## 解析

drivers/of/platform.c

arch\_initcall\_sync(of\_platform\_default\_populate\_init);

of\_platform\_default\_populate\_init

{

of\_platform\_default\_populate(NULL,NULL,NULL);

}

of\_platform\_default\_populate(NULL,NULL,NULL)

{

struct device\_node \*root = NULL;

struct of\_dev\_auxdata \*lookup = NULL;

struct device \*parent = NULL;

of\_platform\_populate(root,of\_default\_bus\_match\_table,lookup,parent);

}

of\_platform\_populate(root,of\_default\_bus\_match\_table,lookup,parent)

{

struct device\_node \*child;

int rc = 0;

root = root?of\_node\_get(root):of\_find\_node\_by\_path(“/”);

for\_each\_child\_of\_node(root,child) {

rc = of\_platform\_bus\_create(child,of\_default\_bus\_match\_table,NULL,NULL,true);

}

}

of\_platform\_bus\_create(child,NULL,of\_default\_bus\_match\_table,NULL,NULL,true)

{

struct platform\_device \*dev;

struct device\_node \*bus = child;

const char \*bus\_id = NULL;

void \*platform\_data = NULL;

struct of\_device\_id \*matches = of\_default\_bus\_match\_table;

dev = of\_platform\_device\_create\_pdate(bus,NULL,NULL,NULL);

}

of\_platform\_device\_create\_pdate(bus,NULL,NULL,NULL)

{

struct device\_node \*np = bus;

char \*bus\_id = NULL;

void \*platform\_data = NULL;

struct device \*parent = NULL;

struct platform\_device \*dev;

dev = of\_device\_alloc(np,bus\_id,parent);

}

of\_device\_alloc(np,bus\_id,parent)

{

int num\_irq;

struct resource \*res;

int rc;

num\_irq = of\_irq\_count(nq);

if(num\_irq) {

res = kcalloc(num\_irq,sizeof(\*res),GFP\_KERNEL);

dev->num\_resources = num\_reg;

dev->resource = res;

for(i = 0;i<num\_reg;i++,res++) {

if(of\_irq\_to\_resource\_table(np,res,num\_irq)!=num\_irq)

{

}

}

}

}

of\_irq\_to\_resource\_table(np,res,num\_irq) {

struct device\_node \*dev = np;

struct resource \*res = res;

int nr\_irqs = num\_irq;

int i;

for(i =0;i<nr\_irqs;i++,res++) {

if(of\_irq\_to\_resource(dev,i,res) <=0)

break;

}

return i;

}

of\_irq\_to\_resource(dev,i,res)

{

int irq = of\_irq\_get(dev,index);

}

of\_irq\_get(dev,index)

{

int rc;

struct irq\_domain \*domain;

struct of\_phandle\_args oirq;

rc = of\_irq\_parse\_one(dev,index,&oirq);

domain = irq\_find\_host(oirq.np);

return irq\_create\_of\_mapping(&oirq);

}

irq\_create\_of\_mapping(&oirq)

{

struct irq\_fwspec fwspec;

struct of\_phandle\_args \*irq\_data = &oirq;

of\_phandle\_args\_to\_fwspec(irq\_data->np,irq\_data->args

,irq\_data->args\_count,&fwspec);

return irq\_create\_fwspec\_mapping(&fwspec);

}

of\_phandle\_args\_to\_fwspec(irq\_data->np,irq\_data->args

,irq\_data->args\_count,&fwspec)

{

int i;

fwspec->fwnode = np ?&np->fwnode:NULL;

fwspec->param\_count = count;

for(i=0;i<count;i++)

fwspec->param[i] =args[i];

}

irq\_create\_fwspec\_mapping(&fwspec)

{

struct irq\_data \*irq\_data;

struct irq\_domain \*domain;

irq\_hw\_number\_t hwirq;

unsigned int type = IRQ\_TYPE\_NONE;

int virq;

if(fwspec->fwnode) {

domain = irq\_find\_matching\_fwspec(fwspec,DOMAIN\_BUS\_WIRED);

}

if(irq\_domain\_translate(domain,fwspec,&hwirq,&type))

return 0;

virq = irq\_find\_mapping(domain,hwirq);

if(virq)

{

return 0;

}

if(irq\_domain\_is\_hierarchy(domain)) {

virq = irq\_domain\_alloc\_irqs(domain,1,NUMA\_NO\_NODE,fwspec);

