# Atog hcd & udc

## 注册atog\_hcd driver: "aotg\_hcd"

### aotg\_init()

module\_init(aotg\_init);

static int \_\_init aotg\_init(void)

{

**platform\_driver\_register**(&aotg\_hcd\_driver);

create\_acts\_hcd\_proc();

}

### struct platform\_driver aotg\_hcd\_driver //aotg hcd driver

struct platform\_driver aotg\_hcd\_driver = {

**.probe = aotg\_probe,**

.remove = aotg\_remove,

.shutdown = aotg\_hcd\_shutdown,

.driver = {

.owner = THIS\_MODULE,

.name = platform\_drv\_name, // "aotg\_hcd"

.of\_match\_table = **aotg\_of\_match**,

#ifdef CONFIG\_PM

.pm = DEV\_PM\_OPS,

#endif

},

};

struct of\_device\_id **aotg\_of\_match**[] = {

{.compatible = "actions,s700-usb2.0-0"}, //S700

{.compatible = "actions,s700-usb2.0-1"}, //S700

{.compatible = "actions,s900-usb2.0-0"}, //S900

{.compatible = "actions,s900-usb2.0-1"}, //S900

{},

};

## 注册atog\_hcd device

### actions,s700-usb2.0-0

usb2h0: usb@e01d0000 {

compatible = "actions,s700-usb2.0-0";

reg = <0 0xe01d0000 0 0x1000>;

interrupts = <GIC\_SPI 24 IRQ\_TYPE\_LEVEL\_HIGH>;

power-domains = <&powergate POWER\_DOMAIN\_USB2H0>;

clocks = <&clock CLK\_USB2H0\_PLLEN>, <&clock CLK\_USB2H0\_PHY>, <&clock CLK\_USB2H0\_CCE>;

clock-names = "usbh0\_pllen", "usbh0\_phy", "usbh0\_cce";

resets = <&reset RESET\_USBH0>;

reset-names = "usb2h0";

};

## 注册atog\_udc driver: "aotg\_udc"

### aotg\_init()

module\_init(aotg\_init);

static int \_\_init aotg\_init(void)

{

**platform\_driver\_register**(&aotg\_udc\_driver);

create\_acts\_udc\_proc();

}

### struct platform\_driver aotg\_udc\_driver //aotg udc driver

struct platform\_driver aotg\_udc\_driver = {

.driver = {

.name = "aotg\_udc",

.owner = THIS\_MODULE,

.of\_match\_table = **aotg\_of\_match**,

},

**.probe = aotg\_probe,**

.remove = aotg\_remove,

};

struct of\_device\_id **aotg\_of\_match**[] = {

{.compatible = "actions,s700-usb2.0-0"}, //S700

{.compatible = "actions,s700-usb2.0-1"}, //S700

{.compatible = "actions,s900-usb2.0-0"}, //S900

{.compatible = "actions,s900-usb2.0-1"}, //S900

{},

};

## 注册atog\_udc device

### actions,s700-usb2.0-1

usb2h1: usb@e01d8000 {

compatible = "actions,s700-usb2.0-1";

reg = <0 0xe01d8000 0 0x1000>;

interrupts = <GIC\_SPI 61 IRQ\_TYPE\_LEVEL\_HIGH>;

power-domains = <&powergate POWER\_DOMAIN\_USB2H1>;

clocks = <&clock CLK\_USB2H1\_PLLEN>, <&clock CLK\_USB2H1\_PHY>, <&clock CLK\_USB2H1\_CCE>;

clock-names = "usbh1\_pllen", "usbh1\_phy", "usbh1\_cce";

resets = <&reset RESET\_USBH1>;

reset-names = "usb2h1";

};

## probe

### aotg\_probe() //初始化aotg\_plat\_data

struct aotg\_plat\_data aotg\_data[2];

int aotg\_probe(struct platform\_device \*pdev)

{

**aotg\_hcd\_get\_dts(pdev)**; **//pdev->id= [0: "actions,s700-usb2.0-0", 1:"actions,s700-usb2.0-1"]**

struct resource \*res\_mem = platform\_get\_resource(pdev, IORESOURCE\_MEM, 0);

request\_mem\_region(res\_mem->start, res\_mem->end - res\_mem->start + 1, dev\_name(&pdev->dev));

aotg\_data[pdev->id].base = devm\_ioremap(&pdev->dev, res\_mem->start, res\_mem->end - res\_mem->start + 1);

aotg\_plat\_data\_fill(&pdev->dev, pdev->id); //aotg\_data[pdev->id].usbecs = [S700: 0xE024c094, 0xE0228094]

aotg\_data[pdev->id].rsrc\_start = res\_mem->start;

aotg\_data[pdev->id].rsrc\_len = res\_mem->end - res\_mem->start + 1;

pdev->dev.dma\_mask = &aotg\_dmamask;

pdev->dev.coherent\_dma\_mask = DMA\_BIT\_MASK(32);

aotg\_data[pdev->id].**irq = platform\_get\_irq(pdev, 0);**

aotg\_data[pdev->id].dev = &pdev->dev;

device\_init\_wakeup(&pdev->dev, true);

if (pdev->id) aotg\_data[pdev->id].clk\_usbh\_pllen = devm\_clk\_get(&pdev->dev, "usbh1\_pllen");

else aotg\_data[pdev->id].clk\_usbh\_pllen = devm\_clk\_get(&pdev->dev, "usbh0\_pllen");

if (pdev->id) aotg\_data[pdev->id].clk\_usbh\_phy = devm\_clk\_get(&pdev->dev, "usbh1\_phy");

else aotg\_data[pdev->id].clk\_usbh\_phy = devm\_clk\_get(&pdev->dev, "usbh0\_phy");

if (ic\_type[pdev->id] == S900) {

if (pdev->id) aotg\_data[pdev->id].clk\_usbh\_cce = devm\_clk\_get(&pdev->dev, "usbh1\_cce");

else aotg\_data[pdev->id].clk\_usbh\_cce = devm\_clk\_get(&pdev->dev, "usbh0\_cce");

}

}

### aotg\_hcd\_get\_dts()

static int aotg\_hcd\_get\_dts(struct platform\_device \*pdev)

{

struct device\_node \*of\_node = pdev->dev.of\_node;

if (of\_device\_is\_compatible(of\_node, aotg\_of\_match[0].compatible)) { // "actions,s700-usb2.0-0"

pdev->id = 0; ic\_type[0] = S700;

} else if (of\_device\_is\_compatible(of\_node, aotg\_of\_match[1].compatible)) { // "actions,s700-usb2.0-1"

pdev->id = 1; ic\_type[1] = S700;

} else if (of\_device\_is\_compatible(of\_node, aotg\_of\_match[2].compatible)) {

pdev->id = 0; ic\_type[0] = S900;

} else if (of\_device\_is\_compatible(of\_node, aotg\_of\_match[3].compatible)) {

pdev->id = 1; ic\_type[1] = S900;

}

//udc enable

if (!of\_find\_property(of\_node, **"aotg\_udc\_enable"**, NULL)) pr\_info("usb2-%d can't find aotg\_udc\_enable config\n", pdev->id);

else **aotg\_udc\_enable[pdev->id]** = !!(be32\_to\_cpup((const \_\_be32 \*)of\_get\_property(of\_node, "aotg\_udc\_enable", NULL)));

...

return 0;

}

## 注册usb\_gadget.dev和usb\_udc.dev

struct aotg\_udc \* aotg\_udc

aotg\_udc **->dev = aotg\_data[id].dev; //”actions,s700-usb2.0-1”**

struct usb\_gadget \*gadget

dev\_set\_name(&gadget->dev, "gadget");

gadget->dev.parent = aotg\_udc->dev;

**struct usb\_udc \*udc**

udc->dev.parent = aotg\_udc->dev

udc->gadget = aotg\_udc->gadget

### aotg\_init()

module\_init(aotg\_init);

static int \_\_init aotg\_init(void)

{

**aotg\_udc\_add**();

start\_mon\_wq = create\_singlethread\_workqueue("aotg\_start\_mon\_wq");

INIT\_DELAYED\_WORK(&start\_mon\_wker, **start\_mon**);

queue\_delayed\_work(start\_mon\_wq, &start\_mon\_wker, msecs\_to\_jiffies(10000));

}

### aotg\_udc\_add()

enum aotg\_mode\_e aotg\_mode[2];

void aotg\_udc\_add(void)

{

int id;

if (owl\_get\_boot\_mode()) return;

i**f (aotg\_udc\_enable[0])** { **id = 0;** } //aotg\_udc\_enable[pdev->id]由aotg\_hcd\_get\_dts()决定

**else if (aotg\_udc\_enable[1])** { **id = 1;** }

else { pr\_info("No aotg\_udc being enabled!\n"); return; }

aotg\_mode[id] = UDC\_MODE;

**aotg\_udc\_init(id);**

}

### aotg\_udc\_init() //初始化aotg\_udc, usb\_gadget, aotg\_ep, usb\_ep

void aotg\_udc\_init(int id)

{

struct aotg\_udc \*udc = &memory; // aotg\_udc初始化

udc->port\_specific = &aotg\_data[id]; // struct aotg\_plat\_data, 由aotg\_probe()初始化

udc->irq = aotg\_data[id].irq;

**udc->dev = aotg\_data[id].dev; //指向”actions,s700-usb2.0-1”设备**

udc->base = aotg\_data[id].base;

udc->id = id; // aotg\_udc\_enable[pdev->id]=1, 由dts的"aotg\_udc\_enable"决定

acts\_udc\_controller = udc;

**udc\_reinit(udc);**

**aotg\_hardware\_init(id);**

request\_irq(udc->irq, **aotg\_udc\_irq**, 0, udc\_driver\_name, udc);

**usb\_add\_gadget\_udc**(udc->dev, &udc->gadget);

}

#### struct aotg\_udc memory //初始化aotg\_udc: default值

**struct aotg\_udc** **memory** = {

.lock = \_\_SPIN\_LOCK\_UNLOCKED(memory.lock),

.gadget = { // struct usb\_gadget gadget;

.ops = &aotg\_udc\_ops,

.ep0 = &memory.ep[0].ep,

**.max\_speed = USB\_SPEED\_HIGH,**

.speed = USB\_SPEED\_UNKNOWN,

.name = udc\_driver\_name, // "aotg\_udc"

.dev = {

.init\_name = "gadget",

},

},

/\*control endpoint \*/

.ep[0] = { // struct aotg\_ep ep

.ep = { // struct usb\_ep ep;

.name = ep0name, // "ep0"

.ops = &aotg\_ep\_ops,

.maxpacket = EP0\_PACKET\_SIZE, //64

},

.dev = &memory, //struct aotg\_udc

.maxpacket = EP0\_PACKET\_SIZE, //64

},

/\*bulk out endpoint \*/

.ep[1] = {

.ep = {

.ops = &aotg\_ep\_ops,

},

.dev = &memory,

.bEndpointAddress = 1,

.mask = 1,

.reg\_udccs = OUT1CS, // 0x0000000B

.reg\_udccon = OUT1CON, // 0x0000000A

.reg\_udcbc = OUT1BCL, // 0x00000008

.reg\_udcfifo = 0,

.reg\_maxckl = HCIN1MAXPCKL, // 0x000001E2

.reg\_fifostaddr = OUT1STARTADDRESS, // 0x00000304

.reg\_dmalinkaddr = HCIN1DMALINKADDR, // 0x00000810

.reg\_curaddr = HCIN1DMACURADDR, // 0x00000814

.reg\_dmactrl = HCIN1DMACTRL, // 0x00000818

.reg\_dmacomplete\_cnt = HCIN1DMACOMPLETECNT, // 0x0000081C

},

/\*bulk in endpoint \*/

.ep[2] = {

.ep = {

.ops = &aotg\_ep\_ops,

},

.dev = &memory,

.bEndpointAddress = USB\_DIR\_IN | 1,

.mask = USB\_UDC\_IN\_MASK | 1,

.reg\_udccs = IN1CS,

.reg\_udccon = IN1CON,

.reg\_udcbc = IN1BCL,

.reg\_udcfifo = 0,

.reg\_maxckl = HCOUT1MAXPCKL,

.reg\_fifostaddr = IN1STARTADDRESS,

.reg\_dmalinkaddr = HCOUT1DMALINKADDR,

.reg\_curaddr = HCOUT1DMACURADDR,

.reg\_dmactrl = HCOUT1DMACTRL,

.reg\_dmacomplete\_cnt = HCOUT1DMACOMPLETECNT,

},

/\*bulk out endpoint \*/

.ep[3] = {

.ep = {

.ops = &aotg\_ep\_ops,

},

.dev = &memory,

.bEndpointAddress = 2,

.mask = 2,

.reg\_udccs = OUT2CS,

.reg\_udccon = OUT2CON,

.reg\_udcbc = OUT2BCL,

.reg\_udcfifo = 0,

.reg\_maxckl = HCIN2MAXPCKL,

.reg\_fifostaddr = OUT2STARTADDRESS,

.reg\_dmalinkaddr = HCIN2DMALINKADDR,

.reg\_curaddr = HCIN2DMACURADDR,

.reg\_dmactrl = HCIN2DMACTRL,

.reg\_dmacomplete\_cnt = HCIN2DMACOMPLETECNT,

},

/\*bulk in endpoint \*/

.ep[4] = {

.ep = {

.ops = &aotg\_ep\_ops,

},

.dev = &memory,

.bEndpointAddress = USB\_DIR\_IN | 2,

.mask = USB\_UDC\_IN\_MASK | 2,

.reg\_udccs = IN2CS,

.reg\_udccon = IN2CON,

.reg\_udcbc = IN2BCL,

.reg\_udcfifo = 0,

.reg\_maxckl = HCOUT2MAXPCKL,

.reg\_fifostaddr = IN2STARTADDRESS,

.reg\_dmalinkaddr = HCOUT2DMALINKADDR,

.reg\_curaddr = HCOUT2DMACURADDR,

.reg\_dmactrl = HCOUT2DMACTRL,

.reg\_dmacomplete\_cnt = HCOUT2DMACOMPLETECNT,

},

/\*bulk out endpoint \*/

.ep[5] = {

.ep = {

.ops = &aotg\_ep\_ops,

},

.dev = &memory,

.bEndpointAddress = 3,

.mask = 3,

.reg\_udccs = OUT3CS,

.reg\_udccon = OUT3CON,

.reg\_udcbc = OUT3BCL,

.reg\_udcfifo = 0,

.reg\_maxckl = HCIN3MAXPCKL,

.reg\_fifostaddr = OUT3STADDR,

.reg\_dmalinkaddr = HCIN3DMALINKADDR,

.reg\_curaddr = HCIN3DMACURADDR,

.reg\_dmactrl = HCIN3DMACTRL,

.reg\_dmacomplete\_cnt = HCIN3DMACOMPLETECNT,

},

/\* interupt in \*/

.ep[6] = {

.ep = {

.ops = &aotg\_ep\_ops,

},

.dev = &memory,

.bEndpointAddress = USB\_DIR\_IN | 3,

.mask = USB\_UDC\_IN\_MASK | 3,

.reg\_udccs = IN3CS,

.reg\_udccon = IN3CON,

.reg\_udcbc = IN3BCL,

.reg\_udcfifo = 0,

.reg\_maxckl = HCOUT3MAXPCKL,

.reg\_fifostaddr = IN3STADDR,

.reg\_dmalinkaddr = HCOUT3DMALINKADDR,

.reg\_curaddr = HCOUT3DMACURADDR,

.reg\_dmactrl = HCOUT3DMACTRL,

.reg\_dmacomplete\_cnt = HCOUT3DMACOMPLETECNT,

},

/\*iso in endpoint \*/

.ep[7] = {

.ep = {

.ops = &aotg\_ep\_ops,

},

.dev = &memory,

.bEndpointAddress = USB\_DIR\_IN | 4,

.mask = USB\_UDC\_IN\_MASK | 4,

.reg\_udccs = IN4CS,

.reg\_udccon = IN4CON,

.reg\_udcbc = IN4BCL,

.reg\_udcfifo = 0,

.reg\_maxckl = HCOUT4MAXPCKL,

.reg\_fifostaddr = IN4STADDR,

.reg\_dmalinkaddr = HCOUT4DMALINKADDR,

.reg\_curaddr = HCOUT4DMACURADDR,

.reg\_dmactrl = HCOUT4DMACTRL,

.reg\_dmacomplete\_cnt = HCOUT4DMACOMPLETECNT,

},

/\*bulk out endpoint \*/

.ep[8] = {

.ep = {

.ops = &aotg\_ep\_ops,

},

.dev = &memory,

.bEndpointAddress = 4,

.mask = 4,

.reg\_udccs = OUT4CS,

.reg\_udccon = OUT4CON,

.reg\_udcbc = OUT4BCL,

.reg\_udcfifo = 0,

.reg\_maxckl = HCIN4MAXPCKL,

.reg\_fifostaddr = OUT4STADDR,

.reg\_dmalinkaddr = HCIN4DMALINKADDR,

.reg\_curaddr = HCIN4DMACURADDR,

.reg\_dmactrl = HCIN4DMACTRL,

.reg\_dmacomplete\_cnt = HCIN4DMACOMPLETECNT,

},

/\*bulk in endpoint \*/

.ep[9] = {

.ep = {

.ops = &aotg\_ep\_ops,

},

.dev = &memory,

.bEndpointAddress = USB\_DIR\_IN | 5,

.mask = USB\_UDC\_IN\_MASK | 5,

.reg\_udccs = IN5CS,

.reg\_udccon = IN5CON,

.reg\_udcbc = IN5BCL,

.reg\_udcfifo = 0,

.reg\_maxckl = HCOUT5MAXPCKL,

.reg\_fifostaddr = IN5STADDR,

.reg\_dmalinkaddr = HCOUT5DMALINKADDR,

.reg\_curaddr = HCOUT5DMACURADDR,

.reg\_dmactrl = HCOUT5DMACTRL,

.reg\_dmacomplete\_cnt = HCOUT5DMACOMPLETECNT,

},

/\*bulk out endpoint \*/

.ep[10] = {

.ep = {

.ops = &aotg\_ep\_ops,

},

.dev = &memory,

.bEndpointAddress = 5,

.mask = 5,

.reg\_udccs = OUT5CS,

.reg\_udccon = OUT5CON,

.reg\_udcbc = OUT5BCL,

.reg\_udcfifo = 0,

.reg\_maxckl = HCIN5MAXPCKL,

.reg\_fifostaddr = OUT5STADDR,

.reg\_dmalinkaddr = HCIN5DMALINKADDR,

.reg\_curaddr = HCIN5DMACURADDR,

.reg\_dmactrl = HCIN5DMACTRL,

.reg\_dmacomplete\_cnt = HCIN5DMACOMPLETECNT,

},

};

### struct usb\_gadget\_ops aotg\_udc\_ops

static const struct usb\_gadget\_ops aotg\_udc\_ops = {

.get\_frame = aotg\_udc\_get\_frame,

.wakeup = aotg\_udc\_wakeup,

.pullup = aotg\_udc\_pullup,

.vbus\_session = aotg\_udc\_vbus\_session,

.vbus\_draw = aotg\_udc\_vbus\_draw,

.udc\_start = aotg\_udc\_start,

.udc\_stop = aotg\_udc\_stop,

};

### struct usb\_ep\_ops aotg\_ep\_ops

static struct usb\_ep\_ops aotg\_ep\_ops = {

.enable = aotg\_ep\_enable,

.disable = aotg\_ep\_disable,

.alloc\_request = aotg\_ep\_alloc\_request,

.free\_request = aotg\_ep\_free\_request,

.queue = aotg\_ep\_queue,

.dequeue = aotg\_ep\_dequeue,

.set\_halt = aotg\_ep\_set\_halt,

.fifo\_flush = aotg\_ep\_fifo\_flush, /\*not sure \*/

};

### udc\_reinit() //将aotg\_ep添加到gadget->ep\_list链表中

void udc\_reinit(struct aotg\_udc \*dev)

