 6;      /\* 32 ms = 2^(6-1) frames \*/
77. break;
78. }
79. }
80. if (d->bInterval < i || d->bInterval > j) {
81. dev\_warn(ddev, "config %d interface %d altsetting %d "
82. "endpoint 0x%X has an invalid bInterval %d, "
83. "changing to %d\n",
84. cfgno, inum, asnum,
85. d->bEndpointAddress, d->bInterval, n);
86. endpoint->desc.bInterval = n;
87. }
89. /\* Some buggy low-speed devices have Bulk endpoints, which is
90. \* explicitly forbidden by the USB spec.  In an attempt to make
91. \* them usable, we will try treating them as Interrupt endpoints.
92. \*/
93. if (to\_usb\_device(ddev)->speed == USB\_SPEED\_LOW &&
94. usb\_endpoint\_xfer\_bulk(d)) {
95. dev\_warn(ddev, "config %d interface %d altsetting %d "
96. "endpoint 0x%X is Bulk; changing to Interrupt\n",
97. cfgno, inum, asnum, d->bEndpointAddress);
98. endpoint->desc.bmAttributes = USB\_ENDPOINT\_XFER\_INT;
99. endpoint->desc.bInterval = 1;
100. if (le16\_to\_cpu(endpoint->desc.wMaxPacketSize) > 8)
101. endpoint->desc.wMaxPacketSize = cpu\_to\_le16(8);
102. }
104. /\*
105. \* Some buggy high speed devices have bulk endpoints using
106. \* maxpacket sizes other than 512.  High speed HCDs may not
107. \* be able to handle that particular bug, so let's warn...
108. \*/
109. if (to\_usb\_device(ddev)->speed == USB\_SPEED\_HIGH
110. && usb\_endpoint\_xfer\_bulk(d)) {
111. unsigned maxp;
113. maxp = le16\_to\_cpu(endpoint->desc.wMaxPacketSize) & 0x07ff;
114. if (maxp != 512)
115. dev\_warn(ddev, "config %d interface %d altsetting %d "
116. "bulk endpoint 0x%X has invalid maxpacket %d\n",
117. cfgno, inum, asnum, d->bEndpointAddress,
118. maxp);
119. }
121. /\* Parse a possible SuperSpeed endpoint companion descriptor \*/
122. if (to\_usb\_device(ddev)->speed == USB\_SPEED\_SUPER)
123. usb\_parse\_ss\_endpoint\_companion(ddev, cfgno,
124. inum, asnum, endpoint, buffer, size);
126. /\* Skip over any Class Specific or Vendor Specific descriptors;
127. \* find the next endpoint or interface descriptor \*/
128. endpoint->extra = buffer;
129. i = find\_next\_descriptor(buffer, size, USB\_DT\_ENDPOINT,
130. USB\_DT\_INTERFACE, &n);
131. endpoint->extralen = i;
132. retval = buffer - buffer0 + i;
133. if (n > 0)
134. dev\_dbg(ddev, "skipped %d descriptor%s after %s\n",
135. n, plural(n), "endpoint");
136. return retval;
138. skip\_to\_next\_endpoint\_or\_interface\_descriptor:
139. i = find\_next\_descriptor(buffer, size, USB\_DT\_ENDPOINT,
140. USB\_DT\_INTERFACE, NULL);
141. return buffer - buffer0 + i;
142. }

10-23行，根据端口描述符长度去更新buffer和size,并确认端口描述符的长度；

25-35行，确认端口的地址及端口数量，标准的端口地址是从1到15；

37-41行，将接口描述符中的端口数当成接口下端口数组endpoint的索引，获取用于存放当前endpoint信息的地址，并把当前的端口描述符保存到endpoint下；

45-87行，对于中断和等时传输类型的端口，根据usb设备速度设置时间时隔interval;

93-102行，在usb2.0协议中有定义说：在低速时是没有bulk类型的端口的，但实际上有可能存在这种情况，这里的话，我们把这个bulk端口当做interrupt类型端口使用；

