

The logo features the text "FAR SIGHT" in white, bold, sans-serif capital letters. A red, stylized vertical line separates the word "FAR" from "SIGHT". The text is centered within a dark green, textured, downward-pointing triangle that has a 3D effect with lighter green highlights on its sides.

FAR SIGHT

嵌入式培训专家

USB 设备驱动开发

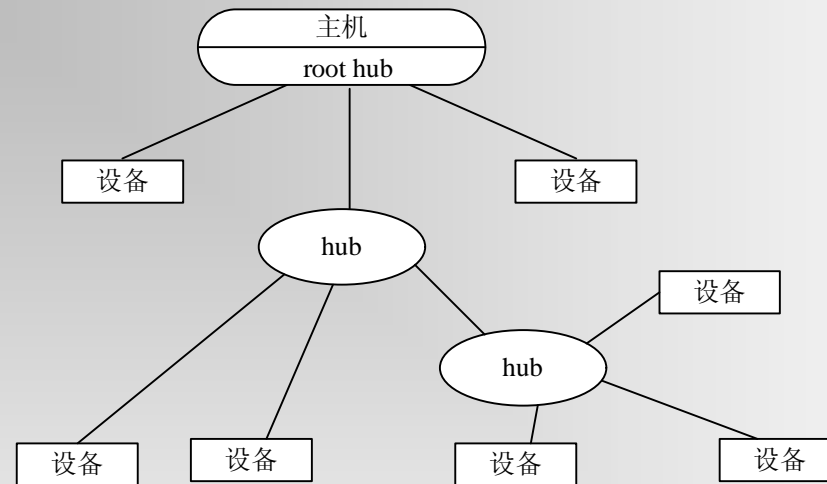
主讲：宋宝华

www.farsight.com.cn

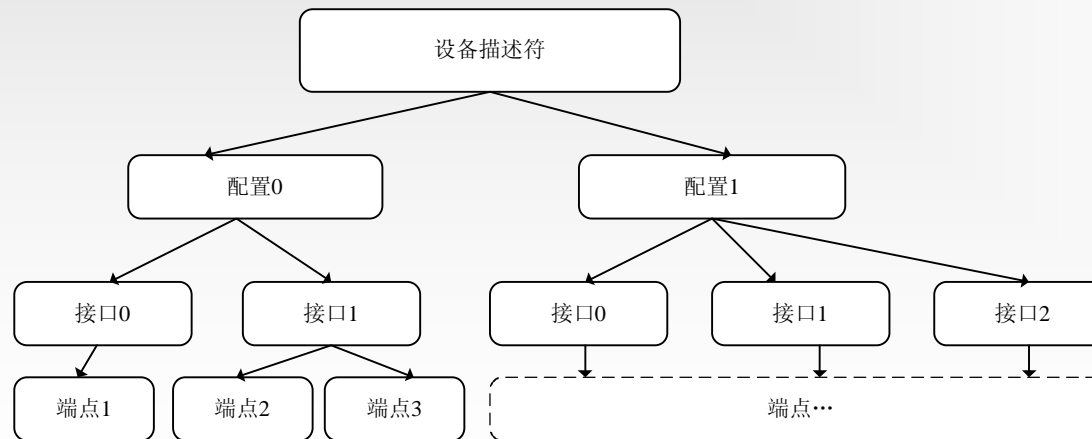
今天的内容

- Ø 1.USB及驱动框架简介
 - Ø 1.1 USB协议
 - Ø 1.2 USB驱动的体系结构
- Ø 2.主机端驱动
 - Ø 2.1 主机控制器驱动
 - Ø 2.2 设备驱动
- Ø 3.设备端驱动
 - Ø 3.1 设备控制器驱动
 - Ø 3.2 gadget驱动
- Ø 4. USB OTG