{

dev->ep0state = EP0\_WAIT\_FOR\_SETUP;

dev->gadget.speed = USB\_SPEED\_UNKNOWN;

dev->state = UDC\_UNKNOWN;

memset(&dev->stats, 0, sizeof(struct udc\_stats));

for (i = 0; i < AOTG\_UDC\_NUM\_ENDPOINTS; i++) {

struct aotg\_ep \*ep = &dev->ep[i];

if (i != 0) **list\_add\_tail(&ep->ep.ep\_list, &dev->gadget.ep\_list);**

ep->ep.desc = NULL;

ep->dev = dev;

ep->stopped = 0;

INIT\_LIST\_HEAD(&ep->queue);

ep->udc\_irqs = 0;

}

}

### usb\_add\_gadget\_udc() //初始化usb\_udc，注册"gadget"，注册”aotg\_udc”

//usb\_add\_gadget\_udc(udc->dev, &udc->gadget); // udc->dev指向**”actions,s700-usb2.0-1”设备**

int usb\_add\_gadget\_udc(struct device \*parent, struct usb\_gadget \*gadget)

{

return usb\_add\_gadget\_udc\_release(parent, gadget, NULL);

}

int usb\_add\_gadget\_udc\_release(struct device \*parent, struct usb\_gadget \*gadget, void (\*release)(struct device \*dev))

{

dev\_set\_name(&gadget->dev, "gadget");

INIT\_WORK(&gadget->work, usb\_gadget\_state\_work);

gadget->dev.parent = parent;

dma\_set\_coherent\_mask(&gadget->dev, parent->coherent\_dma\_mask);

gadget->dev.dma\_parms = parent->dma\_parms;

gadget->dev.dma\_mask = parent->dma\_mask;

gadget->dev.release = usb\_udc\_nop\_release;

ret = **device\_register(&gadget->dev);**

**struct usb\_udc \*udc = kzalloc(sizeof(\*udc), GFP\_KERNEL);**

device\_initialize(&udc->dev);

udc->dev.release = usb\_udc\_release;

udc->dev.class = udc\_class;

udc->dev.groups = usb\_udc\_attr\_groups;

udc->dev.parent = parent;

dev\_set\_name(&udc->dev, "%s", kobject\_name(&parent->kobj));

**list\_add\_tail(&udc->list, &udc\_list);**

udc->gadget = gadget;

ret = **device\_add(&udc->dev);**

usb\_gadget\_set\_state(gadget, USB\_STATE\_NOTATTACHED);

}

int **device\_register**(struct device \*dev)

{

device\_initialize(dev);

return **device\_add**(dev);

}

# Dw3

## module\_platform\_driver()

module\_platform\_driver(dwc3\_driver);

static struct platform\_driver dwc3\_driver = {

.probe = dwc3\_probe,

.remove = dwc3\_remove,

.driver = {

**.name = "dwc3",**

.of\_match\_table = of\_match\_ptr(of\_dwc3\_match),

.pm = DWC3\_PM\_OPS,

},

};

static const struct of\_device\_id of\_dwc3\_match[] = {

{ **.compatible = "synopsys,dwc3"}**,

{ },

};

## DTS: dwc3@e0400000

owl\_dwc3@e040ce04 {

compatible = "actions,s700-dwc3";

reg = <0 0xe040ce04 0 0xc>;

ranges;

power-domains = <&powergate POWER\_DOMAIN\_USB3>;

clocks = <&clock CLK\_USB3\_480MPLL0>, <&clock CLK\_USB3\_480MPHY0>, <&clock CLK\_USB3\_5GPHY>, <&clock CLK\_USB3\_CCE>, <&clock CLK\_USB3\_MAC>;

clock-names = "usb3\_480mpll0", "usb3\_480mphy0", "usb3\_5gphy", "usb3\_cce", "usb3\_mac";

resets = <&reset RESET\_USB3>;

reset-names = "usb3";

dwc3@e0400000 {

**compatible = "synopsys,dwc3";**

reg = <0 0xe0400000 0 0xcd00>;

interrupts = <0 23 4>;

usb-phy = <&usb2\_phy>, <&usb3\_phy>;

};

};

## dwc3\_probe() //初始化dwc3

static int dwc3\_probe(struct platform\_device \*pdev)

{

void \*mem = devm\_kzalloc(dev, sizeof(\*dwc) + DWC3\_ALIGN\_MASK, GFP\_KERNEL); //#define DWC3\_ALIGN\_MASK (16 - 1)

**struct dwc3** \*dwc = PTR\_ALIGN(mem, DWC3\_ALIGN\_MASK + 1);

dwc->mem = mem;

res = platform\_get\_resource(pdev, IORESOURCE\_IRQ, 0);

dwc->xhci\_resources[1].start = res->start;

dwc->xhci\_resources[1].end = res->end;

dwc->xhci\_resources[1].flags = res->flags;

dwc->xhci\_resources[1].name = res->name;

res = platform\_get\_resource(pdev, IORESOURCE\_MEM, 0);

dwc->xhci\_resources[0].start = res->start;

dwc->xhci\_resources[0].end = dwc->xhci\_resources[0].start + DWC3\_XHCI\_REGS\_END; //#define DWC3\_XHCI\_REGS\_END 0x7fff

dwc->xhci\_resources[0].flags = res->flags;

dwc->xhci\_resources[0].name = res->name;

dwc->usb2\_phy = devm\_usb\_get\_phy\_by\_phandle(dev, "usb-phy", 0);

dwc->usb3\_phy = devm\_usb\_get\_phy\_by\_phandle(dev, "usb-phy", 1);

usb\_phy\_set\_suspend(dwc->usb2\_phy, 0);

usb\_phy\_set\_suspend(dwc->usb3\_phy, 0);

spin\_lock\_init(&dwc->lock);

platform\_set\_drvdata(pdev, dwc);

res = devm\_request\_mem\_region(dev, res->start + DWC3\_GLOBALS\_REGS\_START,

resource\_size(res) - DWC3\_GLOBALS\_REGS\_START, dev\_name(dev));

regs = devm\_ioremap\_nocache(dev, res->start, resource\_size(res));

dwc->regs = regs;

dwc->regs\_size = resource\_size(res);

dwc->dev = dev;

dev->dma\_mask = dev->parent->dma\_mask;

dev->dma\_parms = dev->parent->dma\_parms;

dma\_set\_coherent\_mask(dev, dev->parent->coherent\_dma\_mask);

if (!strncmp("super", maximum\_speed, 5)) dwc->maximum\_speed = DWC3\_DCFG\_SUPERSPEED;

else if (!strncmp("high", maximum\_speed, 4)) dwc->maximum\_speed = DWC3\_DCFG\_HIGHSPEED;

else if (!strncmp("full", maximum\_speed, 4)) dwc->maximum\_speed = DWC3\_DCFG\_FULLSPEED1;

else if (!strncmp("low", maximum\_speed, 3)) dwc->maximum\_speed = DWC3\_DCFG\_LOWSPEED;

else dwc->maximum\_speed = DWC3\_DCFG\_SUPERSPEED;

dwc->needs\_fifo\_resize = of\_property\_read\_bool(node, "tx-fifo-resize");

pm\_runtime\_enable(dev);

pm\_runtime\_get\_sync(dev);

pm\_runtime\_forbid(dev);

dwc3\_cache\_hwparams(dwc);

ret = dwc3\_alloc\_event\_buffers(dwc, DWC3\_EVENT\_BUFFERS\_SIZE);

ret = **dwc3\_core\_init(dwc);**

ret = dwc3\_event\_buffers\_setup(dwc);

#if IS\_ENABLED(CONFIG\_USB\_DWC3\_DUAL\_ROLE) //CONFIG\_USB\_DWC3\_DUAL\_ROLE=y

mode = DWC3\_MODE\_DEVICE;

#endif

switch (mode) {

case DWC3\_MODE\_DEVICE: **dwc3\_set\_mode**(dwc, DWC3\_GCTL\_PRTCAP\_**DEVICE**); **dwc3\_gadget\_init(dwc);** break;

case DWC3\_MODE\_HOST: dwc3\_set\_mode(dwc, DWC3\_GCTL\_PRTCAP\_HOST); dwc3\_host\_init(dwc); break;

case DWC3\_MODE\_DRD: dwc3\_set\_mode(dwc, DWC3\_GCTL\_PRTCAP\_OTG); dwc3\_host\_init(dwc); dwc3\_gadget\_init(dwc); break;

}

dwc->mode = mode;

ret = dwc3\_debugfs\_init(dwc);

pm\_runtime\_allow(dev);

#if SUPPORT\_NOT\_RMMOD\_USBDRV

dwc3\_plug\_init(dwc);

#endif

#ifdef MASK\_SUPPER\_SPEED\_FOR\_ADFUS

dwc3\_set\_dwc\_for\_adfus(dwc);

#endif

}

## dwc3\_gadget\_init() //初始化usb\_gadget

int dwc3\_gadget\_init(struct dwc3 \*dwc)

{

dwc->ctrl\_req = dma\_alloc\_coherent(dwc->dev, sizeof(\*dwc->ctrl\_req),&dwc->ctrl\_req\_addr, GFP\_KERNEL);

dwc->ep0\_trb = dma\_alloc\_coherent(dwc->dev, sizeof(\*dwc->ep0\_trb),&dwc->ep0\_trb\_addr, GFP\_KERNEL);

dwc->setup\_buf = kzalloc(DWC3\_EP0\_BOUNCE\_SIZE, GFP\_KERNEL); //#define DWC3\_EP0\_BOUNCE\_SIZE 512

dwc->ep0\_bounce = dma\_alloc\_coherent(dwc->dev,DWC3\_EP0\_BOUNCE\_SIZE, &dwc->ep0\_bounce\_addr,GFP\_KERNEL);

dwc->gadget.ops = &dwc3\_gadget\_ops;

dwc->gadget.max\_speed = USB\_SPEED\_SUPER;

dwc->gadget.speed = USB\_SPEED\_UNKNOWN;

dwc->gadget.sg\_supported = true;

dwc->gadget.name = "dwc3-gadget";

ret = **dwc3\_gadget\_init\_endpoints**(dwc);

reg = dwc3\_readl(dwc->regs, DWC3\_DCFG);

reg |= DWC3\_DCFG\_LPM\_CAP;

dwc3\_writel(dwc->regs, DWC3\_DCFG, reg);

ret = **usb\_add\_gadget\_udc(dwc->dev, &dwc->gadget);**

#ifdef USB\_CHARGE\_DETECT

usb\_charge\_detect\_init();

#endif

}

### struct usb\_gadget\_ops dwc3\_gadget\_ops

static const struct usb\_gadget\_ops dwc3\_gadget\_ops = {

.get\_frame = dwc3\_gadget\_get\_frame,

.wakeup = dwc3\_gadget\_wakeup,

.set\_selfpowered = dwc3\_gadget\_set\_selfpowered,

.pullup = dwc3\_gadget\_pullup,

.udc\_start = dwc3\_gadget\_start,

.udc\_stop = dwc3\_gadget\_stop,

};

## dwc3\_gadget\_init\_endpoints()

static int dwc3\_gadget\_init\_endpoints(struct dwc3 \*dwc)

{

INIT\_LIST\_HEAD(&dwc->gadget.ep\_list);

ret = dwc3\_gadget\_init\_hw\_endpoints(dwc, dwc->num\_out\_eps, 0);

ret = dwc3\_gadget\_init\_hw\_endpoints(dwc, dwc->num\_in\_eps, 1);

}

## dwc3\_gadget\_init\_hw\_endpoints() //初始化dwc3\_ep, usb\_ep

static int dwc3\_gadget\_init\_hw\_endpoints(struct dwc3 \*dwc, u8 num, u32 direction)

{

**struct dwc3\_ep** \*dep;

for (i = 0; i < num; i++) {

u8 epnum = (i << 1) | (!!direction);

dep = kzalloc(sizeof(\*dep), GFP\_KERNEL);

dep->dwc = dwc;

dep->number = epnum;

dwc->eps[epnum] = dep;

snprintf(dep->name, sizeof(dep->name), "ep%d%s", epnum >> 1,(epnum & 1) ? "in" : "out");

dep->endpoint.name = dep->name;

dep->direction = (epnum & 1);

if (epnum == 0 || epnum == 1) {

dep->endpoint.maxpacket = 512;

dep->endpoint.maxburst = 1;

dep->endpoint.ops = &dwc3\_gadget\_ep0\_ops;

if (!epnum) dwc->gadget.ep0 = &dep->endpoint;

} else {

dep->endpoint.maxpacket = 1024;

dep->endpoint.max\_streams = 15;

dep->endpoint.ops = &dwc3\_gadget\_ep\_ops;

list\_add\_tail(&dep->endpoint.ep\_list,&dwc->gadget.ep\_list);

ret = dwc3\_alloc\_trb\_pool(dep);

}

INIT\_LIST\_HEAD(&dep->request\_list);

INIT\_LIST\_HEAD(&dep->req\_queued);

}

}

### struct usb\_ep\_ops dwc3\_gadget\_ep0\_ops

static const struct usb\_ep\_ops dwc3\_gadget\_ep0\_ops = {

.enable = dwc3\_gadget\_ep0\_enable,

.disable = dwc3\_gadget\_ep0\_disable,

.alloc\_request = dwc3\_gadget\_ep\_alloc\_request,

.free\_request = dwc3\_gadget\_ep\_free\_request,

.queue = dwc3\_gadget\_ep0\_queue,

.dequeue = dwc3\_gadget\_ep\_dequeue,

.set\_halt = dwc3\_gadget\_ep0\_set\_halt,

.set\_wedge = dwc3\_gadget\_ep\_set\_wedge,

};

### struct usb\_ep\_ops dwc3\_gadget\_ep\_ops

static const struct usb\_ep\_ops dwc3\_gadget\_ep\_ops = {

.enable = dwc3\_gadget\_ep\_enable,

.disable = dwc3\_gadget\_ep\_disable,

.alloc\_request = dwc3\_gadget\_ep\_alloc\_request,

.free\_request = dwc3\_gadget\_ep\_free\_request,

.queue = dwc3\_gadget\_ep\_queue,

.dequeue = dwc3\_gadget\_ep\_dequeue,

.set\_halt = dwc3\_gadget\_ep\_set\_halt,

.set\_wedge = dwc3\_gadget\_ep\_set\_wedge,

};

## usb\_add\_gadget\_udc() //初始化usb\_udc，注册usb\_gadget.dev，注册usb\_udc.dev="synopsys,dwc3"

//usb\_add\_gadget\_udc(udc->dev, &udc->gadget); // udc->dev指向**"synopsys,dwc3"设备**

int usb\_add\_gadget\_udc(struct device \*parent, struct usb\_gadget \*gadget)

{

return usb\_add\_gadget\_udc\_release(parent, gadget, NULL);

}

int usb\_add\_gadget\_udc\_release(struct device \*parent, struct usb\_gadget \*gadget, void (\*release)(struct device \*dev))

{

dev\_set\_name(&gadget->dev, "gadget");

INIT\_WORK(&gadget->work, usb\_gadget\_state\_work);

gadget->dev.parent = parent;

dma\_set\_coherent\_mask(&gadget->dev, parent->coherent\_dma\_mask);

gadget->dev.dma\_parms = parent->dma\_parms;

gadget->dev.dma\_mask = parent->dma\_mask;

gadget->dev.release = usb\_udc\_nop\_release;

ret = **device\_register(&gadget->dev);**

**struct usb\_udc \*udc = kzalloc(sizeof(\*udc), GFP\_KERNEL);**

device\_initialize(&udc->dev);

udc->dev.release = usb\_udc\_release;

udc->dev.class = udc\_class;

udc->dev.groups = usb\_udc\_attr\_groups;

udc->dev.parent = parent;

dev\_set\_name(&udc->dev, "%s", kobject\_name(&parent->kobj));

**list\_add\_tail(&udc->list, &udc\_list);**

udc->gadget = gadget;

ret = **device\_add(&udc->dev);**

usb\_gadget\_set\_state(gadget, USB\_STATE\_NOTATTACHED);

}

int **device\_register**(struct device \*dev)

{

device\_initialize(dev);

return **device\_add**(dev);

}

# webcam udc driver: g\_webcam

## webcam\_init() //初始化usb\_composite\_driver："g\_webcam"

module\_init(webcam\_init);

static int \_\_init webcam\_init(void)

{

return **usb\_composite\_probe**(&webcam\_driver);

}

### struct usb\_composite\_driver webcam\_driver

static \_\_refdata **struct usb\_composite\_driver** webcam\_driver = {

.name = "g\_webcam",

.dev = &webcam\_device\_descriptor,

.strings = webcam\_device\_strings,

.max\_speed = USB\_SPEED\_SUPER,

.bind = webcam\_bind,

.unbind = webcam\_unbind,

};

struct usb\_composite\_driver {

const char \*name; // "g\_webcam"

…

struct usb\_gadget\_driver gadget\_driver; //由usb\_composite\_probe()初始化

};

### struct usb\_device\_descriptor webcam\_device\_descriptor

#define WEBCAM\_VENDOR\_ID 0x1d6b /\* Linux Foundation \*/

#define WEBCAM\_PRODUCT\_ID 0x0102 /\* Webcam A/V gadget \*/

#define WEBCAM\_DEVICE\_BCD 0x0010 /\* 0.10 \*/

static struct usb\_device\_descriptor webcam\_device\_descriptor = {

.bLength = USB\_DT\_DEVICE\_SIZE, //18

.bDescriptorType = USB\_DT\_DEVICE, //0x01

.bcdUSB = cpu\_to\_le16(0x0200),

.bDeviceClass = USB\_CLASS\_MISC, //0xef

.bDeviceSubClass = 0x02,

.bDeviceProtocol = 0x01,

.bMaxPacketSize0 = 0, /\* dynamic \*/

.idVendor = cpu\_to\_le16(WEBCAM\_VENDOR\_ID),

.idProduct = cpu\_to\_le16(WEBCAM\_PRODUCT\_ID),

.bcdDevice = cpu\_to\_le16(WEBCAM\_DEVICE\_BCD),

.iManufacturer = 0, //"Linux Foundation"

.iProduct = 0, //"Webcam gadget"

.iSerialNumber = 0, /\* dynamic \*/

.bNumConfigurations = 0, /\* dynamic \*/

};

### struct usb\_gadget\_strings \*webcam\_device\_strings

static struct usb\_gadget\_strings \*webcam\_device\_strings[] = {

&webcam\_stringtab,

NULL,

};

static struct usb\_gadget\_strings webcam\_stringtab = {

.language = 0x0409, /\* en-us \*/

.strings = webcam\_strings,

};

static struct usb\_string webcam\_strings[] = {

[USB\_GADGET\_MANUFACTURER\_IDX].s = webcam\_vendor\_label,

[USB\_GADGET\_PRODUCT\_IDX].s = webcam\_product\_label,

[USB\_GADGET\_SERIAL\_IDX].s = "",

[STRING\_DESCRIPTION\_IDX].s = webcam\_config\_label,

{ }

};

static char webcam\_vendor\_label[] = "Linux Foundation";

static char webcam\_product\_label[] = "Webcam gadget";

static char webcam\_config\_label[] = "Video";

## usb\_composite\_probe() //初始化usb\_gadget\_driver

struct usb\_composite\_driver driver;

driver->name = "g\_webcam";

driver->gadget\_driver = composite\_driver\_template;

struct usb\_gadget\_driver \*gadget\_driver;

gadget\_driver->function = (char \*) driver->name; //"g\_webcam"

int **usb\_composite\_probe**(struct usb\_composite\_driver \*driver)

{

struct usb\_gadget\_driver \*gadget\_driver;

driver->gadget\_driver = composite\_driver\_template;

gadget\_driver = &driver->gadget\_driver;

gadget\_driver->function = (char \*) driver->name; //"g\_webcam"

gadget\_driver->driver.name = driver->name; //"g\_webcam"

gadget\_driver->max\_speed = driver->max\_speed;

return **usb\_gadget\_probe\_driver**(gadget\_driver);

}

### struct usb\_gadget\_driver composite\_driver\_template

static const **struct usb\_gadget\_driver** composite\_driver\_template = {

**.bind = composite\_bind,**

.unbind = composite\_unbind,

.setup = composite\_setup,

.disconnect = composite\_disconnect,

.suspend = composite\_suspend,

.resume = composite\_resume,

.driver = {.owner = THIS\_MODULE,},

};

struct usb\_gadget\_driver {

char \*function; //"g\_webcam"