109-119行，对于运行在高速下的bulk端口，usb2.0定义这类端口最大传输长度为512，如果不为512则发出警告；

122-124行，如果当前设备是超速usb设备，则还要usb\_parse\_ss\_endpoint\_companion取得用于描述高速设备的描述符；

129-132行，继续分解端口和接口描述符；

由这个配置描述符分解过程可以知道，配置描述符里包含下面内容：配置描述符，关联接口描述符或接口描述符，端口描述符对于超速设备还有用于描述超速设备的描述符，这个分解过程验证了：

usb设备由一个或多个usb 配置组成，usb配置由一个或多个usb接口组成，usb接口下包含多个usb接口设置，usb接口设置由多个usb端口组成；

讲到这里usb\_enumerate\_device中的得要部分已经讲完，接下来回到usb\_new\_device继续讲解；

31-32行，根据usb设备的端口获取usb设备的设备号；

35行，通过announce\_device输出一些usb设备信息；

42行，将usb设备通过device\_add添加到系统中。

到这里为止，usb设备枚举第一阶段算是已经完成，这部分主要工作是：遍历hub下的各个port，查看这些port是否有新设备加入，如果有新设备加入，则为这个设备申请资源，并去获取设备描述符，配置描述符去完善这个usb 设备，最后将这个port上的usb 设备注册到系统中；

当usb设备通过device\_add注册到usb 总线上后，系统会去遍历usb总线上的所有驱动，寻找与usb设备驱动，通过usb\_match\_device进行匹配。对于usb设备，它的驱动应该是usb\_generic\_driver，当驱动与设备匹配成功后，就会调用usb\_generic\_driver中的generic\_probe函数，它个函数会从usb设备中找出一个合适 的usb配置，并对usb设备进行配置；

**[plain]** [view plaincopy](http://blog.csdn.net/fanqipin/article/details/8480972)

1. static int generic\_probe(struct usb\_device \*udev)
2. {
3. int err, c;
5. /\* Choose and set the configuration.  This registers the interfaces
6. \* with the driver core and lets interface drivers bind to them.
7. \*/
8. if (usb\_device\_is\_owned(udev))
9. ;       /\* Don't configure if the device is owned \*/
10. else if (udev->authorized == 0)
11. dev\_err(&udev->dev, "Device is not authorized for usage\n");
12. else {
13. c = usb\_choose\_configuration(udev);
14. if (c >= 0) {
15. err = usb\_set\_configuration(udev, c);
16. if (err) {
17. dev\_err(&udev->dev, "can't set config #%d, error %d\n",
18. c, err);
19. /\* This need not be fatal.  The user can try to
20. \* set other configurations. \*/
21. }
22. }
23. }
24. /\* USB device state == configured ... usable \*/
25. usb\_notify\_add\_device(udev);
27. return 0;
28. }

generic\_probe函数由三部分程序构成：第一是由usb\_choose\_configuration选择一个适合的usb配置；第二是根据选择得到的配置值配置usb设备；第三是如果系统支持usb class类设备文件系统，则在/sys/usb\_device/下创建usb设备的设备属性文件；

选择配置函数usb\_choose\_configuration中有许多注释信息，在这里我已经把它们都去掉了，具体程序如下所示：

**[plain]** [view plaincopy](http://blog.csdn.net/fanqipin/article/details/8480972)

1. int usb\_choose\_configuration(struct usb\_device \*udev)
2. {
3. int i;
4. int num\_configs;
5. int insufficient\_power = 0;
6. struct usb\_host\_config \*c, \*best;
8. best = NULL;
9. c = udev->config;
10. num\_configs = udev->descriptor.bNumConfigurations;
11. for (i = 0; i < num\_configs; (i++, c++)) {
12. struct usb\_interface\_descriptor \*desc = NULL;
14. /\* It's possible that a config has no interfaces! \*/
15. if (c->desc.bNumInterfaces > 0)
16. desc = &c->intf\_cache[0]->altsetting->desc;

19. /\* Rule out configs that draw too much bus current \*/
20. if (c->desc.bMaxPower \* 2 > udev->bus\_mA) {
21. insufficient\_power++;
22. continue;
23. }
25. if (i == 0 && num\_configs > 1 && desc &&
26. (is\_rndis(desc) || is\_activesync(desc))) {
27. #if !defined(CONFIG\_USB\_NET\_RNDIS\_HOST) && !defined(CONFIG\_USB\_NET\_RNDIS\_HOST\_MODULE)
28. continue;
29. #else
30. best = c;
31. #endif
32. } else if (udev->descriptor.bDeviceClass !=
33. USB\_CLASS\_VENDOR\_SPEC &&
34. (desc && desc->bInterfaceClass !=
35. USB\_CLASS\_VENDOR\_SPEC)) {
36. best = c;
37. break;
38. }
40. /\* If all the remaining configs are vendor-specific,
41. \* choose the first one. \*/
42. else if (!best)
43. best = c;
44. }
46. if (insufficient\_power > 0)
47. dev\_info(&udev->dev, "rejected %d configuration%s "
48. "due to insufficient available bus power\n",
49. insufficient\_power, plural(insufficient\_power));
51. if (best) {
52. i = best->desc.bConfigurationValue;
53. dev\_dbg(&udev->dev,
54. "configuration #%d chosen from %d choice%s\n",
55. i, num\_configs, plural(num\_configs));
56. } else {
57. i = -1;
58. dev\_warn(&udev->dev,
59. "no configuration chosen from %d choice%s\n",
60. num\_configs, plural(num\_configs));
61. }
62. return i;
63. }