Ø 拓扑结构



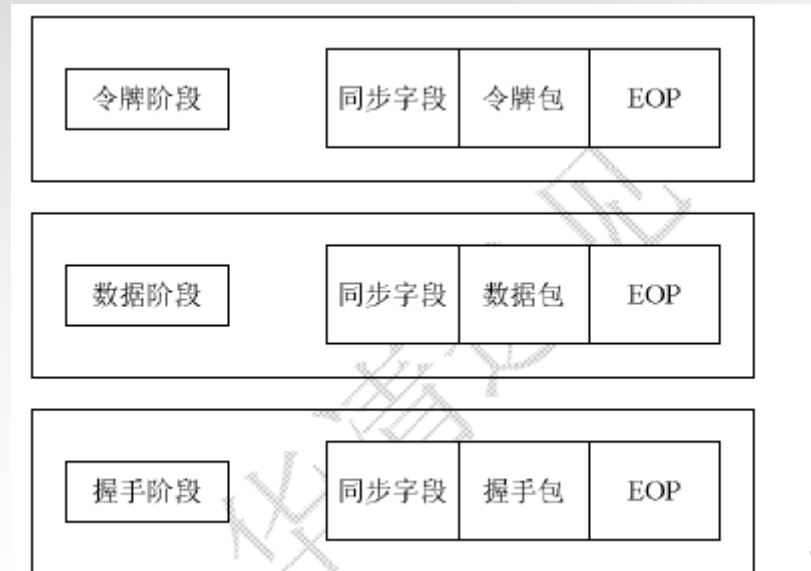
Ø 设备、配置、接口和端点



Ø 传输方式

- Ø 控制 (Control) 传输方式
- Ø 同步 (Synchronization) 传输方式
- Ø 中断 (Interrupt) 传输方式
- Ø 批量 (Bulk) 传输方式

Ø 事务处理



USB协议(3)

Ø包格式

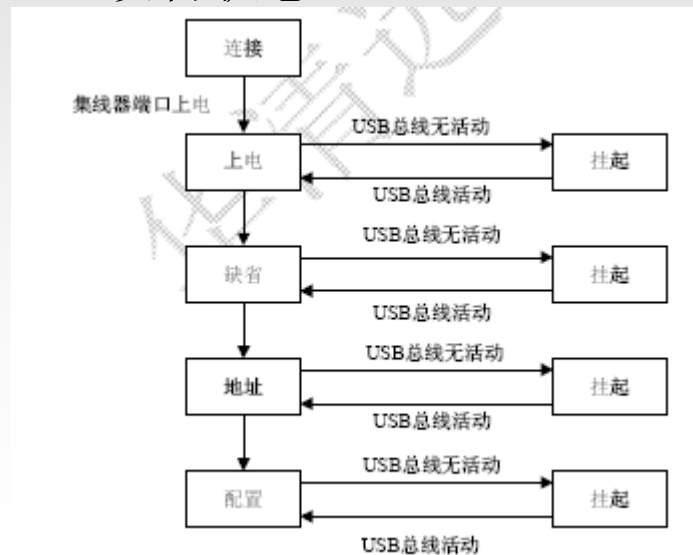
IN令牌包字段	PID	ADDR	ENDP	CRC
位数	8	7	4	5

SOF令牌包字段	PID	帧号字段	CRC
位数	8	11	5

数据包字段	PID	数据字段	CRC
位数	8	0~1024*8	16

握手包字段	PID
位数	8

Ø设备状态



直观的例子—U盘

Ø lsusb

```

bDeviceSubClass  0
bDeviceProtocol  0
bMaxPacketSize0 64
idVendor         0x0781 SanDisk Corp.
idProduct        0x5151
bcdDevice        0.10
iManufacturer    1 SanDisk Corporation
iProduct         2 Cruzer Micro
iSerial          3 20060877500A1BE1FDE1
bNumConfigurations 1
Configuration Descriptor:
  bLength          9
  bDescriptorType  2
  wTotalLength     32
  bNumInterfaces  1
  bConfigurationValue 1
  iConfiguration  0
  bmAttributes     0x80
  MaxPower        200mA
    
```

Interface Descriptor:

```

bLength          9
bDescriptorType  4
bInterfaceNumber 0
bAlternateSetting 0
bNumEndpoints   2
bInterfaceClass  8 Mass Storage
bInterfaceSubClass 6 SCSI
bInterfaceProtocol 80 Bulk (Zip)
iInterface       0
    
```

Endpoint Descriptor:

```

bLength          7
bDescriptorType  5
bEndpointAddress 0x81 EP 1 IN
bmAttributes     2
  Transfer Type   Bulk
  Synch Type     none
wMaxPacketSize   512
bInterval        0
    
```