…

struct device\_driver driver; // driver.name = "g\_webcam"

};

## usb\_gadget\_probe\_driver() //寻找usb\_udc：“atog\_udc”

int **usb\_gadget\_probe\_drive**r(struct usb\_gadget\_driver \*driver)

{

struct usb\_udc \*udc = NULL;

list\_for\_each\_entry(udc, &**udc\_list**, list) // 参见usb\_add\_gadget\_udc()，指向usb\_udc

if (!udc->driver) goto found;

return -ENODEV;

found:

return **udc\_bind\_to\_driver**(udc, driver);

}

## udc\_bind\_to\_driver() //添加g\_webcam driver到usb\_udc.driver中

static int **udc\_bind\_to\_driver**(struct usb\_udc \*udc, struct usb\_gadget\_driver \*driver)

{

**udc->driver = driver;**

**udc->dev.driver = &driver->driver;**

**udc->gadget->dev.driver = &driver->driver;**

ret = **driver->bind**(udc->gadget, driver); // **composite\_bind()**

ret = **usb\_gadget\_udc\_start(**udc->gadget, driver);

**kobject\_uevent**(&udc->dev.kobj, **KOBJ\_CHANGE**);

return 0;

}

## composite\_bind() //初始化usb\_composite\_dev

// **driver->bind**(udc->gadget, driver);

static int **composite\_bind**(struct usb\_gadget \*gadget,struct usb\_gadget\_driver \*gdriver)

{

struct usb\_composite\_driver \*composite = to\_cdriver(gdriver);

**struct usb\_composite\_dev \*cdev = kzalloc(sizeof \*cdev, GFP\_KERNEL);**

spin\_lock\_init(&cdev->lock);

cdev->gadget = gadget;

set\_gadget\_data(gadget, cdev);

INIT\_LIST\_HEAD(&cdev->configs);

INIT\_LIST\_HEAD(&cdev->gstrings);

status = **composite\_dev\_prepare**(composite, cdev); //继续初始化usb\_composite\_dev

status = **composite->bind**(cdev); // **webcam\_bind()**

**update\_unchanged\_dev\_desc**(&cdev->desc, composite->dev);

}

int **composite\_dev\_prepare**(struct usb\_composite\_driver \*composite,struct usb\_composite\_dev \*cdev) //继续初始化usb\_composite\_dev

{

struct usb\_gadget \*gadget = cdev->gadget;

cdev->req = usb\_ep\_alloc\_request(gadget->ep0, GFP\_KERNEL); //ep->ops->alloc\_request(ep, gfp\_flags); //aotg\_ep\_alloc\_request()

cdev->req->buf = kmalloc(USB\_COMP\_EP0\_BUFSIZ, GFP\_KERNEL);

ret = device\_create\_file(&gadget->dev, &dev\_attr\_suspended);

cdev->req->complete = composite\_setup\_complete;

gadget->ep0->driver\_data = cdev;

cdev->driver = composite; //usb\_composite\_driver结构体

if (CONFIG\_USB\_GADGET\_VBUS\_DRAW <= USB\_SELF\_POWER\_VBUS\_MAX\_DRAW)

usb\_gadget\_set\_selfpowered(gadget);

usb\_ep\_autoconfig\_reset(gadget);

}

## webcam\_bind() //初始化usb\_configuration: "Video"

static int \_\_init **webcam\_bind**(struct usb\_composite\_dev \*cdev)

{

ret = usb\_string\_ids\_tab(cdev, webcam\_strings);

webcam\_device\_descriptor.iManufacturer =webcam\_strings[USB\_GADGET\_MANUFACTURER\_IDX].id;

webcam\_device\_descriptor.iProduct =webcam\_strings[USB\_GADGET\_PRODUCT\_IDX].id;

webcam\_config\_driver.iConfiguration =webcam\_strings[STRING\_DESCRIPTION\_IDX].id;

**usb\_add\_config**(cdev, &webcam\_config\_driver,**webcam\_config\_bind**); //初始化usb\_configuration: "Video"

usb\_composite\_overwrite\_options(cdev, &coverwrite);

}

### struct usb\_configuration webcam\_config\_driver

static **struct usb\_configuration** webcam\_config\_driver = {

**.label = webcam\_config\_label, // "Video"**

.bConfigurationValue = 1,

.iConfiguration = 0, /\* dynamic \*/

.bmAttributes = USB\_CONFIG\_ATT\_SELFPOWER,

.MaxPower = CONFIG\_USB\_GADGET\_VBUS\_DRAW,

};

## webcam\_config\_bind()

static int \_\_init **webcam\_config\_bind**(struct usb\_configuration \*c)

{

return **uvc\_bind\_config**(c,

uvc\_fs\_control\_cls,

uvc\_ss\_control\_cls,

uvc\_fs\_streaming\_cls,

uvc\_hs\_streaming\_cls,

uvc\_ss\_streaming\_cls);

}

### Control descriptor

#### uvc\_fs\_control\_cls // uvc\_descriptor\_header : full speed control(usb 2.0)

static const struct uvc\_descriptor\_header \* const uvc\_fs\_control\_cls[] = {

(const struct uvc\_descriptor\_header \*) &uvc\_control\_header,

(const struct uvc\_descriptor\_header \*) &uvc\_camera\_terminal,

(const struct uvc\_descriptor\_header \*) &uvc\_processing,

(const struct uvc\_descriptor\_header \*) &uvc\_output\_terminal,

NULL,

};

#### uvc\_ss\_control\_cls // uvc\_descriptor\_header : super speed control(usb 3.0)

static const struct uvc\_descriptor\_header \* const uvc\_ss\_control\_cls[] = {

(const struct uvc\_descriptor\_header \*) &uvc\_control\_header,

(const struct uvc\_descriptor\_header \*) &uvc\_camera\_terminal,

(const struct uvc\_descriptor\_header \*) &uvc\_processing,

(const struct uvc\_descriptor\_header \*) &uvc\_output\_terminal,

NULL,

};

#### uvc\_control\_header

static const struct UVC\_HEADER\_DESCRIPTOR(1) uvc\_control\_header = {

.bLength = UVC\_DT\_HEADER\_SIZE(1), .bDescriptorType = USB\_DT\_CS\_INTERFACE,

.bDescriptorSubType = UVC\_VC\_HEADER,

.bcdUVC = cpu\_to\_le16(0x0100),

.wTotalLength = 0, /\* dynamic \*/

.dwClockFrequency = cpu\_to\_le32(48000000),

.bInCollection = 0, /\* dynamic \*/

.baInterfaceNr[0] = 0, /\* dynamic \*/

};

#### uvc\_camera\_terminal

static const struct uvc\_camera\_terminal\_descriptor uvc\_camera\_terminal = {

.bLength = UVC\_DT\_CAMERA\_TERMINAL\_SIZE(3), .bDescriptorType = USB\_DT\_CS\_INTERFACE,

.bDescriptorSubType = UVC\_VC\_INPUT\_TERMINAL,

.bTerminalID = 1,

.wTerminalType = cpu\_to\_le16(0x0201),

.bAssocTerminal = 0,

.iTerminal = 0,

.wObjectiveFocalLengthMin = cpu\_to\_le16(0),

.wObjectiveFocalLengthMax = cpu\_to\_le16(0),

.wOcularFocalLength = cpu\_to\_le16(0),

.bControlSize = 3,

.bmControls[0] = 2,

.bmControls[1] = 0,

.bmControls[2] = 0,

};

#### uvc\_processing

static const struct uvc\_processing\_unit\_descriptor uvc\_processing = {

.bLength = UVC\_DT\_PROCESSING\_UNIT\_SIZE(2), .bDescriptorType = USB\_DT\_CS\_INTERFACE,

.bDescriptorSubType = UVC\_VC\_PROCESSING\_UNIT,

.bUnitID = 2,

.bSourceID = 1,

.wMaxMultiplier = cpu\_to\_le16(16\*1024),

.bControlSize = 2,

.bmControls[0] = 1,

.bmControls[1] = 0,

.iProcessing = 0,

};

#### uvc\_output\_terminal

static const struct uvc\_output\_terminal\_descriptor uvc\_output\_terminal = {

.bLength = UVC\_DT\_OUTPUT\_TERMINAL\_SIZE, .bDescriptorType = USB\_DT\_CS\_INTERFACE,

.bDescriptorSubType = UVC\_VC\_OUTPUT\_TERMINAL,

.bTerminalID = 3,

.wTerminalType = cpu\_to\_le16(0x0101),

.bAssocTerminal = 0,

.bSourceID = 2,

.iTerminal = 0,

};

### Streaming descriptor

static const struct uvc\_descriptor\_header \* const uvc\_fs\_streaming\_cls[] = {

(const struct uvc\_descriptor\_header \*) &uvc\_input\_header,

(const struct uvc\_descriptor\_header \*) &uvc\_format\_yuv,

(const struct uvc\_descriptor\_header \*) &uvc\_frame\_yuv\_360p,

(const struct uvc\_descriptor\_header \*) &uvc\_frame\_yuv\_720p,

(const struct uvc\_descriptor\_header \*) &uvc\_format\_mjpg,

(const struct uvc\_descriptor\_header \*) &uvc\_frame\_mjpg\_360p,

(const struct uvc\_descriptor\_header \*) &uvc\_frame\_mjpg\_720p,

(const struct uvc\_descriptor\_header \*) &uvc\_color\_matching,

NULL,

};

static const struct uvc\_descriptor\_header \* const uvc\_hs\_streaming\_cls[] = {

(const struct uvc\_descriptor\_header \*) &uvc\_input\_header,

(const struct uvc\_descriptor\_header \*) &uvc\_format\_yuv,

(const struct uvc\_descriptor\_header \*) &uvc\_frame\_yuv\_360p,

(const struct uvc\_descriptor\_header \*) &uvc\_frame\_yuv\_720p,

(const struct uvc\_descriptor\_header \*) &uvc\_format\_mjpg,

(const struct uvc\_descriptor\_header \*) &uvc\_frame\_mjpg\_360p,

(const struct uvc\_descriptor\_header \*) &uvc\_frame\_mjpg\_720p,

(const struct uvc\_descriptor\_header \*) &uvc\_color\_matching,

NULL,

};

static const struct uvc\_descriptor\_header \* const uvc\_ss\_streaming\_cls[] = {

(const struct uvc\_descriptor\_header \*) &uvc\_input\_header,

(const struct uvc\_descriptor\_header \*) &uvc\_format\_yuv,

(const struct uvc\_descriptor\_header \*) &uvc\_frame\_yuv\_360p,

(const struct uvc\_descriptor\_header \*) &uvc\_frame\_yuv\_720p,

(const struct uvc\_descriptor\_header \*) &uvc\_format\_mjpg,

(const struct uvc\_descriptor\_header \*) &uvc\_frame\_mjpg\_360p,

(const struct uvc\_descriptor\_header \*) &uvc\_frame\_mjpg\_720p,

(const struct uvc\_descriptor\_header \*) &uvc\_color\_matching,

NULL,

};

static const struct UVC\_INPUT\_HEADER\_DESCRIPTOR(1, 2) uvc\_input\_header = {

.bLength = UVC\_DT\_INPUT\_HEADER\_SIZE(1, 2),

.bDescriptorType = USB\_DT\_CS\_INTERFACE,

.bDescriptorSubType = UVC\_VS\_INPUT\_HEADER,

.bNumFormats = 2,

.wTotalLength = 0, /\* dynamic \*/

.bEndpointAddress = 0, /\* dynamic \*/

.bmInfo = 0,

.bTerminalLink = 3,

.bStillCaptureMethod = 0,

.bTriggerSupport = 0,

.bTriggerUsage = 0,

.bControlSize = 1,

.bmaControls[0][0] = 0,

.bmaControls[1][0] = 4,

};

static const struct uvc\_format\_uncompressed uvc\_format\_yuv = {

.bLength = UVC\_DT\_FORMAT\_UNCOMPRESSED\_SIZE,

.bDescriptorType = USB\_DT\_CS\_INTERFACE,

.bDescriptorSubType = UVC\_VS\_FORMAT\_UNCOMPRESSED,

.bFormatIndex = 1,

.bNumFrameDescriptors = 2,

.guidFormat =

{ 'Y', 'U', 'Y', '2', 0x00, 0x00, 0x10, 0x00,

0x80, 0x00, 0x00, 0xaa, 0x00, 0x38, 0x9b, 0x71},

.bBitsPerPixel = 16,

.bDefaultFrameIndex = 1,

.bAspectRatioX = 0,

.bAspectRatioY = 0,

.bmInterfaceFlags = 0,

.bCopyProtect = 0,

};

static const struct UVC\_FRAME\_UNCOMPRESSED(3) uvc\_frame\_yuv\_360p = {

.bLength = UVC\_DT\_FRAME\_UNCOMPRESSED\_SIZE(3),

.bDescriptorType = USB\_DT\_CS\_INTERFACE,

.bDescriptorSubType = UVC\_VS\_FRAME\_UNCOMPRESSED,

.bFrameIndex = 1,

.bmCapabilities = 0,

.wWidth = cpu\_to\_le16(640),

.wHeight = cpu\_to\_le16(360),

.dwMinBitRate = cpu\_to\_le32(18432000),

.dwMaxBitRate = cpu\_to\_le32(55296000),

.dwMaxVideoFrameBufferSize = cpu\_to\_le32(460800),

.dwDefaultFrameInterval = cpu\_to\_le32(666666),

.bFrameIntervalType = 3,

.dwFrameInterval[0] = cpu\_to\_le32(666666),

.dwFrameInterval[1] = cpu\_to\_le32(1000000),

.dwFrameInterval[2] = cpu\_to\_le32(5000000),

};

static const struct UVC\_FRAME\_UNCOMPRESSED(1) uvc\_frame\_yuv\_720p = {

.bLength = UVC\_DT\_FRAME\_UNCOMPRESSED\_SIZE(1),

.bDescriptorType = USB\_DT\_CS\_INTERFACE,

.bDescriptorSubType = UVC\_VS\_FRAME\_UNCOMPRESSED,

.bFrameIndex = 2,

.bmCapabilities = 0,

.wWidth = cpu\_to\_le16(1280),

.wHeight = cpu\_to\_le16(720),

.dwMinBitRate = cpu\_to\_le32(29491200),

.dwMaxBitRate = cpu\_to\_le32(29491200),

.dwMaxVideoFrameBufferSize = cpu\_to\_le32(1843200),

.dwDefaultFrameInterval = cpu\_to\_le32(5000000),

.bFrameIntervalType = 1,

.dwFrameInterval[0] = cpu\_to\_le32(5000000),

};

static const struct uvc\_format\_mjpeg uvc\_format\_mjpg = {

.bLength = UVC\_DT\_FORMAT\_MJPEG\_SIZE,

.bDescriptorType = USB\_DT\_CS\_INTERFACE,

.bDescriptorSubType = UVC\_VS\_FORMAT\_MJPEG,

.bFormatIndex = 2,

.bNumFrameDescriptors = 2,

.bmFlags = 0,

.bDefaultFrameIndex = 1,

.bAspectRatioX = 0,

.bAspectRatioY = 0,

.bmInterfaceFlags = 0,

.bCopyProtect = 0,

};

static const struct UVC\_FRAME\_MJPEG(3) uvc\_frame\_mjpg\_360p = {

.bLength = UVC\_DT\_FRAME\_MJPEG\_SIZE(3),

.bDescriptorType = USB\_DT\_CS\_INTERFACE,

.bDescriptorSubType = UVC\_VS\_FRAME\_MJPEG,

.bFrameIndex = 1,

.bmCapabilities = 0,

.wWidth = cpu\_to\_le16(640),

.wHeight = cpu\_to\_le16(360),

.dwMinBitRate = cpu\_to\_le32(18432000),

.dwMaxBitRate = cpu\_to\_le32(55296000),

.dwMaxVideoFrameBufferSize = cpu\_to\_le32(460800),

.dwDefaultFrameInterval = cpu\_to\_le32(666666),

.bFrameIntervalType = 3,

.dwFrameInterval[0] = cpu\_to\_le32(666666),

.dwFrameInterval[1] = cpu\_to\_le32(1000000),

.dwFrameInterval[2] = cpu\_to\_le32(5000000),

};

static const struct UVC\_FRAME\_MJPEG(1) uvc\_frame\_mjpg\_720p = {

.bLength = UVC\_DT\_FRAME\_MJPEG\_SIZE(1),

.bDescriptorType = USB\_DT\_CS\_INTERFACE,

.bDescriptorSubType = UVC\_VS\_FRAME\_MJPEG,

.bFrameIndex = 2,

.bmCapabilities = 0,

.wWidth = cpu\_to\_le16(1280),

.wHeight = cpu\_to\_le16(720),

.dwMinBitRate = cpu\_to\_le32(29491200),

.dwMaxBitRate = cpu\_to\_le32(29491200),

.dwMaxVideoFrameBufferSize = cpu\_to\_le32(1843200),

.dwDefaultFrameInterval = cpu\_to\_le32(5000000),

.bFrameIntervalType = 1,

.dwFrameInterval[0] = cpu\_to\_le32(5000000),

};

static const struct uvc\_color\_matching\_descriptor uvc\_color\_matching = {

.bLength = UVC\_DT\_COLOR\_MATCHING\_SIZE,

.bDescriptorType = USB\_DT\_CS\_INTERFACE,

.bDescriptorSubType = UVC\_VS\_COLORFORMAT,

.bColorPrimaries = 1,

.bTransferCharacteristics = 1,

.bMatrixCoefficients = 4,

};

## uvc\_bind\_config() //初始化uvc\_device和usb\_function: “uvc”

int \_\_init **uvc\_bind\_config**(**struct usb\_configuration** \*c,

const struct uvc\_descriptor\_header \* const \*fs\_control,

const struct uvc\_descriptor\_header \* const \*ss\_control,

const struct uvc\_descriptor\_header \* const \*fs\_streaming,

const struct uvc\_descriptor\_header \* const \*hs\_streaming,

const struct uvc\_descriptor\_header \* const \*ss\_streaming)

{

**struct uvc\_device \*uvc = kzalloc(sizeof(\*uvc), GFP\_KERNEL);**

uvc->state = UVC\_STATE\_DISCONNECTED;

//初始化uvc\_device的desc: Descriptors

uvc->desc.fs\_control = fs\_control;

uvc->desc.ss\_control = ss\_control;

uvc->desc.fs\_streaming = fs\_streaming;

uvc->desc.hs\_streaming = hs\_streaming;

uvc->desc.ss\_streaming = ss\_streaming;

if (uvc\_en\_us\_strings[UVC\_STRING\_CONTROL\_IDX].id == 0) { //初始化IDA Descripter

ret = usb\_string\_ids\_tab(c->cdev, uvc\_en\_us\_strings);

uvc\_iad.iFunction =uvc\_en\_us\_strings[UVC\_STRING\_CONTROL\_IDX].id;

uvc\_control\_intf.iInterface =uvc\_en\_us\_strings[UVC\_STRING\_CONTROL\_IDX].id;

ret = uvc\_en\_us\_strings[UVC\_STRING\_STREAMING\_IDX].id;

uvc\_streaming\_intf\_alt0.iInterface = ret;

uvc\_streaming\_intf\_alt1.iInterface = ret;

}

//初始化usb\_function

**uvc->func.name = "uvc";**

uvc->func.strings = uvc\_function\_strings;

**uvc->func.bind = uvc\_function\_bind;**

uvc->func.unbind = uvc\_function\_unbind;

uvc->func.get\_alt = uvc\_function\_get\_alt;

uvc->func.set\_alt = uvc\_function\_set\_alt;

uvc->func.disable = uvc\_function\_disable;

uvc->func.setup = uvc\_function\_setup;

ret = **usb\_add\_function**(c, &uvc->func);

}

## usb\_add\_function() //将usb\_function添加到usb\_configuration中

int **usb\_add\_function**(s**truct usb\_configuration** \*config,**struct usb\_function** \*function)

{

function->config = config;

**list\_add\_tail(&function->list, &config->functions);**

value = **function->bind**(config, function); // **uvc\_function\_bind**()

if (!config->fullspeed && function->fs\_descriptors) config->fullspeed = true;

if (!config->highspeed && function->hs\_descriptors) config->highspeed = true;

if (!config->superspeed && function->ss\_descriptors) config->superspeed = true;