15-16行，如果当前配置下有接口，则选择得到接口0中接口设置0的接口描述符；

20-23行，判断当前usb设备从hub上可以获得的电流是滞能够满足当前配置，如果不能满足则选择下一个配置；

25-38行，对于一些micsoft公司喜欢的非标准设备配置，除非指定说需要，否则不考虑这些配置；

32-38行，选择非生产产家规定类的配置比由生产产家规定类的配置更合适；

42-43行，如果只有生产 产家规定类的配置，那么选择这睦配置中的第一个配置；

46-49行，如果这些配置中所需要的电流比较从hub上获得的允许值大，就会发出警告；

51-62行，选择好合适 的配置值后，函数usb\_choose\_configuration返回；

如果得到了合适的配置值，则通过usb\_set\_configuration将这个值对usb设备进行配置；

**[plain]** [view plaincopy](http://blog.csdn.net/fanqipin/article/details/8480972)

1. int usb\_set\_configuration(struct usb\_device \*dev, int configuration)
2. {
3. int i, ret;
4. struct usb\_host\_config \*cp = NULL;
5. struct usb\_interface \*\*new\_interfaces = NULL;
6. struct usb\_hcd \*hcd = bus\_to\_hcd(dev->bus);
7. int n, nintf;
9. if (dev->authorized == 0 || configuration == -1)
10. configuration = 0;
11. else {
12. for (i = 0; i < dev->descriptor.bNumConfigurations; i++) {
13. if (dev->config[i].desc.bConfigurationValue ==
14. configuration) {
15. cp = &dev->config[i];
16. break;
17. }
18. }
19. }
20. if ((!cp && configuration != 0))
21. return -EINVAL;
23. if (cp && configuration == 0)
24. dev\_warn(&dev->dev, "config 0 descriptor??\n");
26. n = nintf = 0;
27. if (cp) {
28. nintf = cp->desc.bNumInterfaces;
29. new\_interfaces = kmalloc(nintf \* sizeof(\*new\_interfaces),
30. GFP\_NOIO);
31. if (!new\_interfaces) {
32. dev\_err(&dev->dev, "Out of memory\n");
33. return -ENOMEM;
34. }
36. for (; n < nintf; ++n) {
37. new\_interfaces[n] = kzalloc(
38. sizeof(struct usb\_interface),
39. GFP\_NOIO);
40. if (!new\_interfaces[n]) {
41. dev\_err(&dev->dev, "Out of memory\n");
42. ret = -ENOMEM;
43. free\_interfaces:
44. while (--n >= 0)
45. kfree(new\_interfaces[n]);
46. kfree(new\_interfaces);
47. return ret;
48. }
49. }
51. i = dev->bus\_mA - cp->desc.bMaxPower \* 2;
52. if (i < 0)
53. dev\_warn(&dev->dev, "new config #%d exceeds power "
54. "limit by %dmA\n",
55. configuration, -i);
56. }
58. /\* Wake up the device so we can send it the Set-Config request \*/
59. ret = usb\_autoresume\_device(dev);
60. if (ret)
61. goto free\_interfaces;
63. /\* if it's already configured, clear out old state first.
64. \* getting rid of old interfaces means unbinding their drivers.
65. \*/
66. mutex\_lock(hcd->bandwidth\_mutex);
67. if (dev->state != USB\_STATE\_ADDRESS)
68. usb\_disable\_device(dev, 1); /\* Skip ep0 \*/
70. /\* Get rid of pending async Set-Config requests for this device \*/
71. cancel\_async\_set\_config(dev);
73. ret = usb\_hcd\_alloc\_bandwidth(dev, cp, NULL, NULL);
74. if (ret < 0) {
75. mutex\_unlock(hcd->bandwidth\_mutex);
76. usb\_autosuspend\_device(dev);
77. goto free\_interfaces;
78. }
80. ret = usb\_control\_msg(dev, usb\_sndctrlpipe(dev, 0),
81. USB\_REQ\_SET\_CONFIGURATION, 0, configuration, 0,
82. NULL, 0, USB\_CTRL\_SET\_TIMEOUT);
83. if (ret < 0) {
84. /\* All the old state is gone, so what else can we do?
85. \* The device is probably useless now anyway.
86. \*/
87. cp = NULL;
88. }
90. dev->actconfig = cp;
91. if (!cp) {
92. usb\_set\_device\_state(dev, USB\_STATE\_ADDRESS);