Endpoint Descriptor:

```

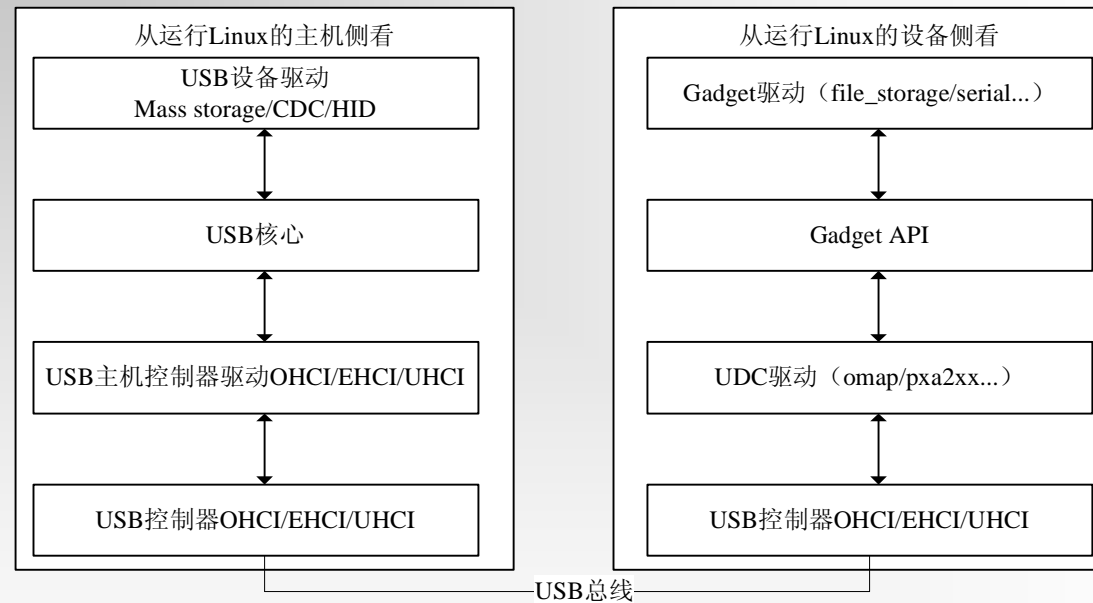
bLength          7
bDescriptorType  5
bEndpointAddress 0x01 EP 1 OUT
bmAttributes     2
  Transfer Type   Bulk
  Synch Type     none
wMaxPacketSize   512
bInterval        1
    
```

Language IDs: (length=4)

0409 English(US)

USB 驱动体系结构

- Ø 从运行Linux的主机侧
- Ø 从运行Linux的设备侧



USB 主机控制器驱动

数据结构:

Ø *struct usb_hcd*

Ø *struct hc_driver*

```
static const struct hc_driver xxx_hc_driver = {
    .description =          xxx_hcd_name,
    .product_desc =        "xxx OTG Controller",
    .hcd_priv_size =       sizeof(xxx_hcd_t),
    .irq =                  xxx_hcd_irq,
    .flags =                HCD_MEMORY | HCD_USB2,
    .start =                xxx_hcd_start,
    .stop =                 xxx_hcd_stop,
    .urb_enqueue =          xxx_hcd_urb_enqueue,
    .urb_dequeue =          xxx_hcd_urb_dequeue,
    .endpoint_disable =     xxx_hcd_endpoint_disable,
    .get_frame_number =     xxx_hcd_get_frame_number,
    .hub_status_data =      xxx_hcd_hub_status_data,
    .hub_control =          xxx_hcd_hub_control,
};
```

MS1

API:

```
struct usb_hcd *usb_create_hcd (const struct hc_driver *driver, struct device *dev, char *bus_name);
```

MS1

Starts processing a USB transfer request specified by a USB Request Block (URB). mem_flags indicates the type of memory allocation to use while processing this URB.