}

## uvc\_function\_bind() //初始化usb\_endpoint\_descriptor, usb\_request

static int \_\_init **uvc\_function\_bind**(struct usb\_configuration \*c, struct usb\_function \*f)

{

struct uvc\_device \*uvc = to\_uvc(f);

//初始化usb\_endpoint\_descriptor: fs, hs ss

streaming\_interval = clamp(streaming\_interval, 1U, 16U);

streaming\_maxpacket = clamp(streaming\_maxpacket, 1U, 3072U);

streaming\_maxburst = min(streaming\_maxburst, 15U);

if (streaming\_maxpacket <= 1024) { max\_packet\_mult = 1; max\_packet\_size = streaming\_maxpacket; }

else if (streaming\_maxpacket <= 2048) { max\_packet\_mult = 2; max\_packet\_size = streaming\_maxpacket / 2; }

else { max\_packet\_mult = 3; max\_packet\_size = streaming\_maxpacket / 3;}

uvc\_fs\_streaming\_ep.wMaxPacketSize = min(streaming\_maxpacket, 1023U);

uvc\_fs\_streaming\_ep.bInterval = streaming\_interval;

uvc\_hs\_streaming\_ep.wMaxPacketSize = max\_packet\_size;

uvc\_hs\_streaming\_ep.wMaxPacketSize |= ((max\_packet\_mult - 1) << 11);

uvc\_hs\_streaming\_ep.bInterval = streaming\_interval;

uvc\_ss\_streaming\_ep.wMaxPacketSize = max\_packet\_size;

uvc\_ss\_streaming\_ep.bInterval = streaming\_interval;

uvc\_ss\_streaming\_comp.bmAttributes = max\_packet\_mult - 1;

uvc\_ss\_streaming\_comp.bMaxBurst = streaming\_maxburst;

uvc\_ss\_streaming\_comp.wBytesPerInterval = max\_packet\_size \* max\_packet\_mult \* streaming\_maxburst;

//初始化usb\_ep : uvc->control\_ep, uvc->video.ep

**struct usb\_ep** \*ep = **usb\_ep\_autoconfig**(cdev->gadget, &uvc\_control\_ep);

uvc->control\_ep = ep; //dwc3.control\_endpoint

ep->driver\_data = uvc;

if (gadget\_is\_superspeed(c->cdev->gadget))

ep = **usb\_ep\_autoconfig\_ss**(cdev->gadget, &uvc\_ss\_streaming\_ep,&uvc\_ss\_streaming\_comp);

else if (gadget\_is\_dualspeed(cdev->gadget)) ep = usb\_ep\_autoconfig(cdev->gadget, &uvc\_hs\_streaming\_ep);

else ep = usb\_ep\_autoconfig(cdev->gadget, &uvc\_fs\_streaming\_ep);

uvc->video.ep = ep; //dwc3.streaming\_endpoint

ep->driver\_data = uvc;

uvc\_fs\_streaming\_ep.bEndpointAddress = uvc->video.ep->address;

uvc\_hs\_streaming\_ep.bEndpointAddress = uvc->video.ep->address;

uvc\_ss\_streaming\_ep.bEndpointAddress = uvc->video.ep->address;

//初始化usb\_interface\_descriptor

ret = usb\_interface\_id(c, f); //id = c->next\_interface\_id; c->interface[id] = f; c->next\_interface\_id = id + 1

uvc\_iad.bFirstInterface = ret;

uvc\_control\_intf.bInterfaceNumber = ret;

uvc->control\_intf = ret;

ret = usb\_interface\_id(c, f); //id = c->next\_interface\_id; c->interface[id] = f; c->next\_interface\_id = id + 1

uvc\_streaming\_intf\_alt0.bInterfaceNumber = ret;

uvc\_streaming\_intf\_alt1.bInterfaceNumber = ret;

uvc->streaming\_intf = ret;

//分配内存，保存lsusb获取到的所有的descriptor内容到同一块buffer中

f->fs\_descriptors = uvc\_copy\_descriptors(uvc, USB\_SPEED\_FULL);

if (gadget\_is\_dualspeed(cdev->gadget)) f->hs\_descriptors = uvc\_copy\_descriptors(uvc, USB\_SPEED\_HIGH);

if (gadget\_is\_superspeed(c->cdev->gadget)) f->ss\_descriptors = uvc\_copy\_descriptors(uvc, USB\_SPEED\_SUPER);

//初始化usb\_request,

uvc->**control\_buf = kmalloc**(UVC\_MAX\_REQUEST\_SIZE, GFP\_KERNEL); // UVC\_MAX\_REQUEST\_SIZE=64

uvc->**control\_req = usb\_ep\_alloc\_request(**cdev->gadget->ep0, GFP\_KERNEL); // **dwc3\_gadget\_ep\_alloc\_request()**

uvc->control\_req->buf = uvc->control\_buf;

uvc->control\_req->complete = **uvc\_function\_ep0\_complete**;

uvc->control\_req->context = uvc;

ret = **usb\_function\_deactivate**(f); // **dwc3\_gadget\_pullup() ：off**

ret = **uvc\_video\_init**(&uvc->video);

ret = **uvc\_register\_video**(uvc);

return 0;

}

### struct usb\_endpoint\_descriptor uvc\_control

static struct usb\_endpoint\_descriptor uvc\_control\_ep \_\_initdata = {

.bLength = USB\_DT\_ENDPOINT\_SIZE,

.bDescriptorType = USB\_DT\_ENDPOINT,

.bEndpointAddress = USB\_DIR\_IN,

.bmAttributes = USB\_ENDPOINT\_**XFER\_INT,**

.wMaxPacketSize = cpu\_to\_le16(UVC\_STATUS\_MAX\_PACKET\_SIZE),

.bInterval = 8,

};

### struct usb\_endpoint\_descriptor uvc\_fs\_streaming\_ep

static struct usb\_endpoint\_descriptor uvc\_fs\_streaming\_ep \_\_initdata = {

.bLength = USB\_DT\_ENDPOINT\_SIZE,

.bDescriptorType = USB\_DT\_ENDPOINT,

.bEndpointAddress = USB\_DIR\_IN,

.bmAttributes = USB\_ENDPOINT\_SYNC\_ASYNC | USB\_ENDPOINT\_**XFER\_ISOC**,

.wMaxPacketSize = 0,

.bInterval = 0,

};

### struct usb\_endpoint\_descriptor uvc\_hs\_streaming\_ep

static struct usb\_endpoint\_descriptor uvc\_hs\_streaming\_ep \_\_initdata = {

.bLength = USB\_DT\_ENDPOINT\_SIZE,

.bDescriptorType = USB\_DT\_ENDPOINT,

.bEndpointAddress = USB\_DIR\_IN,

.bmAttributes = USB\_ENDPOINT\_SYNC\_ASYNC| USB\_ENDPOINT\_**XFER\_ISOC**,

.wMaxPacketSize = 0,

.bInterval = 0,

};

### struct usb\_endpoint\_descriptor uvc\_ss\_streaming\_ep

static struct usb\_endpoint\_descriptor uvc\_ss\_streaming\_ep \_\_initdata = {

.bLength = USB\_DT\_ENDPOINT\_SIZE,

.bDescriptorType = USB\_DT\_ENDPOINT,

.bEndpointAddress = USB\_DIR\_IN,

.bmAttributes = USB\_ENDPOINT\_SYNC\_ASYNC| USB\_ENDPOINT\_**XFER\_ISOC,**

.wMaxPacketSize = 0,

.bInterval = 0,

};

static struct usb\_ss\_ep\_comp\_descriptor uvc\_ss\_streaming\_comp \_\_initdata = {

.bLength = sizeof(uvc\_ss\_streaming\_comp),

.bDescriptorType = USB\_DT\_SS\_ENDPOINT\_COMP,

/\* The following 3 values can be tweaked if necessary. \*/

.bMaxBurst = 0,

.bmAttributes = 0,

.wBytesPerInterval = cpu\_to\_le16(1024),

};

### struct usb\_interface\_descriptor uvc\_streaming\_intf\_alt0

static struct usb\_interface\_descriptor uvc\_streaming\_intf\_alt0 \_\_initdata = {

.bLength = USB\_DT\_INTERFACE\_SIZE,

.bDescriptorType = USB\_DT\_INTERFACE,

.bInterfaceNumber = UVC\_INTF\_VIDEO\_STREAMING,

.bAlternateSetting = 0,

.bNumEndpoints = 0,

.bInterfaceClass = USB\_CLASS\_VIDEO,

.bInterfaceSubClass = UVC\_SC\_VIDEOSTREAMING,

.bInterfaceProtocol = 0x00,

.iInterface = 0,

};

### struct usb\_interface\_descriptor uvc\_streaming\_intf\_alt1

static struct usb\_interface\_descriptor uvc\_streaming\_intf\_alt1 \_\_initdata = {

.bLength = USB\_DT\_INTERFACE\_SIZE,

.bDescriptorType = USB\_DT\_INTERFACE,

.bInterfaceNumber = UVC\_INTF\_VIDEO\_STREAMING,

.bAlternateSetting = 1,

.bNumEndpoints = 1,

.bInterfaceClass = USB\_CLASS\_VIDEO,

.bInterfaceSubClass = UVC\_SC\_VIDEOSTREAMING,

.bInterfaceProtocol = 0x00,

.iInterface = 0,

};

## usb\_ep\_autoconfig()

struct usb\_ep \*usb\_ep\_autoconfig(struct usb\_gadget \*gadget,struct usb\_endpoint\_descriptor \*desc)

{

return usb\_ep\_autoconfig\_ss(gadget, desc, NULL);

}

## usb\_ep\_autoconfig\_ss()

struct usb\_ep \*usb\_ep\_autoconfig\_ss(struct usb\_gadget \*gadget,struct usb\_endpoint\_descriptor \*desc,struct usb\_ss\_ep\_comp\_descriptor \*ep\_comp)

{

struct usb\_ep \*ep;

list\_for\_each\_entry (ep, &gadget->ep\_list, ep\_list) { //ep = dwc3\_gadget\_init\_hw\_endpoints()

if (**ep\_matches**(gadget, ep, desc, ep\_comp)) //ep->name = "ep%d%s" = epnum >> 1,(epnum & 1) ? "in" : "out"

goto found\_ep;

}

found\_ep:

ep->desc = NULL;

ep->comp\_desc = NULL;

**return ep;**

}

## ep\_matches()

static int ep\_matches (struct usb\_gadget \*gadget, struct usb\_ep \*ep, struct usb\_endpoint\_descriptor \*desc,

struct usb\_ss\_ep\_comp\_descriptor \*ep\_comp)

{

//dwc3: ep->name = "ep%d%s" = epnum >> 1,(epnum & 1) ? "in" : "out"

if ('e' != ep->name[0]) return 0; //some other naming convention

type = desc->bmAttributes & USB\_ENDPOINT\_XFERTYPE\_MASK;

if ('-' != ep->name[2]) { //type-restriction: "-iso", "-bulk", or "-int"

tmp = strrchr (ep->name, '-');

if (tmp) { //"-iso", "-bulk", or "-int"

…

} else { tmp = ep->name + strlen (ep->name); }

tmp--;

if (!isdigit (\*tmp)) { //direction-restriction: "in", "out"

if (desc->bEndpointAddress & USB\_DIR\_IN) {if ('n' != \*tmp) return 0; }

else {if ('t' != \*tmp) return 0;}

}

}

if (usb\_endpoint\_xfer\_bulk(desc)) { //USB\_ENDPOINT\_XFER\_BULK

}

max = 0x7ff & usb\_endpoint\_maxp(desc);

switch (type) {

case USB\_ENDPOINT\_XFER\_INT:

if (!gadget\_is\_dualspeed(gadget) && max > 64) return 0;

case USB\_ENDPOINT\_XFER\_ISOC:

if (ep->maxpacket < max) return 0; //ep.maxpacket = 1024;

if (!gadget\_is\_dualspeed(gadget) && max > 1023) return 0;

if ((desc->wMaxPacketSize & cpu\_to\_le16(3<<11))) { if (!gadget\_is\_dualspeed(gadget)) return 0;}

break;

}

desc->bEndpointAddress &= USB\_DIR\_IN;

if (isdigit (ep->name [2])) {

u8 num = simple\_strtoul (&ep->name [2], NULL, 10); desc->bEndpointAddress |= num;

} else if (desc->bEndpointAddress & USB\_DIR\_IN) {

if (++gadget->in\_epnum > 15) return 0;

desc->bEndpointAddress = USB\_DIR\_IN | gadget->in\_epnum;

} else {

if (++gadget->out\_epnum > 15) return 0;

desc->bEndpointAddress |= gadget->out\_epnum;

}

if ((USB\_ENDPOINT\_XFER\_BULK == type) && !ep\_comp) {

int size = ep->maxpacket; if (size > 64) size = 64; desc->wMaxPacketSize = cpu\_to\_le16(size);

}

ep->address = desc->bEndpointAddress;

return 1;

}

## uvc\_video\_init() //初始化uvc\_video

static int uvc\_video\_init(struct uvc\_video \*video)

{

INIT\_LIST\_HEAD(&video->req\_free);

spin\_lock\_init(&video->req\_lock);

video->fcc = V4L2\_PIX\_FMT\_YUYV;

video->bpp = 16;

video->width = 320;

video->height = 240;

video->imagesize = 320 \* 240 \* 2;

**uvc\_queue\_init**(&video->queue, V4L2\_BUF\_TYPE\_**VIDEO\_OUTPUT**);

}

## uvc\_queue\_init() //初始化vb2\_queue

static int uvc\_queue\_init(struct uvc\_video\_queue \*queue, enum v4l2\_buf\_type type)

{

queue->queue.type = type; //V4L2\_BUF\_TYPE\_VIDEO\_OUTPUT

queue->queue.io\_modes = **VB2\_MMAP | VB2\_USERPTR**;

queue->queue.drv\_priv = queue;

queue->queue.buf\_struct\_size = sizeof(struct uvc\_buffer);

queue->queue.ops = **&uvc\_queue\_qops**;

queue->queue.mem\_ops = **&vb2\_vmalloc\_memops**;

ret = vb2\_queue\_init(&queue->queue);

mutex\_init(&queue->mutex);

spin\_lock\_init(&queue->irqlock);

INIT\_LIST\_HEAD(&queue->irqqueue);

queue->flags = 0;

}

### struct vb2\_ops uvc\_queue\_qops

static struct vb2\_ops uvc\_queue\_qops = {

.queue\_setup = uvc\_queue\_setup,

.buf\_prepare = uvc\_buffer\_prepare,

.buf\_queue = uvc\_buffer\_queue,

};

### struct vb2\_mem\_ops vb2\_vmalloc\_memops

const struct vb2\_mem\_ops vb2\_vmalloc\_memops = {

.alloc = vb2\_vmalloc\_alloc,

.put = vb2\_vmalloc\_put,

.get\_userptr = vb2\_vmalloc\_get\_userptr,

.put\_userptr = vb2\_vmalloc\_put\_userptr,

.map\_dmabuf = vb2\_vmalloc\_map\_dmabuf,

.unmap\_dmabuf = vb2\_vmalloc\_unmap\_dmabuf,

.attach\_dmabuf = vb2\_vmalloc\_attach\_dmabuf,

.detach\_dmabuf = vb2\_vmalloc\_detach\_dmabuf,

.vaddr = vb2\_vmalloc\_vaddr,

.mmap = vb2\_vmalloc\_mmap,

.num\_users = vb2\_vmalloc\_num\_users,

};

## uvc\_register\_video() //初始化video\_device

static int uvc\_register\_video(struct uvc\_device \*uvc)

{

struct usb\_composite\_dev \*cdev = uvc->func.config->cdev;

struct video\_device \*video = video\_device\_alloc();

video->parent = &cdev->gadget->dev;

video->fops = **&uvc\_v4l2\_fops**;

video->release = video\_device\_release;

strlcpy(video->name, cdev->gadget->name, sizeof(video->name));

uvc->vdev = video;

video\_set\_drvdata(video, uvc);

return **video\_register\_device**(video, VFL\_TYPE\_GRABBER, -1);

}

### struct v4l2\_file\_operations uvc\_v4l2\_fops

static struct v4l2\_file\_operations uvc\_v4l2\_fops = {

.owner = THIS\_MODULE,

.open = uvc\_v4l2\_open,

.release = uvc\_v4l2\_release,

.ioctl = uvc\_v4l2\_ioctl,

.mmap = uvc\_v4l2\_mmap,

.poll = uvc\_v4l2\_poll,

#ifndef CONFIG\_MMU

.get\_unmapped\_area = uvc\_v4l2\_get\_unmapped\_area,

#endif

};

## usb\_gadget\_udc\_start()

static inline int usb\_gadget\_udc\_start(**struct usb\_gadget** \*gadget, **struct usb\_gadget\_driver** \*driver)

{

return gadget->ops->udc\_start(gadget, driver); // **dwc3\_gadget\_start()**

}

## dwc3\_gadget\_start() //irq(), HW\_config(), usb\_cmd(),

static int dwc3\_gadget\_start(struct usb\_gadget \*g, struct usb\_gadget\_driver \*driver)

{

struct dwc3 \*dwc = gadget\_to\_dwc(g);

struct dwc3\_ep \*dep;

irq = platform\_get\_irq(to\_platform\_device(dwc->dev), 0);

request\_threaded\_irq(irq, **dwc3\_interrupt**, **dwc3\_thread\_interrupt**, IRQF\_SHARED | IRQF\_ONESHOT, "dwc3", dwc);

spin\_lock\_irqsave(&dwc->lock, flags);

dwc->gadget\_driver = driver;

reg = dwc3\_readl(dwc->regs, DWC3\_DCFG);

reg &= ~(DWC3\_DCFG\_SPEED\_MASK);

if (dwc->revision < DWC3\_REVISION\_220A) reg |= DWC3\_DCFG\_SUPERSPEED;

else reg |= dwc->maximum\_speed;

dwc3\_writel(dwc->regs, DWC3\_DCFG, reg);

dwc->start\_config\_issued = false;

**dwc3\_gadget\_ep0\_desc.wMaxPacketSize** = cpu\_to\_le16(512);

dep = dwc->eps[0];

ret = **\_\_dwc3\_gadget\_ep\_enable**(dep, &**dwc3\_gadget\_ep0\_desc**, NULL, false);

dep = dwc->eps[1];

ret = **\_\_dwc3\_gadget\_ep\_enable**(dep, &**dwc3\_gadget\_ep0\_desc,** NULL, false);

dwc->ep0state = EP0\_SETUP\_PHASE;

**dwc3\_ep0\_out\_start**(dwc); //cmd = DWC3\_DEPCMD\_STARTTRANSFER

**dwc3\_gadget\_enable\_irq**(dwc); // dwc3\_writel(dwc->regs, DWC3\_DEVTEN, reg); //#define DWC3\_DEVTEN 0xc708

spin\_unlock\_irqrestore(&dwc->lock, flags);

}

### struct usb\_endpoint\_descriptor dwc3\_gadget\_ep0\_desc

static struct usb\_endpoint\_descriptor dwc3\_gadget\_ep0\_desc = {

.bLength = USB\_DT\_ENDPOINT\_SIZE,

.bDescriptorType = USB\_DT\_ENDPOINT,

.bmAttributes = USB\_ENDPOINT\_**XFER\_CONTROL**,

};

## \_\_dwc3\_gadget\_ep\_enable() //初始化dwc3\_trb

static int \_\_dwc3\_gadget\_ep\_enable(struct dwc3\_ep \*dep, const struct usb\_endpoint\_descriptor \*desc,

const struct usb\_ss\_ep\_comp\_descriptor \*comp\_desc, bool ignore)

{

struct dwc3 \*dwc = dep->dwc;

if (!(dep->flags & DWC3\_EP\_ENABLED))

ret = **dwc3\_gadget\_start\_config**(dwc, dep); // cmd = DWC3\_DEPCMD\_DEPSTARTCFG

ret = **dwc3\_gadget\_set\_ep\_config**(dwc, dep, desc, comp\_desc, ignore); //cmd = DWC3\_DEPCMD\_SETEPCONFIG

if (!(dep->flags & DWC3\_EP\_ENABLED)) {

ret = dwc3\_gadget\_set\_xfer\_resource(dwc, dep);