93. usb\_hcd\_alloc\_bandwidth(dev, NULL, NULL, NULL);
94. mutex\_unlock(hcd->bandwidth\_mutex);
95. usb\_autosuspend\_device(dev);
96. goto free\_interfaces;
97. }
98. mutex\_unlock(hcd->bandwidth\_mutex);
99. usb\_set\_device\_state(dev, USB\_STATE\_CONFIGURED);
101. /\* Initialize the new interface structures and the
102. \* hc/hcd/usbcore interface/endpoint state.
103. \*/
104. for (i = 0; i < nintf; ++i) {
105. struct usb\_interface\_cache \*intfc;
106. struct usb\_interface \*intf;
107. struct usb\_host\_interface \*alt;
109. cp->interface[i] = intf = new\_interfaces[i];
110. intfc = cp->intf\_cache[i];
111. intf->altsetting = intfc->altsetting;
112. intf->num\_altsetting = intfc->num\_altsetting;
113. intf->intf\_assoc = find\_iad(dev, cp, i);
114. kref\_get(&intfc->ref);
116. alt = usb\_altnum\_to\_altsetting(intf, 0);
118. /\* No altsetting 0?  We'll assume the first altsetting.
119. \* We could use a GetInterface call, but if a device is
120. \* so non-compliant that it doesn't have altsetting 0
121. \* then I wouldn't trust its reply anyway.
122. \*/
123. if (!alt)
124. alt = &intf->altsetting[0];
126. intf->cur\_altsetting = alt;
127. usb\_enable\_interface(dev, intf, true);
128. intf->dev.parent = &dev->dev;
129. intf->dev.driver = NULL;
130. intf->dev.bus = &usb\_bus\_type;
131. intf->dev.type = &usb\_if\_device\_type;
132. intf->dev.groups = usb\_interface\_groups;
133. intf->dev.dma\_mask = dev->dev.dma\_mask;
134. INIT\_WORK(&intf->reset\_ws, \_\_usb\_queue\_reset\_device);
135. intf->minor = -1;
136. device\_initialize(&intf->dev);
137. pm\_runtime\_no\_callbacks(&intf->dev);
138. dev\_set\_name(&intf->dev, "%d-%s:%d.%d",
139. dev->bus->busnum, dev->devpath,
140. configuration, alt->desc.bInterfaceNumber);
141. }
142. kfree(new\_interfaces);
144. if (cp->string == NULL &&
145. !(dev->quirks & USB\_QUIRK\_CONFIG\_INTF\_STRINGS))
146. cp->string = usb\_cache\_string(dev, cp->desc.iConfiguration);
148. for (i = 0; i < nintf; ++i) {
149. struct usb\_interface \*intf = cp->interface[i];
151. dev\_dbg(&dev->dev,
152. "adding %s (config #%d, interface %d)\n",
153. dev\_name(&intf->dev), configuration,
154. intf->cur\_altsetting->desc.bInterfaceNumber);
155. device\_enable\_async\_suspend(&intf->dev);
156. ret = device\_add(&intf->dev);
157. if (ret != 0) {
158. dev\_err(&dev->dev, "device\_add(%s) --> %d\n",
159. dev\_name(&intf->dev), ret);
160. continue;
161. }
162. create\_intf\_ep\_devs(intf);
163. }
165. usb\_autosuspend\_device(dev);
166. return 0;
167. }

9-24行，根据配置值得到usb配置；

27-49行，根据接口数申请用于暂时存放当前配置下接口的内存空间，这里包含两部分，首先，new\_interfaces是一个指针数组，所以要根据接口数为这个指针数据申请空间，其次还要为接口这个结构本身申请空间；

51-57行，如果当前配置不能满足电源供给需求，则发出警告；

80-88行，向usb device 侧发送设备配置请求；

99行，如果配置成功，则将usb 设备设置成USB\_STATE\_CONFIGURED态；

104-141行，对当前配置下的各接口进行初始化， 完善配置结构下interface结构，并初始化各接口设备对就的设备模型；

148-163行，将当前配置下的各接口设备进行注册，并创建端口设备文件；

到这里为止，usb的设备枚举基本上已经完成 ，接下来系统会根据各接口的类型为它分配接口驱动，这里所说的接口驱动就是驱动工程师所需要去完成 的关于usb设备功能使用程序。当然，如果当前port上接的是一个hub，则它会调用hub\_probe对hub进行配置。