MC SYSTEM, 2008-12-12

USB设备驱动体系结构



USB设备驱动数据结构和API

Ø struct usb_driver

```
static struct usb_driver skel_driver =
{
    .name = "skeleton",
    .probe = skel_probe,
    .disconnect = skel_disconnect,
    .id_table = skel_table,
};
```

Ø struct usb_device_id

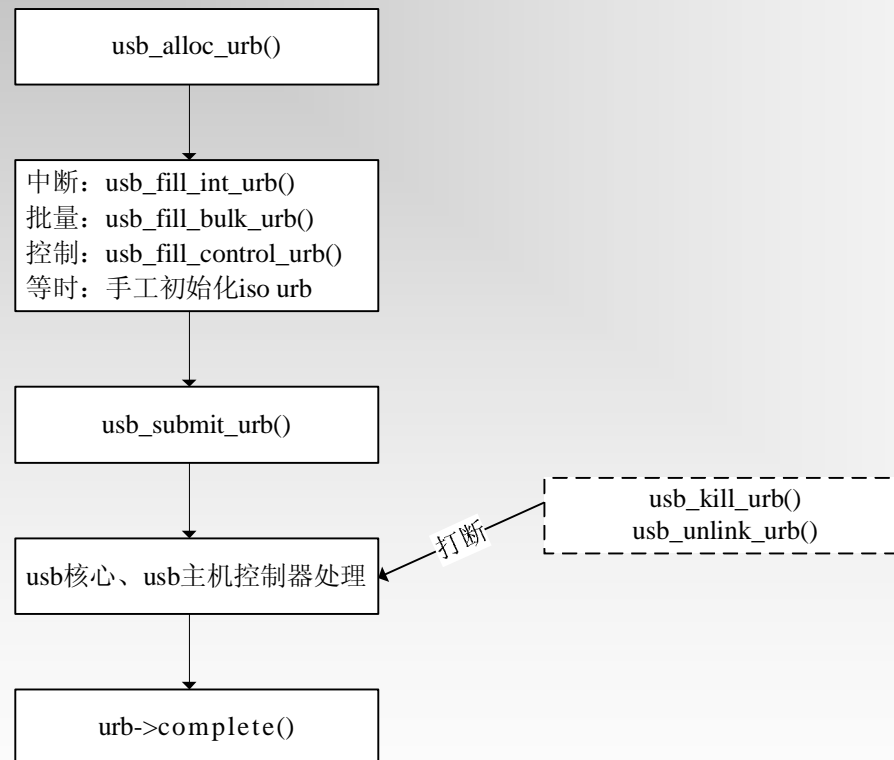
```
static struct usb_device_id skel_table [] = {
    { USB_DEVICE(USB_SKEL_VENDOR_ID,
        USB_SKEL_PRODUCT_ID) },
    {}
};
MODULE_DEVICE_TABLE(usb, skel_table);
```

Ø USB字符设备

```
int usb_register_dev(struct usb_interface *intf,
                    struct usb_class_driver *class_driver);
void usb_deregister_dev(struct usb_interface *intf,
                       struct usb_class_driver *class_driver);
struct usb_class_driver
{
    char *name; /*sysfs中用来描述设备名*/
    struct file_operations *fops; /*文件操作结构体指针*/
    int minor_base; /*开始次设备号*/
};
```

USB设备驱动--URB

ØUSB请求块 (USB request block, urb)





华清远见

USB设备驱动实例分析

- ØUSB骨架程序
- ØUSB串口驱动
- ØUSB键盘驱动

USB设备控制器驱动

数据结构

Ø struct usb_ep_ops

```
static struct usb_ep_ops xxx_ep_ops = {
    .enable      = xxx_ep_enable,
    .disable     = xxx_ep_disable,
    .alloc_request = xxx_alloc_request,
    .free_request = xxx_free_request,
    .alloc_buffer = xxx_alloc_buffer,
    .free_buffer  = xxx_free_buffer,
    .queue       = xxx_ep_queue,
    .dequeue     = xxx_ep_dequeue,
    .set_halt    = xxx_ep_set_halt,
};
```

Ø struct xxx_udc

```
struct xxx_udc
{
    struct usb_gadget gadget;
    struct usb_gadget_driver *driver;
    ...
};
```

Ø struct usb_gadget_ops

```
static const struct usb_gadget_ops xxx_udc_ops =
{
    .get_frame      = xxx_udc_get_frame,
    .wakeup         = xxx_udc_wakeup,
};
```

USB Gadget 驱动--组成(1)

```
Ø struct usb_gadget_driver
static struct usb_gadget_driver zero_driver = {
#ifdef CONFIG_USB_GADGET_DUALSPEED
    .speed      = USB_SPEED_HIGH,
#else
    .speed      = USB_SPEED_FULL,
#endif
    .function   = (char *) longname,
    .bind       = zero_bind,
    .unbind     = __exit_p(zero_unbind),

    .setup      = zero_setup,
    .disconnect = zero_disconnect,

    .suspend    = zero_suspend,
    .resume     = zero_resume,

    .driver     = {
        .name     = (char *) shortname,
        .owner    = THIS_MODULE,
    },
};
```