**dep->endpoint.desc = desc;**

dep->comp\_desc = comp\_desc;

dep->type = usb\_endpoint\_type(desc);

**dep->flags |= DWC3\_EP\_ENABLED;**

reg = dwc3\_readl(dwc->regs, DWC3\_DALEPENA);

reg |= DWC3\_DALEPENA\_EP(dep->number);

dwc3\_writel(dwc->regs, DWC3\_DALEPENA, reg);

if (!usb\_endpoint\_xfer\_isoc(desc)) return 0;

memset(&trb\_link, 0, sizeof(trb\_link));

struct dwc3\_trb \*trb\_st\_hw = &dep->trb\_pool[0];

**struct dwc3\_trb** \*trb\_link = &dep->trb\_pool[DWC3\_TRB\_NUM - 1];

trb\_link->bpl = lower\_32\_bits(dwc3\_trb\_dma\_offset(dep, trb\_st\_hw));

trb\_link->bph = upper\_32\_bits(dwc3\_trb\_dma\_offset(dep, trb\_st\_hw));

trb\_link->ctrl |= DWC3\_TRBCTL\_LINK\_TRB;

trb\_link->ctrl |= DWC3\_TRB\_CTRL\_HWO;

}

}

# echo USB\_B\_IN > /sys/monitor/usb\_port/config/usb\_con\_msg

## dwc3\_gadget\_start() //irq(), HW\_config(), usb\_cmd(),

static int dwc3\_gadget\_start(struct usb\_gadget \*g, struct usb\_gadget\_driver \*driver)

{

struct dwc3 \*dwc = gadget\_to\_dwc(g);

struct dwc3\_ep \*dep;

irq = platform\_get\_irq(to\_platform\_device(dwc->dev), 0);

request\_threaded\_irq(irq, **dwc3\_interrupt**, **dwc3\_thread\_interrupt**, IRQF\_SHARED | IRQF\_ONESHOT, "dwc3", dwc);

spin\_lock\_irqsave(&dwc->lock, flags);

dwc->gadget\_driver = driver;

reg = dwc3\_readl(dwc->regs, DWC3\_DCFG);

reg &= ~(DWC3\_DCFG\_SPEED\_MASK);

if (dwc->revision < DWC3\_REVISION\_220A) reg |= DWC3\_DCFG\_SUPERSPEED;

else reg |= dwc->maximum\_speed;

dwc3\_writel(dwc->regs, DWC3\_DCFG, reg);

dwc->start\_config\_issued = false;

**dwc3\_gadget\_ep0\_desc.wMaxPacketSize** = cpu\_to\_le16(512);

dep = dwc->eps[0];

ret = **\_\_dwc3\_gadget\_ep\_enable**(dep, &**dwc3\_gadget\_ep0\_desc**, NULL, false);

dep = dwc->eps[1];

ret = **\_\_dwc3\_gadget\_ep\_enable**(dep, &**dwc3\_gadget\_ep0\_desc,** NULL, false);

dwc->ep0state = EP0\_SETUP\_PHASE;

**dwc3\_ep0\_out\_start**(dwc); //cmd = DWC3\_DEPCMD\_STARTTRANSFER

**dwc3\_gadget\_enable\_irq**(dwc); // dwc3\_writel(dwc->regs, DWC3\_DEVTEN, reg); //#define DWC3\_DEVTEN 0xc708

spin\_unlock\_irqrestore(&dwc->lock, flags);

}

## \_\_dwc3\_gadget\_ep\_enable() //初始化dwc3\_trb

static int \_\_dwc3\_gadget\_ep\_enable(struct dwc3\_ep \*dep, const struct usb\_endpoint\_descriptor \*desc,

const struct usb\_ss\_ep\_comp\_descriptor \*comp\_desc, bool ignore)

{

struct dwc3 \*dwc = dep->dwc;

if (!(dep->flags & DWC3\_EP\_ENABLED))

ret = **dwc3\_gadget\_start\_config**(dwc, dep); // cmd = DWC3\_DEPCMD\_DEPSTARTCFG

ret = **dwc3\_gadget\_set\_ep\_config**(dwc, dep, desc, comp\_desc, ignore); //cmd = DWC3\_DEPCMD\_SETEPCONFIG

if (!(dep->flags & DWC3\_EP\_ENABLED)) {

ret = dwc3\_gadget\_set\_xfer\_resource(dwc, dep);

**dep->endpoint.desc = desc;**

dep->comp\_desc = comp\_desc;

dep->type = usb\_endpoint\_type(desc);

**dep->flags |= DWC3\_EP\_ENABLED;**

reg = dwc3\_readl(dwc->regs, DWC3\_DALEPENA);

reg |= DWC3\_DALEPENA\_EP(dep->number);

dwc3\_writel(dwc->regs, DWC3\_DALEPENA, reg);

if (!usb\_endpoint\_xfer\_isoc(desc)) return 0;

memset(&trb\_link, 0, sizeof(trb\_link));

struct dwc3\_trb \*trb\_st\_hw = &dep->trb\_pool[0];

**struct dwc3\_trb** \*trb\_link = &dep->trb\_pool[DWC3\_TRB\_NUM - 1];

trb\_link->bpl = lower\_32\_bits(dwc3\_trb\_dma\_offset(dep, trb\_st\_hw));

trb\_link->bph = upper\_32\_bits(dwc3\_trb\_dma\_offset(dep, trb\_st\_hw));

trb\_link->ctrl |= DWC3\_TRBCTL\_LINK\_TRB;

trb\_link->ctrl |= DWC3\_TRB\_CTRL\_HWO;

}

}

# App:

## Open()

open(devname, O\_RDWR | O\_NONBLOCK);

static int uvc\_v4l2\_open(struct file \*file)

{

struct video\_device \*vdev = video\_devdata(file);

struct uvc\_device \*uvc = video\_get\_drvdata(vdev);

struct uvc\_file\_handle \*handle = kzalloc(sizeof(\*handle), GFP\_KERNEL);

v4l2\_fh\_init(&handle->vfh, vdev);

v4l2\_fh\_add(&handle->vfh);

handle->device = &uvc->video;

file->private\_data = &handle->vfh;

**uvc\_function\_connect**(uvc);

}

void uvc\_function\_connect(struct uvc\_device \*uvc)

{

ret = **usb\_function\_activate**(&uvc->func);

}

int usb\_function\_activate(struct usb\_function \*function)

{

struct usb\_composite\_dev \*cdev = function->config->cdev;

**//cdev->deactivations = 1, function->name = uvc**

if (WARN\_ON(cdev->deactivations == 0)) status = -EINVAL;

else {

cdev->deactivations--;

if (cdev->deactivations == 0)

status = **usb\_gadget\_connect**(cdev->gadget); //gadget->ops->pullup(gadget, 1); // **dwc3\_gadget\_pullup()**

}

}

## dwc3\_thread\_interrupt()

static irqreturn\_t dwc3\_thread\_interrupt(int irq, void \*\_dwc)

{

struct dwc3 \*dwc = \_dwc;

for (i = 0; i < dwc->num\_event\_buffers; i++) {

struct dwc3\_event\_buffer \* evt = dwc->ev\_buffs[i];

left = evt->count;

if (!(evt->flags & DWC3\_EVENT\_PENDING)) continue;

while (left > 0) {

union dwc3\_event event;

event.raw = \*(u32 \*) (evt->buf + evt->lpos);

**dwc3\_process\_event\_entry(dwc, &event);**

evt->lpos = (evt->lpos + 4) % DWC3\_EVENT\_BUFFERS\_SIZE;

left -= 4;

dwc3\_writel(dwc->regs, DWC3\_GEVNTCOUNT(i), 4);

}

evt->count = 0;

evt->flags &= ~DWC3\_EVENT\_PENDING;

ret = IRQ\_HANDLED;

}

}

## dwc3\_process\_event\_entry()

static void dwc3\_process\_event\_entry(struct dwc3 \*dwc,const union dwc3\_event \*event)

{

if (event->type.is\_devspec == 0) return **dwc3\_endpoint\_interrupt**(dwc, &event->depevt);

switch (event->type.type) {

case DWC3\_EVENT\_TYPE\_DEV: dwc3\_gadget\_interrupt(dwc, &event->devt); break;

default: dev\_err(dwc->dev, "UNKNOWN IRQ type %d\n", event->raw);

}

}

## dwc3\_endpoint\_interrupt()

static void dwc3\_endpoint\_interrupt(struct dwc3 \*dwc, const struct dwc3\_event\_depevt \*event)

{

struct dwc3\_ep \*dep;

u8 epnum = event->endpoint\_number;

dep = dwc->eps[epnum];

if (epnum == 0 || epnum == 1) { **dwc3\_ep0\_interrupt**(dwc, event); return; }

switch (event->endpoint\_event) {

case DWC3\_DEPEVT\_XFERCOMPLETE:

dep->resource\_index = 0;

if (usb\_endpoint\_xfer\_isoc(dep->endpoint.desc)) { return; }

dwc3\_endpoint\_transfer\_complete(dwc, dep, event, 1);

break;

case DWC3\_DEPEVT\_XFERNOTREADY:

if (usb\_endpoint\_xfer\_isoc(dep->endpoint.desc)) { dwc3\_gadget\_start\_isoc(dwc, dep, event); }

else { ret = \_\_dwc3\_gadget\_kick\_transfer(dep, 0, 1); }

break;

}

}

## dwc3\_ep0\_interrupt()

void dwc3\_ep0\_interrupt(struct dwc3 \*dwc, const struct dwc3\_event\_depevt \*event)

{

u8 epnum = event->endpoint\_number;

//Transfer Complete while ep0out in state 'Setup Phase'

switch (event->endpoint\_event) {

case DWC3\_DEPEVT\_XFERCOMPLETE: **dwc3\_ep0\_xfer\_complete**(dwc, event); break;

case DWC3\_DEPEVT\_XFERNOTREADY: dwc3\_ep0\_xfernotready(dwc, event); break;

case DWC3\_DEPEVT\_XFERINPROGRESS:

case DWC3\_DEPEVT\_RXTXFIFOEVT:

case DWC3\_DEPEVT\_STREAMEVT:

case DWC3\_DEPEVT\_EPCMDCMPLT:

break;

}

}

## dwc3\_ep0\_xfer\_complete()

static void dwc3\_ep0\_xfer\_complete(struct dwc3 \*dwc, const struct dwc3\_event\_depevt \*event)

{

struct dwc3\_ep \*dep = dwc->eps[event->endpoint\_number];

dep->flags &= ~DWC3\_EP\_BUSY;

dep->resource\_index = 0;

dwc->setup\_packet\_pending = false;

switch (dwc->ep0state) {

case EP0\_SETUP\_PHASE: **dwc3\_ep0\_inspect\_setup**(dwc, event); break;

case EP0\_DATA\_PHASE: dwc3\_ep0\_complete\_data(dwc, event); break;

case EP0\_STATUS\_PHASE: dwc3\_ep0\_complete\_status(dwc, event); break;

default:

WARN(true, "UNKNOWN ep0state %d\n", dwc->ep0state);

}

}

## dwc3\_ep0\_inspect\_setup()

static void dwc3\_ep0\_inspect\_setup(struct dwc3 \*dwc, const struct dwc3\_event\_depevt \*event)

{

struct usb\_ctrlrequest \*ctrl = dwc->ctrl\_req;

len = le16\_to\_cpu(ctrl->wLength);

if (!len) {

dwc->three\_stage\_setup = false;

dwc->ep0\_expect\_in = false;

dwc->ep0\_next\_event = DWC3\_EP0\_NRDY\_STATUS;

} else {

dwc->three\_stage\_setup = true;

dwc->ep0\_expect\_in = !!(ctrl->bRequestType & USB\_DIR\_IN);

dwc->ep0\_next\_event = DWC3\_EP0\_NRDY\_DATA;

}

if ((ctrl->bRequestType & USB\_TYPE\_MASK) == USB\_TYPE\_STANDARD) ret = **dwc3\_ep0\_std\_request**(dwc, ctrl);

else ret = **dwc3\_ep0\_delegate\_req**(dwc, ctrl);

if (ret == USB\_GADGET\_DELAYED\_STATUS) dwc->delayed\_status = true;

if (ret < 0) dwc3\_ep0\_stall\_and\_restart(dwc);

}

## dwc3\_ep0\_std\_request()

**[ 481.162029] [dwc3\_ep0\_std\_request:705] hongyan ctrl->bRequest = 0x5 //USB\_REQ\_SET\_ADDRESS**

**[ 481.192060] [dwc3\_ep0\_std\_request:705] hongyan ctrl->bRequest = 0x6 //USB\_REQ\_GET\_DESCRIPTOR**

**[ 481.200998] [dwc3\_ep0\_std\_request:705] hongyan ctrl->bRequest = 0x9 //USB\_REQ\_SET\_CONFIGURATION**

**[ 481.203060] [dwc3\_ep0\_std\_request:705] hongyan ctrl->bRequest = 0xb //USB\_REQ\_SET\_INTERFACE**

static int dwc3\_ep0\_std\_request(struct dwc3 \*dwc, struct usb\_ctrlrequest \*ctrl)

{

switch (ctrl->bRequest) {

case USB\_REQ\_GET\_STATUS: ret = dwc3\_ep0\_handle\_status(dwc, ctrl); break;

case USB\_REQ\_CLEAR\_FEATURE: ret = dwc3\_ep0\_handle\_feature(dwc, ctrl, 0); break;

case USB\_REQ\_SET\_FEATURE: ret = dwc3\_ep0\_handle\_feature(dwc, ctrl, 1); break;

case USB\_REQ**\_SET\_ADDRESS**: ret = **dwc3\_ep0\_set\_address**(dwc, ctrl); break;

case USB\_REQ\_**SET\_CONFIGURATION**: ret = **dwc3\_ep0\_set\_config**(dwc, ctrl); break;

case USB\_REQ\_SET\_SEL: ret = dwc3\_ep0\_set\_sel(dwc, ctrl); break;

case USB\_REQ\_SET\_ISOCH\_DELAY: ret = dwc3\_ep0\_set\_isoch\_delay(dwc, ctrl); break;

default: ret = **dwc3\_ep0\_delegate\_req**(dwc, ctrl); break;

};

}

## dwc3\_ep0\_delegate\_req()

static int dwc3\_ep0\_delegate\_req(struct dwc3 \*dwc, struct usb\_ctrlrequest \*ctrl)

{

ret = dwc->gadget\_driver->setup(&dwc->gadget, ctrl);

}

## composite\_setup()

int composite\_setup(struct usb\_gadget \*gadget, const struct usb\_ctrlrequest \*ctrl)

{

struct usb\_composite\_dev \*cdev = get\_gadget\_data(gadget);

struct usb\_request \*req = cdev->req;

u16 w\_index = le16\_to\_cpu(ctrl->wIndex);

u8 intf = w\_index & 0xFF;

u16 w\_value = le16\_to\_cpu(ctrl->wValue);

u16 w\_length = le16\_to\_cpu(ctrl->wLength);

struct usb\_function \*f = NULL;

req->zero = 0;

req->complete = composite\_setup\_complete;

req->length = 0;

gadget->ep0->driver\_data = cdev;

switch (ctrl->bRequest) {

case USB\_REQ\_**GET\_DESCRIPTOR**:

switch (w\_value >> 8) {

case USB\_DT\_DEVICE:

cdev->desc.bNumConfigurations = count\_configs(cdev, USB\_DT\_DEVICE);

cdev->desc.bMaxPacketSize0 = cdev->gadget->ep0->maxpacket;

if (gadget\_is\_superspeed(gadget)) {

if (gadget->speed >= USB\_SPEED\_SUPER) {

cdev->desc.bcdUSB = cpu\_to\_le16(0x0300);

cdev->desc.bMaxPacketSize0 = 9;

} else { cdev->desc.bcdUSB = cpu\_to\_le16(0x0210); }

}

value = min(w\_length, (u16) sizeof cdev->desc);

memcpy(req->buf, &cdev->desc, value);

break;

case USB\_DT\_CONFIG:

value = config\_desc(cdev, w\_value);

if (value >= 0) value = min(w\_length, (u16) value);

break;

case USB\_DT\_STRING:

value = get\_string(cdev, req->buf,w\_index, w\_value & 0xff);

if (value >= 0) value = min(w\_length, (u16) value);

break;

}

break;

case USB\_REQ\_**SET\_CONFIGURATION**: value = **set\_config**(cdev, ctrl, w\_value); break;

case USB\_REQ\_**SET\_INTERFACE**:

f = cdev->config->interface[intf];

value = **f->set\_alt**(f, w\_index, w\_value); //set\_alt(1, 0)

break;

**default:**

**unknown:**

switch (ctrl->bRequestType & USB\_RECIP\_MASK) {

case USB\_RECIP\_INTERFACE: f = cdev->config->interface[intf]; break;

case USB\_RECIP\_ENDPOINT:

endp = ((w\_index & 0x80) >> 3) | (w\_index & 0x0f);

list\_for\_each\_entry(f, &cdev->config->functions, list) if (test\_bit(endp, f->endpoints)) break;

break;

}

if (f && f->setup) value = **f->setup**(f, ctrl);

else { struct usb\_configuration \*c = cdev->config; if (c && c->setup) value = c->setup(c, ctrl);}

goto done;

}

if (value >= 0 && value != USB\_GADGET\_DELAYED\_STATUS) {

req->length = value;

req->zero = value < w\_length;

value = usb\_ep\_queue(gadget->ep0, req, GFP\_ATOMIC);

if (value < 0) { req->status = 0; composite\_setup\_complete(gadget->ep0, req);

}

}

done:

return value;

}

## uvc\_function\_setup()

static int uvc\_function\_setup(struct usb\_function \*f, const struct usb\_ctrlrequest \*ctrl)

{

struct uvc\_device \*uvc = to\_uvc(f);

struct v4l2\_event v4l2\_event;

struct uvc\_event \*uvc\_event = (void \*)&v4l2\_event.u.data;

memset(&v4l2\_event, 0, sizeof(v4l2\_event));

v4l2\_event.type = UVC\_EVENT\_SETUP;

memcpy(&uvc\_event->req, ctrl, sizeof(uvc\_event->req));

v4l2\_event\_queue(uvc->vdev, &v4l2\_event);

}

## UVC\_EVENT\_SETUP

sub.type = UVC\_EVENT\_SETUP;

ioctl(dev->fd, VIDIOC\_SUBSCRIBE\_EVENT, &sub);

## =========================:

## USB\_REQ\_CLEAR\_FEATURE

### composite\_setup()

**//bRequestType=21, bRequest=01, w\_value=0100, w\_index=0001, w\_length= 26**

**//bRequestType=21, bRequest=01, w\_value=0200, w\_index=0001, w\_length= 26**

**//bRequestType = 0x21 = USB\_DIR\_OUT | USB\_TYPE\_CLASS | USB\_RECIP\_INTERFACE**

**//bRequest = 0x01 = USB\_REQ\_CLEAR\_FEATURE**

int composite\_setup(struct usb\_gadget \*gadget, const struct usb\_ctrlrequest \*ctrl)

{

switch (ctrl->bRequest) {

case **USB\_REQ\_CLEAR\_FEATURE**:

case USB\_REQ\_SET\_FEATURE:

if (ctrl->bRequestType != (USB\_DIR\_OUT | USB\_RECIP\_INTERFACE)) //(0x00 | 0x01) **goto unknown**;

break;

default:

**unknown:**

switch (ctrl->bRequestType & USB\_RECIP\_MASK) {

case USB\_RECIP\_INTERFACE: **f = cdev->config->interface[intf];**  break;

case USB\_RECIP\_ENDPOINT:

endp = ((w\_index & 0x80) >> 3) | (w\_index & 0x0f);

list\_for\_each\_entry(f, &cdev->config->functions, list) { if (test\_bit(endp, f->endpoints)) break; }

if (&f->list == &cdev->config->functions) f = NULL;

break;

}

if (f && f->setup) { **value = f->setup(f, ctrl);** // uvc\_function\_setup() }

}

}

}

### uvc\_function\_setup() : UVC\_EVENT\_SETUP

static int uvc\_function\_setup(struct usb\_function \*f, const struct usb\_ctrlrequest \*ctrl)