USB Gadget 驱动--组成(2)

Ø 设备、配置、接口、端点描述符

```
static struct usb_device_descriptor
device_desc = {
    ...
};
```

```
static struct usb_config_descriptor
source_sink_config = {
    ...
};
```

```
static struct usb_config_descriptor
source_sink_config = {
    ...
};
```

```
static struct usb_endpoint_descriptor {
fs_source_desc = {
    ...
};
```

```
static struct usb_endpoint_descriptor
fs_sink_desc = {
    ...,
};
....
```

```
static const struct usb_descriptor_header *fs_source_sink_function [] = {
    (struct usb_descriptor_header *) &otg_descriptor,
    (struct usb_descriptor_header *) &source_sink_intf,
    (struct usb_descriptor_header *) &fs_sink_desc,
    (struct usb_descriptor_header *) &fs_source_desc,
    NULL,
};
```

```
int usb_gadget_config_buf(
    const struct usb_config_descriptor *config,
    void *buf,
    unsigned length,
    const struct usb_descriptor_header **desc
)
{
    ...
    /* then interface/endpoint/class/vendor/... */
    len = usb_descriptor_fillbuf(USB_DT_CONFIG_SIZE + (u8*)buf,
        length - USB_DT_CONFIG_SIZE, desc);
    ...
}
```

USB Gadget 驱动--组成(3)

Ø setup

```

static int
zero_setup (struct usb_gadget *gadget, const struct usb_ctrlrequest *ctrl)
{
    struct zero_dev    *dev = get_gadget_data (gadget);
    struct usb_request *req = dev->req;

    ...

    switch (ctrl->bRequest) {
    case USB_REQ_GET_DESCRIPTOR:
    ...

    case USB_REQ_SET_CONFIGURATION:
    ...

    case USB_REQ_GET_CONFIGURATION:
    ...

    case USB_REQ_SET_INTERFACE:
    ...

    case USB_REQ_GET_INTERFACE:
    ...

    /* respond with data transfer before status phase? */
    if (value >= 0) {
        req->length = value;
        req->zero = value < w_length;
        value = usb_ep_queue (gadget->ep0, req, GFP_ATOMIC);
    }
    if (value < 0) {
        DBG (dev, "ep_queue --> %d\n", value);
        req->status = 0;
        zero_setup_complete (gadget->ep0, req);
    }
    }
}
/* device either stalls (value < 0) or reports success */
return value;
}

```

USB Gadget 驱动--usb_request

Ø *struct usb_request* MS6

```
struct usb_request {
    void          *buf;
    unsigned      length;
    dma_addr_t    dma;
    unsigned      no_interrupt:1;
    unsigned      zero:1;
    unsigned      short_not_ok:1;
    void          (*complete)(struct usb_ep *ep,
                              struct usb_request *req);
    void          *context;
    struct list_head list;
    int           status;
    unsigned      actual;
};
```