{

struct uvc\_device \*uvc = to\_uvc(f);

struct v4l2\_event v4l2\_event;

struct uvc\_event \*uvc\_event = (void \*)&v4l2\_event.u.data;

memset(&v4l2\_event, 0, sizeof(v4l2\_event));

v4l2\_event.type = **UVC\_EVENT\_SETUP**;

memcpy(&uvc\_event->req, ctrl, sizeof(uvc\_event->req));

v4l2\_event\_queue(uvc->vdev, &v4l2\_event);

}

## UVC\_EVENT\_SETUP : UVC\_SET\_CUR

### uvc\_gadget\_events\_process()

**//bRequestType=21, bRequest=01, w\_value=0200, w\_index=0001, w\_length= 26**

**//bRequestType = 0x21 = USB\_DIR\_OUT | USB\_TYPE\_CLASS | USB\_RECIP\_INTERFACE**

static void uvc\_gadget\_events\_process(struct uvc\_gadget \*gadget)

{

ret = **v4l2\_dequeue\_event**(gadget->out->fd, &v4l2\_event);

switch (v4l2\_event.type) {

case **UVC\_EVENT\_SETUP**:

memset(&resp, 0, sizeof(struct uvc\_request\_data));

resp.length = -EL2HLT;

**uvc\_gadget\_events\_process\_setup**(gadget, &uvc\_event->req, &resp);

uvc\_send\_response(gadget->out->fd, &resp);

break;

}

}

### v4l2\_dequeue\_event()

static inline int v4l2\_dequeue\_event(int fd, struct v4l2\_event \*event)

{

ret = xioctl(fd, **VIDIOC\_DQEVENT**, event);

}

### uvc\_v4l2\_do\_ioctl()

**//bRequestType=21, bRequest=01, w\_value=0200, w\_index=0001, w\_length= 26**

**//bRequestType = 0x21 = USB\_DIR\_OUT | USB\_TYPE\_CLASS | USB\_RECIP\_INTERFACE**

static long uvc\_v4l2\_do\_ioctl(struct file \*file, unsigned int cmd, void \*arg)

{

case VIDIOC\_**DQEVENT**:

{

struct v4l2\_event \*event = arg;

ret = **v4l2\_event\_dequeue**(&handle->vfh, event,file->f\_flags & O\_NONBLOCK);

if (ret == 0 && **event->type == UVC\_EVENT\_SETUP**) {

struct uvc\_event \*uvc\_event = (void \*)&event->u.data;

**uvc->event\_setup\_out = !(uvc\_event->req.bRequestType & USB\_DIR\_IN);** **//bRequestType: b[7]=USB\_DIR\_OUT=0**

uvc->event\_length = uvc\_event->req.wLength;

}

return ret;

}

}

### uvc\_function\_ep0\_complete():UVC\_EVENT\_DATA

static void uvc\_function\_ep0\_complete(struct usb\_ep \*ep, struct usb\_request \*req)

{

struct uvc\_device \*uvc = req->context;

struct v4l2\_event v4l2\_event;

struct uvc\_event \*uvc\_event = (void \*)&v4l2\_event.u.data;

**if (uvc->event\_setup\_out)** {

uvc->event\_setup\_out = 0;

memset(&v4l2\_event, 0, sizeof(v4l2\_event));

v4l2\_event.type = **UVC\_EVENT\_DATA**;

uvc\_event->data.length = req->actual;

memcpy(&uvc\_event->data.data, req->buf, req->actual);

v4l2\_event\_queue(uvc->vdev, &v4l2\_event);

}

}

### uvc\_gadget\_events\_process\_setup()

**//bRequestType=21, bRequest=01, w\_value=0200, w\_index=0001, w\_length= 26**

static void uvc\_gadget\_events\_process\_setup(struct uvc\_gadget \*gadget,struct usb\_ctrlrequest \*ctrl,struct uvc\_request\_data \*resp)

{

switch (ctrl->bRequestType & USB\_TYPE\_MASK) {

case **USB\_TYPE\_CLASS**: **uvc\_gadget\_events\_process\_class**(gadget, ctrl, resp); break;

}

}

static void uvc\_gadget\_events\_process\_class(struct uvc\_gadget \*gadget,struct usb\_ctrlrequest \*ctrl,struct uvc\_request\_data \*resp)

{

switch (ctrl->wIndex & 0xff) {

case **UVC\_INTF\_STREAMING: uvc\_gadget\_events\_process\_streaming**(gadget, ctrl->bRequest,ctrl->wValue >> 8, resp);break;

}

}

### uvc\_gadget\_events\_process\_streaming()

**//bRequestType=21, bRequest=01, w\_value=0200, w\_index=0001, w\_length= 26**

**//bRequestType = 0x21 = USB\_DIR\_OUT | USB\_TYPE\_CLASS | USB\_RECIP\_INTERFACE**

**// bRequest=0x01 =** **UVC\_SET\_CUR**

**// wIndex=0x0001= UVC\_INTF\_STREAMING**

**// cs = w\_value>>8 = 0x0200>>8 = 0x02 = UVC\_VS\_COMMIT\_CONTROL**

static void uvc\_gadget\_events\_process\_streaming(struct uvc\_gadget \*gadget, uint8\_t req, uint8\_t cs, struct uvc\_request\_data \*resp)

{

switch (req) {

case UVC\_SET\_CUR: gadget->control = cs; break;

}

}

## UVC\_EVENT\_DATA: COMMIT

### uvc\_gadget\_events\_process()

**//bRequestType=21, bRequest=01, w\_value=0200, w\_index=0001, w\_length= 26**

**//bRequestType = 0x21 = USB\_DIR\_OUT | USB\_TYPE\_CLASS | USB\_RECIP\_INTERFACE**

static void uvc\_gadget\_events\_process(struct uvc\_gadget \*gadget)

{

ret = **v4l2\_dequeue\_event**(gadget->out->fd, &v4l2\_event);

switch (v4l2\_event.type) {

case **UVC\_EVENT\_DATA**: uvc\_gadget\_events\_process\_data(gadget, &uvc\_event->data); break;

}

}

### uvc\_gadget\_events\_process\_data()

**//bRequestType=21, bRequest=01, w\_value=0200, w\_index=0001, w\_length= 26**

**//bRequestType = 0x21 = USB\_DIR\_OUT | USB\_TYPE\_CLASS | USB\_RECIP\_INTERFACE**

**// bRequest=0x01 =** **UVC\_SET\_CUR**

**// wIndex=0x0001= UVC\_INTF\_STREAMING**

**// cs = w\_value>>8 = 0x0200>>8 = 0x02 = UVC\_VS\_COMMIT\_CONTROL**

static void uvc\_gadget\_events\_process\_data(struct uvc\_gadget \*gadget, struct uvc\_request\_data \*data)

{

struct uvc\_streaming\_control \*target;

struct uvc\_streaming\_control \*ctrl;

switch (gadget->control) {

case UVC\_VS\_PROBE\_CONTROL: target = &gadget->probe; break;

case **UVC\_VS\_COMMIT\_CONTRO**L: **target = &gadget->commit**; break;

}

ctrl = (struct uvc\_streaming\_control \*)&data->data;

iformat = CLAMP((unsigned int)ctrl->bFormatIndex, 1U, (unsigned int)ARRAY\_SIZE(uvc\_gadget\_formats));

**format = &uvc\_gadget\_formats[iformat - 1];**

nframes = 0;

while (format->frames[nframes].width != 0) nframes++;

iframe = CLAMP((unsigned int)ctrl->bFrameIndex, (unsigned int)1U, nframes);

**frame = &format->frames[iframe - 1];**

interval = frame->intervals;

while (interval[0] < ctrl->dwFrameInterval && interval[1]) interval++;

**target->bFormatIndex = iformat;**

**target->bFrameIndex = iframe;**

switch (format->fcc) {

case V4L2\_PIX\_FMT\_YUYV: target->dwMaxVideoFrameSize = frame->width \* frame->height \* 2; break;

}

target->dwFrameInterval = \*interval;

if (gadget->control == **UVC\_VS\_COMMIT\_CONTROL**) {

**gadget->out->fcc = format->fcc;**

**gadget->out->width = frame->width;**

**gadget->out->height = frame->height;**

**uvc\_gadget\_set\_format**(gadget->out,(format->fcc == V4L2\_PIX\_FMT\_MJPEG) ?gadget->imgsize : 0);

}

}

#### struct uvc\_gadget\_format\_info uvc\_gadget\_formats

static const struct uvc\_gadget\_format\_info uvc\_gadget\_formats[] = {

{ V4L2\_PIX\_FMT\_YUYV, uvc\_gadget\_frames\_yuyv },

//{ V4L2\_PIX\_FMT\_NV12, uvc\_gadget\_frames\_nv12 },

{ V4L2\_PIX\_FMT\_MJPEG, uvc\_gadget\_frames\_mjpg },

};

#### struct uvc\_gadget\_frame\_info uvc\_gadget\_frames\_yuyv

static const struct uvc\_gadget\_frame\_info uvc\_gadget\_frames\_yuyv[] = {

{ 176, 144, { 666666, 1000000, 5000000, 0 }, },

{ 640, 360, { 666666, 1000000, 5000000, 0 }, },

{ 1280, 720, { 50000000, 0 }, },

{ 0, 0, { 0, }, },

};

## USB\_REQ\_SET\_INTERFACE

### composite\_setup()

**// bRequestType=01, bRequest=0b, w\_value=0001, w\_index=0001, w\_length= 0**

int composite\_setup(struct usb\_gadget \*gadget, const struct usb\_ctrlrequest \*ctrl)

{

case USB\_REQ\_SET\_INTERFACE:

f = cdev->config->interface[intf];

value = **f->set\_alt**(f, w\_index, w\_value); // set\_alt(1, 1) : uvc\_function\_set\_alt()

if (value == USB\_GADGET\_DELAYED\_STATUS) {

cdev->delayed\_status++;

}

break;

}

### uvc\_function\_set\_alt()

static int uvc\_function\_set\_alt(struct usb\_function \*f, unsigned interface, unsigned alt)

{

struct uvc\_device \*uvc = to\_uvc(f);

struct v4l2\_event v4l2\_event;

struct uvc\_event \*uvc\_event = (void \*)&v4l2\_event.u.data;

switch (alt) {

**case 1:**

if (uvc->video.ep) {

ret = **config\_ep\_by\_speed**(f->config->cdev->gadget,&(uvc->func), uvc->video.ep);

**usb\_ep\_enable**(uvc->video.ep);

}

memset(&v4l2\_event, 0, sizeof(v4l2\_event));

**v4l2\_event.type = UVC\_EVENT\_STREAMON;**

v4l2\_event\_queue(uvc->vdev, &v4l2\_event);

return USB\_GADGET\_DELAYED\_STATUS;

}

}

## UVC\_EVENT\_STREAMON:

### uvc\_gadget\_events\_process()

static void uvc\_gadget\_events\_process(struct uvc\_gadget \*gadget)

{

ret = v4l2\_dequeue\_event(gadget->out->fd, &v4l2\_event);

switch (v4l2\_event.type) {

case **UVC\_EVENT\_STREAMON:**

**uvc\_gadget\_reqbufs**(gadget->out, NR\_VIDEO\_BUF); //#define NR\_VIDEO\_BUF (4)

uvc\_gadget\_stream(gadget->out, VIDEO\_STREAM\_ON);

break;

}

}

## VIDIOC\_REQBUFS

### uvc\_gadget\_reqbufs()

static int uvc\_gadget\_reqbufs(struct uvc\_gadget\_device \*dev, unsigned **int nbufs)**

{

struct v4l2\_requestbuffers rbuf;

memset(&rbuf, 0, sizeof(struct v4l2\_requestbuffers));

rbuf.count = nbufs;

rbuf.type = dev->type;

rbuf.memory = **V4L2\_MEMORY\_MMAP**;

ret = **v4l2\_request\_buffer**(dev->fd, &rbuf);

**dev->buf** = malloc(rbuf.count \* sizeof(void \*));

}

static inline int v4l2\_request\_buffer(int fd, struct v4l2\_requestbuffers \*rbuf)

{

ret = xioctl(fd, **VIDIOC\_REQBUFS**, rbuf);

}

### uvc\_v4l2\_do\_ioctl()

static long uvc\_v4l2\_do\_ioctl(struct file \*file, unsigned int cmd, void \*arg)

{

case **VIDIOC\_REQBUFS**:

{

struct v4l2\_requestbuffers \*rb = arg;

ret = **uvc\_alloc\_buffers**(&video->queue, rb);

break;

}

}

### uvc\_alloc\_buffers() //初始化vb2\_queue

static int uvc\_alloc\_buffers(**struct uvc\_video\_queue** \*queue, struct v4l2\_requestbuffers \*rb)

{

ret = **vb2\_reqbufs**(&queue->queue, rb);

**return ret ? ret : rb->count;**

}

int vb2\_reqbufs(**struct vb2\_queue** \*q, struct v4l2\_requestbuffers \*req)

{

int ret = \_\_verify\_memory\_type(q, req->memory, req->type);

return ret ? ret : **\_\_reqbufs**(q, req);

}

static int \_\_reqbufs(**struct vb2\_queue** \*q, struct v4l2\_requestbuffers \*req)

{

unsigned int num\_planes = 0;

num\_buffers = min\_t(unsigned int, req->count, VIDEO\_MAX\_FRAME);

memset(q->plane\_sizes, 0, sizeof(q->plane\_sizes));

memset(q->alloc\_ctx, 0, sizeof(q->alloc\_ctx));

**q->memory** = req->memory;

ret = **call\_qop**(q, **queue\_setup**, q, NULL, **&num\_buffers**, **&num\_planes**, **q->plane\_sizes**, q->alloc\_ctx);

ret = **\_\_vb2\_queue\_alloc**(q, req->memory, **num\_buffers**, **num\_planes**); // **num\_buffers =4, num\_planes =1**

allocated\_buffers = ret;

**q->num\_buffers** = allocated\_buffers;

**req->count** = allocated\_buffers;

}

### uvc\_queue\_setup()

static int uvc\_queue\_setup(**struct vb2\_queue** \*vq, const struct v4l2\_format \*fmt,

unsigned int \*nbuffers, unsigned int \*nplanes, unsigned int sizes[], void \*alloc\_ctxs[])

{

**struct uvc\_video\_queue** \*queue = vb2\_get\_drv\_priv(vq);

**struct uvc\_video** \*video = container\_of(queue, struct uvc\_video, queue);

if (\*nbuffers > UVC\_MAX\_VIDEO\_BUFFERS) \*nbuffers = UVC\_MAX\_VIDEO\_BUFFERS;

**\*nplanes = 1;**

**sizes[0] = video->imagesize;**

}

### \_\_vb2\_queue\_alloc() //初始化vb2\_buffer, v4l2\_buffer

static int \_\_vb2\_queue\_alloc(struct vb2\_queue \*q, enum v4l2\_memory memory, unsigned int num\_buffers, unsigned int num\_planes)

{

unsigned int buffer;

for (buffer = 0; buffer < num\_buffers; ++buffer) {

**struct vb2\_buffer** \*vb = kzalloc(q->buf\_struct\_size, GFP\_KERNEL);

**vb->state = VB2\_BUF\_STATE\_DEQUEUED;**

vb->vb2\_queue = q;

vb->**num\_planes = num\_planes;**

vb->v4l2\_buf.index = q->num\_buffers + buffer;

vb->v4l2\_buf.type = q->type;

vb->v4l2\_buf.memory = memory;

if (memory == V4L2\_MEMORY\_MMAP) {

ret = **\_\_vb2\_buf\_mem\_alloc**(vb);

ret = call\_qop(q, **buf\_init**, vb); //struct vb2\_ops uvc\_queue\_qops: not init

}

q->bufs[q->num\_buffers + buffer] = vb;

}

\_\_setup\_offsets(q, buffer);

**return buffer;**

}

### \_\_vb2\_buf\_mem\_alloc() //初始化vb2\_plane, v4l2\_plane

static int \_\_vb2\_buf\_mem\_alloc(struct vb2\_buffer \*vb)

{

struct vb2\_queue \*q = vb->vb2\_queue;

for (plane = 0; plane < vb->num\_planes; ++plane) {

unsigned long size = PAGE\_ALIGN(q->plane\_sizes[plane]); // **video->imagesize**

mem\_priv = call\_memop(q, **alloc**, q->alloc\_ctx[plane], size, q->gfp\_flags); //vb2\_vmalloc\_alloc()

**vb->planes[plane].mem\_priv** = mem\_priv;

**vb->v4l2\_planes[plane].length = q->plane\_sizes[plane];**

}

}

## VIDIOC\_QBUF:

### uvc\_gadget\_reqbufs()

static int uvc\_gadget\_reqbufs(struct uvc\_gadget\_device \*dev, unsigned **int nbufs)**

{

for (i = 0; i < rbuf.count; i++) {

memset(&dev->v4l2\_buf, 0, sizeof(struct v4l2\_buffer));

dev->v4l2\_buf.index = i;

dev->v4l2\_buf.type = dev->type;

dev->v4l2\_buf.memory = V4L2\_MEMORY\_MMAP;

ret = **v4l2\_query\_buffer**(dev->fd, &dev->v4l2\_buf);

**dev->buf[i]** = **mmap**(0, dev->v4l2\_buf.length,PROT\_READ | PROT\_WRITE, MAP\_SHARED, dev->fd, dev->v4l2\_buf.m.offset);

dev->v4l2\_buf.index = i;

dev->v4l2\_buf.type = dev->type;

dev->v4l2\_buf.memory = V4L2\_MEMORY\_MMAP;

ret = **v4l2\_queue\_buf(**dev->fd, &dev->v4l2\_buf);

}

**dev->bufsize** = dev->v4l2\_buf.length;

**dev->nbufs** = rbuf.count;

}

### v4l2\_query\_buffer()

static inline int v4l2\_query\_buffer(int fd, struct v4l2\_buffer \*buf)

{

ret = xioctl(fd, **VIDIOC\_QUERYBUF**, buf);

}

static void \_\_fill\_v4l2\_buffer(struct vb2\_buffer \*vb, struct v4l2\_buffer \*b)

{

struct vb2\_queue \*q = vb->vb2\_queue;

memcpy(b, &vb->v4l2\_buf, offsetof(struct v4l2\_buffer, m));

b->reserved2 = vb->v4l2\_buf.reserved2;

b->reserved = vb->v4l2\_buf.reserved;

if (V4L2\_TYPE\_IS\_MULTIPLANAR(q->type)) {

} else {

**b->length = vb->v4l2\_planes[0].length;**

**b->bytesused = vb->v4l2\_planes[0].bytesused;**

if ( q->memory == V4L2\_MEMORY\_MMAP) **b->m.offset = vb->v4l2\_planes[0].m.mem\_offset;**

else if (q->memory == V4L2\_MEMORY\_USERPTR) b->m.userptr = vb->v4l2\_planes[0].m.userptr;

else if (q->memory == V4L2\_MEMORY\_DMABUF) b->m.fd = vb->v4l2\_planes[0].m.fd;

}

}

### v4l2\_queue\_buf()

static inline int v4l2\_queue\_buf(int fd, struct v4l2\_buffer \*buf)

{

ret = xioctl(fd, **VIDIOC\_QBUF,** buf);

}

### uvc\_v4l2\_do\_ioctl()

static long uvc\_v4l2\_do\_ioctl(struct file \*file, unsigned int cmd, void \*arg)

{

case **VIDIOC\_QBUF:**

ret = **uvc\_queue\_buffer**(&video->queue, arg);

return **uvc\_video\_pump**(video);

break;

}

### uvc\_queue\_buffer():list\_add\_tail(&vb->queued\_entry, &q->queued\_list)

static int uvc\_queue\_buffer(struct uvc\_video\_queue \*queue, struct v4l2\_buffer \*buf)

{

ret = **vb2\_qbuf**(&queue->queue, buf);

queue->flags &= ~UVC\_QUEUE\_PAUSED;

}

int vb2\_qbuf(struct vb2\_queue \*q, struct v4l2\_buffer \*b)