Ø **API**

```
static inline struct usb_request * usb_ep_alloc_request (struct usb_ep *ep, gfp_t gfp_flags); MS5
static inline void usb_ep_free_request (struct usb_ep *ep, struct usb_request *req); MS4
static inline int usb_ep_queue (struct usb_ep *ep, struct usb_request *req, gfp_t gfp_flags); MS3
static inline int usb_ep_dequeue (struct usb_ep *ep, struct usb_request *req); MS2
```

```
MS2 338/**
339 * usb_ep_dequeue - dequeues (cancels, unlinks) an I/O request from an endpoint
340 * @ep: the endpoint associated with the request
341 * @req: the request being canceled
342 *
343 * if the request is still active on the endpoint, it is dequeued and its
344 * completion routine is called (with status -ECONNRESET); else a negative
345 * error code is returned.
346 *
347 * note that some hardware can't clear out write fifos (to unlink the request
348 * at the head of the queue) except as part of disconnecting from usb. such
349 * restrictions prevent drivers from supporting configuration changes,
350 * even to configuration zero (a "chapter 9" requirement).
351 */
MC SYSTEM, 2008-12-13

MS3 278/**
279 * usb_ep_queue - queues (submits) an I/O request to an endpoint.
280 * @ep: the endpoint associated with the request
281 * @req: the request being submitted
282 * @gfp_flags: GFP_* flags to use in case the lower level driver couldn't
283 * pre-allocate all necessary memory with the request.
284 *
285 * This tells the device controller to perform the specified request through
286 * that endpoint (reading or writing a buffer). When the request completes,
287 * including being canceled by usb_ep_dequeue(), the request's completion
288 * routine is called to return the request to the driver. Any endpoint
289 * (except control endpoints like ep0) may have more than one transfer
290 * request queued; they complete in FIFO order. Once a gadget driver
291 * submits a request, that request may not be examined or modified until it
292 * is given back to that driver through the completion callback.
293 *
294 * Each request is turned into one or more packets. The controller driver
295 * never merges adjacent requests into the same packet. OUT transfers
296 * will sometimes use data that's already buffered in the hardware.
297 * Drivers can rely on the fact that the first byte of the request's buffer
298 * always corresponds to the first byte of some USB packet, for both
299 * IN and OUT transfers.
300 *
301 * Bulk endpoints can queue any amount of data; the transfer is packetized
302 * automatically. The last packet will be short if the request doesn't fill it
303 * out completely. Zero length packets (ZLPs) should be avoided in portable
304 * protocols since not all usb hardware can successfully handle zero length
305 * packets. (ZLPs may be explicitly written, and may be implicitly written if
```

```

306 * the request 'zero' flag is set.) Bulk endpoints may also be used
307 * for interrupt transfers; but the reverse is not true, and some endpoints
308 * won't support every interrupt transfer. (Such as 768 byte packets.)
309 *
310 * Interrupt-only endpoints are less functional than bulk endpoints, for
311 * example by not supporting queueing or not handling buffers that are
312 * larger than the endpoint's maxpacket size. They may also treat data
313 * toggle differently.
314 *
315 * Control endpoints ... after getting a setup() callback, the driver queues
316 * one response (even if it would be zero length). That enables the
317 * status ack, after transferring data as specified in the response. Setup
318 * functions may return negative error codes to generate protocol stalls.
319 * (Note that some USB device controllers disallow protocol stall responses
320 * in some cases.) When control responses are deferred (the response is
321 * written after the setup callback returns), then usb_ep_set_halt() may be
322 * used on ep0 to trigger protocol stalls.
323 *
324 * For periodic endpoints, like interrupt or isochronous ones, the usb host
325 * arranges to poll once per interval, and the gadget driver usually will
326 * have queued some data to transfer at that time.
327 *
328 * Returns zero, or a negative error code. Endpoints that are not enabled
329 * report errors; errors will also be
330 * reported when the usb peripheral is disconnected.
331 */

```

MC SYSTEM, 2008-12-13

MS4

```

222/**
223 * usb_ep_free_request - frees a request object
224 * @ep: the endpoint associated with the request
225 * @req: the request being freed
226 *
227 * Reverses the effect of usb_ep_alloc_request().
228 * Caller guarantees the request is not queued, and that it will
229 * no longer be requeued (or otherwise used).
230 */

```

MC SYSTEM, 2008-12-13

MS5

```

202/**
203 * usb_ep_alloc_request - allocate a request object to use with this endpoint
204 * @ep: the endpoint to be used with with the request
205 * @gfp_flags: GFP_* flags to use
206 *
207 * Request objects must be allocated with this call, since they normally

```

```
208 * need controller-specific setup and may even need endpoint-specific
209 * resources such as allocation of DMA descriptors.
210 * Requests may be submitted with usb_ep_queue(), and receive a single
211 * completion callback. Free requests with usb_ep_free_request(), when
212 * they are no longer needed.
213 *
214 * Returns the request, or null if one could not be allocated.
215 */
```

MC SYSTEM, 2008-12-13

MS6

```
22/**
23 * struct usb_request - describes one i/o request
24 * @buf: Buffer used for data. Always provide this; some controllers
25 *      only use PIO, or don't use DMA for some endpoints.
26 * @dma: DMA address corresponding to 'buf'. If you don't set this
27 *      field, and the usb controller needs one, it is responsible
28 *      for mapping and unmapping the buffer.
29 * @length: Length of that data
30 * @no_interrupt: If true, hints that no completion irq is needed.
31 *               Helpful sometimes with deep request queues that are handled
32 *               directly by DMA controllers.
33 * @zero: If true, when writing data, makes the last packet be "short"
34 *        by adding a zero length packet as needed;
35 * @short_not_ok: When reading data, makes short packets be
36 *               treated as errors (queue stops advancing till cleanup).
37 * @complete: Function called when request completes, so this request and
38 *            its buffer may be re-used.
39 *            Reads terminate with a short packet, or when the buffer fills,
40 *            whichever comes first. When writes terminate, some data bytes
41 *            will usually still be in flight (often in a hardware fifo).
42 *            Errors (for reads or writes) stop the queue from advancing
43 *            until the completion function returns, so that any transfers
44 *            invalidated by the error may first be dequeued.
45 * @context: For use by the completion callback
46 * @list: For use by the gadget driver.
47 * @status: Reports completion code, zero or a negative errno.
48 *         Normally, faults block the transfer queue from advancing until
49 *         the completion callback returns.
50 *         Code "-ESHUTDOWN" indicates completion caused by device disconnect,
51 *         or when the driver disabled the endpoint.
52 * @actual: Reports bytes transferred to/from the buffer. For reads (OUT
53 *          transfers) this may be less than the requested length. If the
54 *          short_not_ok flag is set, short reads are treated as errors
55 *          even when status otherwise indicates successful completion.
```

```
56 *      Note that for writes (IN transfers) some data bytes may still
57 *      reside in a device-side FIFO when the request is reported as
58 *      complete.
59 *
60 *      These are allocated/freed through the endpoint they're used with. The
61 *      hardware's driver can add extra per-request data to the memory it returns,
62 *      which often avoids separate memory allocations (potential failures),
63 *      later when the request is queued.
64 *
65 *      Request flags affect request handling, such as whether a zero length
66 *      packet is written (the "zero" flag), whether a short read should be
67 *      treated as an error (blocking request queue advance, the "short_not_ok"
68 *      flag), or hinting that an interrupt is not required (the "no_interrupt"
69 *      flag, for use with deep request queues).
70 *
71 *      Bulk endpoints can use any size buffers, and can also be used for interrupt
72 *      transfers. interrupt-only endpoints can be much less functional.
73 */
74      // NOTE this is analagous to 'struct urb' on the host side,
75      // except that it's thinner and promotes more pre-allocation.
```



华清远见

USB Gadget驱动实例

- Ø zero gadget
- Ø 串口 gadget
- Ø file storage

USB OTG协议

Ø OTG补充规范对USB的扩展

更具节能性的电源管理;允许设备以主机和外设两种形式工作

Ø SRP协议:

OTG从机(B-device)请求A-device重新使能VBUS,而后 A-device使用HNP协议交换两个设备的工作方式,这两步完成后由新的OTG主机开始事务传输。

Ø HNP协议

1. A-device在完成对B-device的使用后, 可以通过查询B-device的 OTG性能描述符来判断是否支持HNP协议。如支持HNP, B-device将返回有效的OTG性能描述符, A-device 则产生一个set_feature命令(即HNP_Enable)来通知B-device可以在总线挂起的时候以主机方式工作, 随后A-device挂起总线。
2. B-device通过上拉电阻(全速时)或者下拉电阻(高速时)拉低D+以示连接断开。随后, 作为对B-device断开的响应, A-device使能它的数据线并开始以从机方式工作。完成这些转换后, B-device和A-device便各自以主机角色和外设角色使用总线。
3. 当B-device正常结束传输事务时便挂起VBUS使能其上拉电阻, 重新以从机方式运行。A-device 检测到总线挂起后, 发出一个连接断开信号并重新以主机方式工作。

USB OTG驱动(1)

Ø 设备控制器新接口

```

struct usb_gadget {
    ...
    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;
    ...
};

/* used by external USB transceiver */
int usb_gadget_vbus_connect(struct usb_gadget *gadget);
int usb_gadget_vbus_disconnect(struct usb_gadget *gadget);

/* call this during SET_CONFIGURATION */
int usb_gadget_vbus_draw(struct usb_gadget *gadget, unsigned mA);

/* these logically control the USB D+ pullup */
int usb_gadget_connect(struct usb_gadget *gadget);
int usb_gadget_disconnect(struct usb_gadget *gadget);

static inline int usb_gadget_wakeup (struct usb_gadget *gadget);