{

struct vb2\_buffer \*vb = q->bufs[b->index];

ret = \_\_verify\_planes\_array(vb, b);

switch (vb->state) {

case VB2\_BUF\_STATE\_DEQUEUED: ret = **\_\_buf\_prepare**(vb, b);

case VB2\_BUF\_STATE\_PREPARED: break;

}

**list\_add\_tail(&vb->queued\_entry, &q->queued\_list);**

**vb->state = VB2\_BUF\_STATE\_QUEUED;**

if (q->streaming) **\_\_enqueue\_in\_driver**(vb);

**\_\_fill\_v4l2\_buffer**(vb, b);

}

### \_\_buf\_prepare()

static int \_\_buf\_prepare(struct vb2\_buffer \*vb, const struct v4l2\_buffer \*b)

{

struct vb2\_queue \*q = vb->vb2\_queue;

switch (q->memory) {

case V4L2\_MEMORY\_MMAP: ret = **\_\_qbuf\_mmap**(vb, b); break;

case V4L2\_MEMORY\_USERPTR: ret = \_\_qbuf\_userptr(vb, b); break;

case V4L2\_MEMORY\_DMABUF: ret = \_\_qbuf\_dmabuf(vb, b); break;

}

if (!ret) ret = call\_qop(q, **buf\_prepare**, vb);

vb->state = VB2\_BUF\_STATE\_PREPARED;

}

### \_\_qbuf\_mmap() //初始化 v4l2\_plane, v4l2\_buffer

static int \_\_qbuf\_mmap(struct vb2\_buffer \*vb, const struct v4l2\_buffer \*b)

{

**\_\_fill\_vb2\_buffer**(vb, b, vb->v4l2\_planes);

}

static void \_\_fill\_vb2\_buffer(struct vb2\_buffer \*vb, const struct v4l2\_buffer \*b, struct v4l2\_plane \*v4l2\_planes)

{

if (V4L2\_TYPE\_IS\_MULTIPLANAR(b->type)) {

} else {

if (V4L2\_TYPE\_IS\_OUTPUT(b->type)) { **v4l2\_planes[0].bytesused = b->bytesused; v4l2\_planes[0].data\_offset = 0;** }

if (V4L2\_MEMORY\_USERPTR) { v4l2\_planes[0].m.userptr = b->m.userptr; v4l2\_planes[0].length = b->length; }

if (V4L2\_MEMORY\_DMABUF) { v4l2\_planes[0].m.fd = b->m.fd; v4l2\_planes[0].length = b->length; v4l2\_planes[0].data\_offset = 0; }

}

**vb->v4l2\_buf.field = b->field;**

**vb->v4l2\_buf.timestamp = b->timestamp;**

vb->v4l2\_buf.flags = b->flags & ~V4L2\_BUFFER\_MASK\_FLAGS;

}

### uvc\_buffer\_prepare() //初始化uvc\_buffer

static int uvc\_buffer\_prepare(struct vb2\_buffer \*vb)

{

struct uvc\_video\_queue \*queue = vb2\_get\_drv\_priv(vb->vb2\_queue);

**struct uvc\_buffer** \*buf = container\_of(vb, struct uvc\_buffer, buf);

**buf->state = UVC\_BUF\_STATE\_QUEUED;**

**buf->mem = vb2\_plane\_vaddr(vb, 0);**

**buf->length = vb2\_plane\_size(vb, 0);**

**buf->bytesused = vb2\_get\_plane\_payload(vb, 0);**

}

void \*vb2\_plane\_vaddr(struct vb2\_buffer \*vb, unsigned int plane\_no)

{

return call\_memop(q, vaddr, **vb->planes[plane\_no].mem\_priv**); //vb2\_vmalloc\_vaddr()

}

static inline unsigned long vb2\_plane\_size(struct vb2\_buffer \*vb, unsigned int plane\_no)

{

return **vb->v4l2\_planes[plane\_no].length**;

}

static inline unsigned long vb2\_get\_plane\_payload(struct vb2\_buffer \*vb, unsigned int plane\_no)

{

return **vb->v4l2\_planes[plane\_no].bytesused**;

}

### \_\_enqueue\_in\_driver() // list\_add\_tail(&buf->queue, &queue->irqqueue)

static void \_\_enqueue\_in\_driver(struct vb2\_buffer \*vb)

{

vb->state = VB2\_BUF\_STATE\_ACTIVE;

atomic\_inc(&q->queued\_count);

for (plane = 0; plane < vb->num\_planes; ++plane)

call\_memop(q, prepare, vb->planes[plane].mem\_priv); // struct vb2\_mem\_ops vb2\_vmalloc\_memops: 没定义

q->ops->buf\_queue(vb); //uvc\_buffer\_queue()

}

static void uvc\_buffer\_queue(struct vb2\_buffer \*vb)

{

struct uvc\_video\_queue \*queue = vb2\_get\_drv\_priv(vb->vb2\_queue);

struct uvc\_buffer \*buf = container\_of(vb, struct uvc\_buffer, buf);

if (likely(!(queue->flags & UVC\_QUEUE\_DISCONNECTED))) {

**list\_add\_tail(&buf->queue, &queue->irqqueue);**

} else {

buf->state = UVC\_BUF\_STATE\_ERROR;

vb2\_buffer\_done(&buf->buf, VB2\_BUF\_STATE\_ERROR);

}

}

### \_\_fill\_v4l2\_buffer() //初始化v4l2\_buffer

static void \_\_fill\_v4l2\_buffer(struct vb2\_buffer \*vb, struct v4l2\_buffer \*b)

{

struct vb2\_queue \*q = vb->vb2\_queue;

memcpy(b, &vb->v4l2\_buf, offsetof(struct v4l2\_buffer, m));

b->reserved2 = vb->v4l2\_buf.reserved2;

b->reserved = vb->v4l2\_buf.reserved;

if (V4L2\_TYPE\_IS\_MULTIPLANAR(q->type)) {

} else {

**b->length = vb->v4l2\_planes[0].length;**

**b->bytesused = vb->v4l2\_planes[0].bytesused;**

if ( q->memory == V4L2\_MEMORY\_MMAP) **b->m.offset = vb->v4l2\_planes[0].m.mem\_offset;**

else if (q->memory == V4L2\_MEMORY\_USERPTR) b->m.userptr = vb->v4l2\_planes[0].m.userptr;

else if (q->memory == V4L2\_MEMORY\_DMABUF) b->m.fd = vb->v4l2\_planes[0].m.fd;

}

}

## VIDIOC\_STREAMON: list\_add\_tail(&video->req[i]->list, &video->req\_free)

### uvc\_gadget\_events\_process()

static void uvc\_gadget\_events\_process(struct uvc\_gadget \*gadget)

{

ret = v4l2\_dequeue\_event(gadget->out->fd, &v4l2\_event);

switch (v4l2\_event.type) {

case UVC\_EVENT\_STREAMON:

uvc\_gadget\_reqbufs(gadget->out, NR\_VIDEO\_BUF); //#define NR\_VIDEO\_BUF (4)

**uvc\_gadget\_stream**(gadget->out, **VIDEO\_STREAM\_ON**);

break;

}

}

static int uvc\_gadget\_stream(struct uvc\_gadget\_device \*dev, int enable)

{

if (enable) ret = **v4l2\_stream\_on**(dev->fd, dev->type);

}

static inline int v4l2\_stream\_on(int fd, int type)

{

ret = xioctl(fd, **VIDIOC\_STREAMON**, &type);

}

### uvc\_v4l2\_do\_ioctl()

uvc\_v4l2\_do\_ioctl(struct file \*file, unsigned int cmd, void \*arg)

{

**case VIDIOC\_STREAMON:**

{

ret = **uvc\_video\_enable**(video, 1);

uvc\_function\_setup\_continue(uvc);

uvc->state = UVC\_STATE\_STREAMING;

}

}

### uvc\_video\_enable()

static int uvc\_video\_enable(struct uvc\_video \*video, int enable)

{

ret = **uvc\_queue\_enable**(&video->queue, 1);

ret = **uvc\_video\_alloc\_requests**(video);

if (video->max\_payload\_size) {

video->encode = uvc\_video\_encode\_bulk;

video->payload\_size = 0;

} else

video->encode = **uvc\_video\_encode\_isoc**;

return **uvc\_video\_pump**(video);

}

### uvc\_queue\_enable()

static int uvc\_queue\_enable(struct uvc\_video\_queue \*queue, int enable)

{

if (enable) {

ret = **vb2\_streamon**(&queue->queue, queue->queue.type);

queue->sequence = 0;

**queue->buf\_used = 0;**

}

}

int vb2\_streamon(struct vb2\_queue \*q, enum v4l2\_buf\_type type)

{

call\_qop(q, start\_streaming, q, atomic\_read(&q->queued\_count)); // struct vb2\_ops uvc\_queue\_qops没有定义start\_streaming()

**q->streaming = 1;**

}

### uvc\_video\_alloc\_requests() // list\_add\_tail(&video->req[i]->list, &video->req\_free);

static int uvc\_video\_alloc\_requests(struct uvc\_video \*video)

{

req\_size = video->ep->maxpacket \* max\_t(unsigned int, video->ep->maxburst, 1) \* (video->ep->mult + 1);

for (i = 0; i < UVC\_NUM\_REQUESTS; ++i) { //#define UVC\_NUM\_REQUESTS 4

**video->req\_buffer[i] = kmalloc**(req\_size, GFP\_KERNEL);

video->req[i] = **usb\_ep\_alloc\_request**(video->ep, GFP\_KERNEL); //初始化usb\_request

video->req[i]->buf = video->req\_buffer[i];

video->req[i]->length = 0;

video->req[i]->complete = uvc\_video\_complete;

video->req[i]->context = video;

**list\_add\_tail(&video->req[i]->list, &video->req\_free);**

}

video->req\_size = req\_size;

}

static inline struct usb\_request \*usb\_ep\_alloc\_request(struct usb\_ep \*ep, gfp\_t gfp\_flags)

{

return ep->ops->alloc\_request(ep, gfp\_flags); // dwc3\_gadget\_ep\_alloc\_request()

}

static struct usb\_request \*dwc3\_gadget\_ep\_alloc\_request(struct usb\_ep \*ep, gfp\_t gfp\_flags)

{

struct dwc3\_ep \*dep = to\_dwc3\_ep(ep);

**struct dwc3\_request \*req = kzalloc**(sizeof(\*req), gfp\_flags);

req->epnum = dep->number;

req->dep = dep;

**return &req->request;**

}

## Video data encode

### uvc\_video\_pump()

static int **uvc\_video\_pump**(struct uvc\_video \*video)

{

while (1) {

if (list\_empty(&video->req\_free)) return 0;

struct usb\_request \*req = list\_first\_entry(&video->req\_free, struct usb\_request, list);

list\_del(&req->list);

struct uvc\_buffer \*buf = **uvc\_queue\_head**(&video->queue);

if (buf == NULL) { break; }

**video->encode**(req, video, buf); // uvc\_video\_encode\_isoc()

ret = **usb\_ep\_queue**(video->ep, req, GFP\_ATOMIC);

if (ret < 0) { usb\_ep\_set\_halt(video->ep); break; }

}

list\_add\_tail(&req->list, &video->req\_free);

}

static struct uvc\_buffer \***uvc\_queue\_head**(struct uvc\_video\_queue \*queue)

{

struct uvc\_buffer \*buf = NULL;

if (!list\_empty(&queue->irqqueue)) **buf = list\_first\_entry(&queue->irqqueue, struct uvc\_buffer,queue);**

else queue->flags |= UVC\_QUEUE\_PAUSED;

return buf;

}

### uvc\_video\_encode\_isoc()

static void uvc\_video\_encode\_isoc(**struct usb\_request** \*req, struct uvc\_video \*video, struct uvc\_buffer \*buf)

{

void \*mem = **req->buf;**

int len = video->req\_size; // len = 1024

ret = **uvc\_video\_encode\_header**(video, buf, mem, len); //ret = 2: header size

mem += ret; len -= ret;

ret = **uvc\_video\_encode\_data**(video, buf, mem, len);

len -= ret;

**req->length** = video->req\_size - len;

if (buf->bytesused == video->queue.buf\_used) {

video->queue.buf\_used = 0;

buf->state = UVC\_BUF\_STATE\_DONE;

**uvc\_queue\_next\_buffer**(&video->queue, buf);

**video->fid ^= UVC\_STREAM\_FID;**

}

}

### uvc\_video\_encode\_header()

**#define UVC\_STREAM\_EOH (1 << 7)**

**#define UVC\_STREAM\_ERR (1 << 6)**

**#define UVC\_STREAM\_STI (1 << 5)**

**#define UVC\_STREAM\_RES (1 << 4)**

**#define UVC\_STREAM\_SCR (1 << 3)**

**#define UVC\_STREAM\_PTS (1 << 2)**

**#define UVC\_STREAM\_EOF (1 << 1)**

**#define UVC\_STREAM\_FID (1 << 0)**

**//data: 0x02 0x80 or 0x02 0x81 : UVC\_STREAM\_EOH | UVC\_STREAM\_FID**

**//data: 0x02 0x82 or 0x02 0x83 : UVC\_STREAM\_EOH | UVC\_STREAM\_FID | UVC\_STREAM\_EOF**

static int **uvc\_video\_encode\_header**(struct uvc\_video \*video, struct uvc\_buffer \*buf, u8 \*data, int len)

{

data[0] = 2; //header size

data[1] = **UVC\_STREAM\_EOH** | **video->fid**; // video->fid ^= UVC\_STREAM\_FID;

if (buf->bytesused - video->queue.buf\_used <= len - 2) data[1] |= **UVC\_STREAM\_EOF**;

return 2;

}

### uvc\_video\_encode\_data()

static int **uvc\_video\_encode\_data**(struct uvc\_video \*video, struct uvc\_buffer \*buf, u8 \*data, int len)

{

struct uvc\_video\_queue \*queue = &video->queue;

void \*mem = buf->mem + queue->buf\_used;

nbytes = min((unsigned int)len, buf->bytesused - queue->buf\_used);

memcpy(data, mem, nbytes); // nbytes = 1024 - 2

**queue->buf\_used += nbytes;**

return nbytes;

}

### uvc\_queue\_next\_buffer()

static struct uvc\_buffer \*uvc\_queue\_next\_buffer(struct uvc\_video\_queue \*queue, struct uvc\_buffer \*buf)

{

struct uvc\_buffer \*nextbuf;

if ((queue->flags & UVC\_QUEUE\_DROP\_INCOMPLETE) && buf->length != buf->bytesused) {

buf->state = UVC\_BUF\_STATE\_QUEUED;

vb2\_set\_plane\_payload(&buf->buf, 0, 0);

return buf;

}

**list\_del(&buf->queue);**

if (!list\_empty(&queue->irqqueue)) **nextbuf = list\_first\_entry(&queue->irqqueue, struct uvc\_buffer, queue);**

else nextbuf = NULL;

buf->buf.v4l2\_buf.sequence = queue->sequence++;

do\_gettimeofday(&buf->buf.v4l2\_buf.timestamp);

vb2\_set\_plane\_payload(&buf->buf, 0, buf->bytesused); // vb->v4l2\_planes[plane\_no].bytesused = buf->bytesused

**vb2\_buffer\_done**(&buf->buf, VB2\_BUF\_STATE\_DONE);

return nextbuf;

}

### vb2\_buffer\_done()

void vb2\_buffer\_done(struct vb2\_buffer \*vb, enum vb2\_buffer\_state state)

{

struct vb2\_queue \*q = vb->vb2\_queue;

for (plane = 0; plane < vb->num\_planes; ++plane)

call\_memop(q, finish, vb->planes[plane].mem\_priv); // struct vb2\_mem\_ops vb2\_vmalloc\_memops: 没定义

vb->state = state;

**list\_add\_tail(&vb->done\_entry, &q->done\_list);**

atomic\_dec(&q->queued\_count);

wake\_up(&q->done\_wq);

}

### usb\_ep\_queue()

static inline int usb\_ep\_queue(struct usb\_ep \*ep, struct usb\_request \*req, gfp\_t gfp\_flags)

{

return ep->ops->queue(ep, req, gfp\_flags); // dwc3\_gadget\_ep\_queue()

}

## USB data queue

### dwc3\_gadget\_ep\_queue() // list\_add\_tail(&req->list, &dep->request\_list);

static int dwc3\_gadget\_ep\_queue(struct usb\_ep \*ep, struct usb\_request \*request, gfp\_t gfp\_flags)

{

struct dwc3\_request \*req = to\_dwc3\_request(request);

struct dwc3\_ep \*dep = to\_dwc3\_ep(ep);

ret = **\_\_dwc3\_gadget\_ep\_queue**(dep, req);

}

static int **\_\_dwc3\_gadget\_ep\_queue**(struct dwc3\_ep \*dep, struct dwc3\_request \*req)

{

struct dwc3 \*dwc = dep->dwc;

**req->request.actual = 0;**

req->request.status = -EINPROGRESS;

req->direction = dep->direction;

req->epnum = dep->number;

ret = **usb\_gadget\_map\_request**(&dwc->gadget, &req->request,dep->direction);

**list\_add\_tail(&req->list, &dep->request\_list);**

if (dep->flags & DWC3\_EP\_PENDING\_REQUEST) {

if (usb\_endpoint\_xfer\_isoc(dep->endpoint.desc)) {

if (list\_empty(&dep->req\_queued)) {

dwc3\_stop\_active\_transfer(dwc, dep->number);

dep->flags = DWC3\_EP\_ENABLED;

}

return 0;

}

ret = \_\_dwc3\_gadget\_kick\_transfer(dep, 0, true);

return ret;

}

if (usb\_endpoint\_xfer\_isoc(dep->endpoint.desc) &&(dep->**flags & DWC3\_EP\_BUSY**) &&!(dep->**flags & DWC3\_EP\_MISSED\_ISOC**)) {

ret = **\_\_dwc3\_gadget\_kick\_transfer**(dep, dep->resource\_index,false);

return ret;

}

return 0;

}

### usb\_gadget\_map\_request()

int usb\_gadget\_map\_request(struct usb\_gadget \*gadget, struct usb\_request \*req, int is\_in)

{

if (req->num\_sgs) {

mapped = dma\_map\_sg(&gadget->dev, req->sg, req->num\_sgs, is\_in ? DMA\_TO\_DEVICE : DMA\_FROM\_DEVICE);

req->num\_mapped\_sgs = mapped;

} else {

req->dma = dma\_map\_single(&gadget->dev, req->buf, req->length, is\_in ? DMA\_TO\_DEVICE : DMA\_FROM\_DEVICE);

}

}

### \_\_dwc3\_gadget\_kick\_transfer() // start\_new = 0

static int \_\_dwc3\_gadget\_kick\_transfer(struct dwc3\_ep \*dep, u16 cmd\_param, int start\_new)

{

struct dwc3\_request \*req;

dep->**flags &= ~DWC3\_EP\_PENDING\_REQUEST**;

if (start\_new) {

if (list\_empty(&dep->req\_queued)) dwc3\_prepare\_trbs(dep, start\_new);

req = next\_request(&dep->req\_queued);

} else {

**dwc3\_prepare\_trbs**(dep, start\_new);

req = **next\_request**(&dep->req\_queued);

}

struct dwc3\_gadget\_ep\_cmd\_params params;

memset(&params, 0, sizeof(params));

if (start\_new) {

params.param0 = upper\_32\_bits(req->trb\_dma);

params.param1 = lower\_32\_bits(req->trb\_dma);

cmd = DWC3\_DEPCMD\_STARTTRANSFER;

} else { cmd = **DWC3\_DEPCMD\_UPDATETRANSFER**; }

cmd |= DWC3\_DEPCMD\_PARAM(cmd\_param);

ret = **dwc3\_send\_gadget\_ep\_cmd**(dwc, dep->number, cmd, &params);

dep->**flags |= DWC3\_EP\_BUSY**;

if (start\_new) {

dep->resource\_index = dwc3\_gadget\_ep\_get\_transfer\_index(dwc,dep->number);

WARN\_ON\_ONCE(!dep->resource\_index);

}

}

### dwc3\_prepare\_trbs()

static void dwc3\_prepare\_trbs(struct dwc3\_ep \*dep, bool starting)

{

struct dwc3\_request \*req, \*n;

trbs\_left = (dep->busy\_slot - dep->free\_slot) & DWC3\_TRB\_MASK;

if (!trbs\_left) {

trbs\_left = DWC3\_TRB\_NUM;

if (usb\_endpoint\_xfer\_isoc(dep->endpoint.desc)) { dep->busy\_slot = 1; dep->free\_slot = 1;}

else { dep->busy\_slot = 0; dep->free\_slot = 0; }

}

list\_for\_each\_entry\_safe(req, n, &**dep->request\_list**, list) {

last\_one = false;

dma\_addr\_t dma = **req->request.dma**;

unsigned length = **req->request.length;**

trbs\_left--; if (!trbs\_left) last\_one = 1;

**if (list\_is\_last(&req->list, &dep->request\_list)) last\_one = 1;**

**dwc3\_prepare\_one\_trb**(dep, req, **dma, length, last\_one**, false, 0);

if (last\_one) break;

}

}

### dwc3\_prepare\_one\_trb() //初始化struct dwc3\_trb

#### list\_move\_tail(&req->list, &dep->req\_queued)

static void dwc3\_prepare\_one\_trb(struct dwc3\_ep \*dep, struct dwc3\_request \*req, dma\_addr\_t dma,

unsigned length, unsigned last, unsigned chain, unsigned node)

{

**struct dwc3\_trb** \*trb = &dep->trb\_pool[dep->free\_slot & DWC3\_TRB\_MASK];

if (!req->trb) {

**dwc3\_gadget\_move\_request\_queued(**req);

req->trb = trb;

req->trb\_dma = dwc3\_trb\_dma\_offset(dep, trb);

req->start\_slot = dep->free\_slot & DWC3\_TRB\_MASK;

}

dep->free\_slot++; //free\_slot=[1, 100]

if (((dep->free\_slot & DWC3\_TRB\_MASK) == DWC3\_TRB\_NUM - 1) && usb\_endpoint\_xfer\_isoc(dep->endpoint.desc))

dep->free\_slot++;

**trb->size = DWC3\_TRB\_SIZE\_LENGTH(length);**

**trb->bpl = lower\_32\_bits(dma);**

**trb->bph = upper\_32\_bits(dma);**

trb->ctrl |= DWC3\_TRB\_CTRL\_HWO;

}

static inline void **dwc3\_gadget\_move\_request\_queued**(struct dwc3\_request \*req)

{

**req->queued = true**;

**list\_move\_tail(&req->list, &dep->req\_queued);**

}

## USB Data dequeue

### dwc3\_endpoint\_interrupt()

static void dwc3\_endpoint\_interrupt(struct dwc3 \*dwc, const struct dwc3\_event\_depevt \*event)

{

struct dwc3\_ep \*dep = dwc->eps[epnum];

switch (event->endpoint\_event) {

case **DWC3\_DEPEVT\_XFERINPROGRESS**:

**dwc3\_endpoint\_transfer\_complete**(dwc, dep, event, 0);

break;

}

### dwc3\_endpoint\_transfer\_complete()

void dwc3\_endpoint\_transfer\_complete(struct dwc3 \*dwc,struct dwc3\_ep \*dep, const struct dwc3\_event\_depevt \*event,int start\_new)

{

clean\_busy = **dwc3\_cleanup\_done\_reqs**(dwc, dep, event, status);

if (clean\_busy)

dep->**flags &= ~DWC3\_EP\_BUSY**;

}

### dwc3\_cleanup\_done\_reqs()

static int dwc3\_cleanup\_done\_reqs(struct dwc3 \*dwc, struct dwc3\_ep \*dep, const struct dwc3\_event\_depevt \*event, int status)

{

do {

**struct dwc3\_request** \*req = **next\_request**(&dep->req\_queued);

i = 0;

do {

slot = req->start\_slot + i;

if ((slot == DWC3\_TRB\_NUM - 1) && usb\_endpoint\_xfer\_isoc(dep->endpoint.desc)) slot++;

slot %= DWC3\_TRB\_NUM;

**struct dwc3\_trb** \*trb = &**dep->trb\_pool**[slot];

ret = **\_\_dwc3\_cleanup\_done\_trbs**(dwc, dep, req, trb, event, status);

if (ret) break;

}while (++i < req->request.num\_mapped\_sgs);

**dwc3\_gadget\_giveback**(dep, req, status);

if (ret) break;

} while (1);

if (usb\_endpoint\_xfer\_isoc(dep->endpoint.desc) && **list\_empty(&dep->req\_queued)**) {

if (**list\_empty(&dep->request\_list**)) {

dep->**flags = DWC3\_EP\_PENDING\_REQUEST**;

} else {

dwc3\_**stop\_active**\_transfer(dwc, dep->number);

dep->**flags = DWC3\_EP\_ENABLED;**

}

return 1;

}

return 1;

}

### \_\_dwc3\_cleanup\_done\_trbs()

static int \_\_dwc3\_cleanup\_done\_trbs(struct dwc3 \*dwc, struct dwc3\_ep \*dep,

struct dwc3\_request \*req, struct dwc3\_trb \*trb, const struct dwc3\_event\_depevt \*event, int status)

{

**count = trb->size & DWC3\_TRB\_SIZE\_MASK;**

if (dep->direction) {

if (count) {

trb\_status = DWC3\_TRB\_SIZE\_TRBSTS(trb->size);

if (trb\_status == DWC3\_TRBSTS\_MISSED\_ISOC) { dep->**flags |= DWC3\_EP\_MISSED\_ISOC**; }

else { **status = -ECONNRESET;**  }

} else { dep->flags &= ~DWC3\_EP\_MISSED\_ISOC; }

} else {

if (count && (event->status & DEPEVT\_STATUS\_SHORT)) s\_pkt = 1;

}

**req->request.actual += req->request.length - count**;

if (s\_pkt) return 1;

if ((event->status & DEPEVT\_STATUS\_LST) && (trb->ctrl & (DWC3\_TRB\_CTRL\_LST | DWC3\_TRB\_CTRL\_HWO))) return 1;

if ((event->status & DEPEVT\_STATUS\_IOC) && (trb->ctrl & DWC3\_TRB\_CTRL\_IOC)) return 1;

**return 0;**

}

### dwc3\_gadget\_giveback()

#### list\_del(&req->list);

void dwc3\_gadget\_giveback(struct dwc3\_ep \*dep, struct dwc3\_request \*req, int status)

{

if (**req->queued**) {

i = 0;

do {

**dep->busy\_slot++**;

if (((dep->busy\_slot & DWC3\_TRB\_MASK) == DWC3\_TRB\_NUM- 1) && usb\_endpoint\_xfer\_isoc(dep->endpoint.desc))

dep->busy\_slot++;

} while(++i < req->request.num\_mapped\_sgs); // i=0

**req->queued = false;**

}

**list\_del(&req->list);**

req->trb = NULL;

if (req->request.status == -EINPROGRESS) req->request.status = status;

if (dwc->ep0\_bounced && dep->number == 0) dwc->ep0\_bounced = false;

else **usb\_gadget\_unmap\_request**(&dwc->gadget, &req->request, req->direction);

**req->request.complete**(&dep->endpoint, &req->request);

}

# Buffer Management

## Buffer流程

### vb2\_buffer

用户填充vb2\_buffer数据后：

1. 设置v4l2\_planes[0].bytesused = b->bytesused，vb2\_buffer的实际大小
2. 将vb2\_buffer. queued \_entry挂到vb2\_queue. queued \_list中
3. 设置vb2\_buffer状态为VB2\_BUF\_STATE\_QUEUED
4. 设置uvc\_buffer的状态为UVC\_BUF\_STATE\_QUEUED
5. 设置uvc\_buffer->mem，uvc\_buffer->length，uvc\_buffer->byteused

vb2\_buffer中的数据处理中:

1. 设置vb2\_buffer状态为VB2\_BUF\_STATE\_ACTIVE
2. 将uvc\_buffer.queue挂到uvc\_video\_queue.irqqueue中

vb2\_buffer中的数据处理后:

1. 将uvc\_buffer.queue从uvc\_video\_queue.irqqueue中移除
2. 设置vb2\_buffer状态为VB2\_BUF\_STATE\_DONE
3. 将vb2\_buffer.done\_entry挂到vb2\_queue.done\_list中
4. 触发vb2\_queue.done\_wq中断

用户获取vb2\_buffer:

1. vb2\_buffer.done\_entry从vb2\_queue.done\_list中移除
2. vb2\_buffer.queued\_entry从vb2\_queue.queued\_list中移除
3. 设置vb2\_buffer状态为VB2\_BUF\_STATE\_DEQUEUED

### uvc\_buffer

获取uvc\_buffer的数据

1. 从uvc\_video\_queue.irqqueue中获取uvc\_buffe.queue
2. usb\_request.list从uvc\_video.req\_free中移除
3. 将uvc\_buffe中的数据拷贝到usb\_request中

## Video init

### struct uvc\_buffer buf

**struct uvc\_buffer** {

**struct vb2\_buffer buf;**

**struct list\_head queue;** //list\_add\_tail(&buf->queue, &queue->irqqueue)

enum uvc\_buffer\_state **state**; //UVC\_BUF\_STATE\_QUEUED

void \***mem**; //vb->planes[plane\_no].mem\_priv

unsigned int **length**; // vb->v4l2\_planes[plane\_no].length

unsigned int **bytesused**; //vb->v4l2\_planes[plane\_no].bytesused;

};

### struct uvc\_video video

**struct uvc\_video** {

**struct usb\_ep \*ep;**

u8 bpp;

u32 fcc;

unsigned int **width**;

unsigned int **height**;

unsigned int **imagesize**;

unsigned int **req\_size**;

// req\_size = video->ep->maxpacket \* max\_t(unsigned int, video->ep->maxburst, 1) \* (video->ep->mult + 1)

**struct usb\_request \*req[UVC\_NUM\_REQUESTS];** //UVC\_NUM\_REQUESTS = 4

\_\_u8 **\*req\_buffer[UVC\_NUM\_REQUESTS]**; //kmalloc(req\_size)

**struct list\_head req\_free;**  //list\_add\_tail(&video->req[i]->list, &video->req\_free);

spinlock\_t req\_lock;

void (\*encode)(); //uvc\_video\_encode\_isoc()

\_\_u32 payload\_size;

\_\_u32 max\_payload\_size;

**struct uvc\_video\_queue queue;**

unsigned int fid;

};

### struct uvc\_video\_queue queue

**struct uvc\_video\_queue** {

**struct vb2\_queue queue;**

struct mutex mutex;

unsigned int flags;

\_\_u32 **sequence**; //0

unsigned int **buf\_used**; //0

spinlock\_t irqlock;

**struct list\_head irqqueue;**  //list\_add\_tail(&buf->queue, &queue->irqqueue)

};

### struct vb2\_queue q

**struct vb2\_queue** {

enum v4l2\_buf\_type type;

unsigned int io\_modes;

unsigned int io\_flags;

struct mutex \*lock;

struct v4l2\_fh \*owner;

const struct vb2\_ops \*ops;

const struct vb2\_mem\_ops \*mem\_ops;

void \*drv\_priv;

unsigned int buf\_struct\_size;

u32 timestamp\_type;

gfp\_t gfp\_flags;

enum v4l2\_memory memory;

**struct vb2\_buffer \*bufs[VIDEO\_MAX\_FRAME];** //vb[req->count]

unsigned int **num\_buffers;** //req->count

**struct list\_head queued\_list;** //list\_add\_tail(&vb->queued\_entry, &q->queued\_list);

atomic\_t queued\_count;

**struct list\_head done\_list;**

spinlock\_t done\_lock;

wait\_queue\_head\_t done\_wq;

void \*alloc\_ctx[VIDEO\_MAX\_PLANES];

unsigned int **plane\_sizes[VIDEO\_MAX\_PLANES];** //sizes[0] = video->imagesize

unsigned int **streaming**:1; //1

struct vb2\_fileio\_data \*fileio;

};

### struct vb2\_buffer vb

**struct vb2\_buffer** {

**struct v4l2\_buffer v4l2\_buf;**

**struct v4l2\_plane v4l2\_planes[VIDEO\_MAX\_PLANES];**

**struct vb2\_queue \*vb2\_queue;** //q

unsigned int num\_planes; //1

enum vb2\_buffer\_state state; //VB2\_BUF\_STATE\_DEQUEUED 🡪 VB2\_BUF\_STATE\_PREPARED 🡪

VB2\_BUF\_STATE\_ACTIVE 🡨 VB2\_BUF\_STATE\_QUEUED 🡨

**struct list\_head queued\_entry;** //list\_add\_tail(&vb->queued\_entry, &q->queued\_list);

**struct list\_head done\_entry;**

**struct vb2\_plane planes[VIDEO\_MAX\_PLANES];** //vb->planes[plane].mem\_priv = mem\_priv

};

### struct v4l2\_buffer b

**struct v4l2\_buffer** {

\_\_u32 index;

\_\_u32 type;

\_\_u32 bytesused; //vb->v4l2\_planes[0].bytesused

\_\_u32 flags;

\_\_u32 field;

struct timeval timestamp;

struct v4l2\_timecode timecode;

\_\_u32 sequence;

\_\_u32 memory;

union {

\_\_u32 offset; //vb->v4l2\_planes[0].m.mem\_offset

unsigned long userptr;

struct v4l2\_plane \*planes;

\_\_s32 fd;

} m;

\_\_u32 **length**; //vb->v4l2\_planes[0].length

\_\_u32 reserved2;

\_\_u32 reserved;

};

### struct v4l2\_plane v4l2\_planes

**struct v4l2\_plane** {

\_\_u32 bytesused; //b->bytesused

\_\_u32 **length**; //q->plane\_sizes[plane]

union {

\_\_u32 mem\_offset;

unsigned long userptr;

\_\_s32 fd;

} m;

\_\_u32 data\_offset; //0

\_\_u32 reserved[11];

};

### struct vb2\_plane

**struct vb2\_plane** {

void \***mem\_priv**; //malloc(q->plane\_sizes[plane])

struct dma\_buf \*dbuf;

unsigned int dbuf\_mapped;

};

### struct usb\_request request

**struct usb\_request** {

void \*buf; //video->req\_buffer[i]

unsigned length; //0

dma\_addr\_t dma; // dma\_map\_single(req->buf, req->length, DMA\_FROM\_DEVICE)

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;

void (\*complete)(); //uvc\_video\_complete()

void \*context; // struct uvc\_video video

**struct list\_head list;** //list\_add\_tail(&video->req[i]->list, &video->req\_free);

int status;

unsigned **actual**; //0

};

### struct dwc3\_request req

**struct dwc3\_request** {

**struct usb\_request request;**

**struct list\_head list; //list\_add\_tail(&req->list, &dep->request\_list);**

**//list\_move\_tail(&req->list, &dep->req\_queued)**

**struct dwc3\_ep \*dep;**

u32 start\_slot; //dep->free\_slot

u8 **epnum**; //dep->number

**struct dwc3\_trb \*trb; // dep->trb\_pool[dep->free\_slot]**

dma\_addr\_t trb\_dma;

unsigned **direction**:1; // dep->direction

unsigned mapped:1;

unsigned queued:1;

};

### struct dwc3\_ep dep

**struct dwc3\_ep** {

**struct usb\_ep endpoint;**

**struct list\_head request\_list; //** **list\_add\_tail(&req->list, &dep->request\_list);**

**struct list\_head req\_queued; //list\_move\_tail(&req->list, &dep->req\_queued)**

**struct dwc3\_trb \*trb\_pool; //trb = &dep->trb\_pool[dep->free\_slot]**

dma\_addr\_t trb\_pool\_dma; // **dep->trb\_pool的dma对应的handle**

u32 **free\_slot**; //free\_slot++

u32 **busy\_slot**;

const struct usb\_ss\_ep\_comp\_descriptor \*comp\_desc;

**struct dwc3 \*dwc;**

unsigned flags;

unsigned current\_trb;

u8 **number**;

u8 type;

u8 resource\_index;

u32 interval;

char name[20];

unsigned **direction**:1;

unsigned stream\_capable:1;

};

#define DWC3\_TRB\_NUM (32)

trb\_pool = dma\_alloc\_coherent(dwc->dev,sizeof(struct dwc3\_trb) \* DWC3\_TRB\_NUM,&dep->trb\_pool\_dma, GFP\_KERNEL);

### struct dwc3\_trb trb

trb: transfer request block

struct dwc3\_trb {

u32 bpl; //DW0-3 // lower\_32\_bits(req->request.dma)

u32 bph; //DW4-7 // upper\_32\_bits(req->request.dma)

u32 size; //DW8-B // DWC3\_TRB\_SIZE\_LENGTH(req->request.length)

u32 ctrl; //DWC-F

} \_\_packed;

## Video qbuf

### Buffer init

INPUT : b->bytesused

============================================================

QBUF:

============================================================

**v4l2\_planes[0].bytesused** = b->bytesused; //imgsize

v4l2\_planes[0].data\_offset = 0;

**list\_add\_tail(&vb->queued\_entry, &q->queued\_list);**

**vb->state** = VB2\_BUF\_STATE\_ACTIVE;

**list\_add\_tail(&buf->queue, &queue->irqqueue);**

### Buffer move

============================================================

buf->bytesused = imgsize = 320x240\*2 = 153600 : (YUYV)

queue->buf\_used = 1022\*n = 0: (n=0~149)

============================================================

**struct usb\_request \*req** = list\_first\_entry(&video->req\_free, struct usb\_request, list);

**list\_del(&req->list);**

**struct uvc\_buffer \*buf** = list\_first\_entry(&queue->irqqueue, struct uvc\_buffer, queue);

data = **req->buf**;

len = **video->req\_size**; //1024

**data[0] = 0x02;**

**data[1] = UVC\_STREAM\_EOH | video->fid;**

data += 2;

len -= 2; //1022

mem = buf->mem + queue->buf\_used;

nbytes = min((unsigned int)len, buf->bytesused - queue->buf\_used); //1022

**memcpy(data, mem, nbytes);**

**queue->buf\_used** += nbytes; // += 1022

len -= nbytes; //0

**req->length** = video->req\_size - len; //1024

**usb\_ep\_queue(video->ep, req, GFP\_ATOMIC);**

============================================================

buf->bytesused = imgsize = 320x240\*2 = 153600 : (YUYV)

queue->buf\_used = 1022\*n = 153300: (n=150)

============================================================

**struct usb\_request \*req** = list\_first\_entry(&video->req\_free, struct usb\_request, list);

**list\_del(&req->list);**

**struct uvc\_buffer \*buf** = list\_first\_entry(&queue->irqqueue, struct uvc\_buffer, queue);

data = **req->buf**;

len = video->req\_size; //1024

**data[0] = 0x02;**

**data[1] = UVC\_STREAM\_EOH | video->fid | UVC\_STREAM\_EOF;**

data += 2;

len -= 2; //1022

mem = buf->mem + queue->buf\_used;

nbytes = min((unsigned int)len, buf->bytesused - queue->buf\_used); //153600-153300=300

**memcpy(data, mem, nbytes);**

**queue->buf\_used** += nbytes; // += 300

len -= nbytes; //1022-300 = 722

**req->length** = video->req\_size - len; //1024 - 722 = 302

video->queue.buf\_used = 0;

**buf->state** = UVC\_BUF\_STATE\_DONE;

**video->fid** ^= UVC\_STREAM\_FID;

**list\_del(&buf->queue);**

**vb->state** = VB2\_BUF\_STATE\_DONE

**list\_add\_tail(&vb->done\_entry, &q->done\_list);**

**wake\_up(&q->done\_wq);**

**usb\_ep\_queue(video->ep, req, GFP\_ATOMIC);**

## Video dqbuf

wait\_event\_interruptible(**q->done\_wq,** !list\_empty(&q->done\_list) || !q->streaming)

**struct vb2\_buffer \*vb** = list\_first\_entry(&q->done\_list, struct vb2\_buffer, done\_entry);

**list\_del(&(\*vb)->done\_entry);**

**list\_del(&vb->queued\_entry);**

**vb->state** = VB2\_BUF\_STATE\_DEQUEUED;

struct dwc3\_trb {

u32 bpl;

u32 bph;

u32 size;

u32 ctrl;

} \_\_packed;

# End