```

USB OTG驱动(2)

Ø Gadget驱动的修改

- 当`gadget->is_otg` 为真时，为每个配置提供一个OTG描述符
- 在`SET_CONFIGURATION`中通过“用户接口”（如`printk`、`LED`等）报告HNP可用
- 当`suspend`开始时，通过“用户接口”报告HNP角色切换（B设备变为B主机，A设备变为A主机）的开始

USB OTG驱动(3)

Ø 主机侧：USB core的更新

Ø OTG 枚举和目标外设列表

```
struct usb_bus {  
    ...  
    u8 otg_port;        /* 0, or index of OTG/HNP port */  
    unsigned is_b_host:1;    /* true during some HNP roleswitches */  
    unsigned b_hnp_enable:1; /* OTG: did A-Host enable HNP? */  
    ...  
};
```

Ø CONFIG_USB_SUSPEND

```
/* selective suspend/resume */  
extern int usb_suspend_device(struct usb_device *dev, u32 state);  
extern int usb_resume_device(struct usb_device *dev);
```

Ø 其他电源管理问题

USB OTG驱动(4)

ØOTG控制器

```
struct otg_transceiver {
    struct device    *dev;
    const char      *label;
    u8               default_a;
    enum usb_otg_state  state;
    struct usb_bus   *host;
    struct usb_gadget *gadget;
    /* to pass extra port status to the root hub */
    u16              port_status;
    u16              port_change;
    /* bind/unbind the host controller */
    int  (*set_host)(struct otg_transceiver *otg,
                    struct usb_bus *host);
    /* bind/unbind the peripheral controller */
    int  (*set_peripheral)(struct otg_transceiver *otg,
                          struct usb_gadget *gadget);
    /* effective for B devices, ignored for A-peripheral */
    int  (*set_power)(struct otg_transceiver *otg,
                    unsigned mA);
    /* for B devices only: start session with A-Host */
    int  (*start_srp)(struct otg_transceiver *otg);
    /* start or continue HNP role switch */
    int  (*start_hnp)(struct otg_transceiver *otg);
};
```

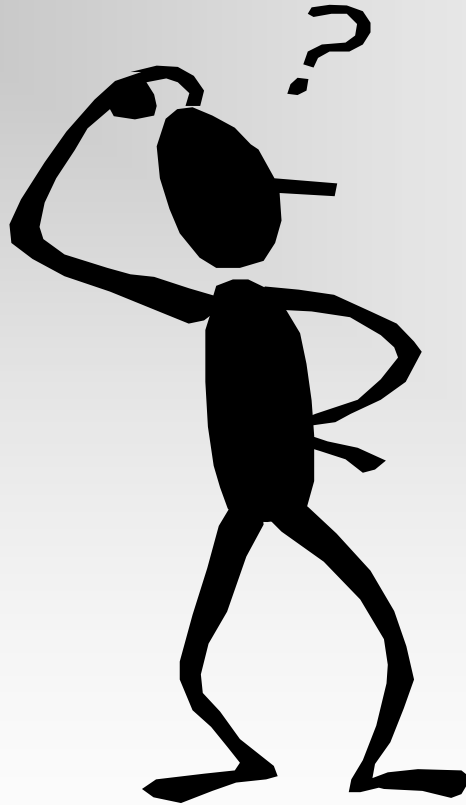
华清远见Linux驱动课程

- ✓ 嵌入式Linux驱动初级班
 - ✓ 通过本课程的学习，学员可以掌握Linux下字符设备、块设备、网络设备的驱动程序开发，同时掌握嵌入式Linux的系统开发和分析方法。
 - ✓ 嵌入式Linux驱动开发高级班
 - ✓ 本课程以案例教学为主，系统地介绍Linux下有关FrameBuffer、MMC卡、USB设备的驱动程序开发。
- ✓ 嵌入式Linux驱动开发教材



华清远见

让我们一起讨论！



FAR  SIGHT

The logo features the word "FARSIGHT" in white, bold, serif capital letters. A red, stylized vertical line separates the "FAR" and "SIGHT" parts. The text is centered within a dark green, textured, downward-pointing triangle that has a 3D effect with a lighter green top edge.

FARSIGHT

The success's road

www.farsight.com.cn

谢谢！