}

irq\_data = irq\_get\_irq\_data(virq);

irqd\_set\_trigger\_type(irq\_data,type);

return virq;

}

irq\_domain\_translate(domain,fwspec,&hwirq,&type)

{

struct irq\_domain \*d = domain;

if(d->ops->translate)

d->ops->translate(d,fwspec,hwirq,type) =

/\*

irq-gic.c

struct irq\_domain\_ops gic\_irq\_domain\_hierarchy\_ops

\*/

gic\_irq\_domain\_translate(d,fwspec,hwirq,type)

{

if(is\_of\_node(fwspec->fwnode))

{

if(fwspec->param\_count <3)

return –EINVAL;

\*hwirq = fwspec->param[1] +16;

if(!fwspec->param[0])

\*hwirq+=16;

\*type = fwspec->param[2] &IRQ\_TYPE\_SENSE\_MASK;

return 0;

}

}

}

struct irq\_domain {

unsigned int revmap\_direct\_max\_irq;

};

irq\_find\_mapping(domain,hwirq)

{

struct irq\_data \*data;

/\*

revmap\_direct\_max\_irq的值为0

\*/

if(hwirq < domain->revmap\_direct\_max\_irq)

{

}

/\*

revmap\_size值为96，走这里

\*/

if(hwirq < domain->revmap\_size)

return domain->linear\_revmap[hwirq];

}

irq\_domain\_alloc\_irqs(domain,1,NUMA\_NO\_NODE,fwspec)

->\_\_irq\_domain\_alloc\_irqs(domain,-1,1,NUMA\_NO\_NODE,fwspec,false,NULL)

{

int virq;

virq = irq\_domain\_alloc\_descs(-1,1,0,NUMA\_NO\_NODE,NULL);

irq\_domain\_alloc\_irq\_data(domain,virq,1);

irq\_domain\_alloc\_irqs\_hierarchy(domain,virq,1,arg);

}

irq\_domain\_alloc\_descs(-1,1,0,NUMA\_NO\_NODE,NULL)

{

int virq = -1;

unsigned int hint;

if(virq >=0)

{

}else {

hint = hwirq % nr\_irqs;

virq = \_\_irq\_alloc\_descs(-1,hint,1,NUMA\_NO\_NODE,THIS\_MODULE,NULL);

}

}

\_\_irq\_alloc\_descs(-1,hint,1,NUMA\_NO\_NODE,THIS\_MODULE,NULL)

{

unsigned int from = hint;

int start;

unsigned int cnt =1;

start = bitmap\_find\_next\_zero\_area(allocated\_irqs,IRQ\_BITMAP\_BITS,from,1,0);

ret = alloc\_descs(start,cnt,NUMA\_NO\_NODE,NULL,THIS\_MODULE);

}

/\*

CONFIG\_SPARSE\_IRQ默认是开启的，所以alloc\_descs会走SPARES\_IRQ分支

\*/

alloc\_descs(start,cnt,NUMA\_NO\_NODE,NULL,THIS\_MODULE)

{

struct irq\_desc \*desc;

unsigned flags =0;

desc = alloc\_desc(start,NUMA\_NO\_NODE,flags,NULL,THIS\_MODULE);

radix\_tree\_insert(&irq\_desc\_tree,irq,desc);

bitmap\_set(allocated\_irqs,start,cnt);

}

alloc\_desc(start,NUMA\_NO\_NODE,flags,NULL,THIS\_MODULE)

{

struct irq\_desc \*desc;

desc = kzalloc\_node(sizeof(\*desc),GFP\_KERNEL,node);

desc\_set\_defaults(start,desc,NUMA\_NO\_NODE,NULL,THIS\_MODULE);

irqd\_set(&desc->irq\_data,flags);

return desc;

}

irq\_domain\_alloc\_irqs\_hierarchy(domain,virq,1,arg)->

gic\_irq\_domain\_alloc(domain,virq,1,arg)

{

irq\_hw\_number\_t hwirq;

struct irq\_fwspec \*fwspec = arg;

ret = gic\_irq\_domain\_translate(domain,fwspec,&hwirq,&type);

ret = gic\_irq\_domain\_map(domain,virq,hwirq);

}

gic\_irq\_domain\_map(domain,virq,hwirq)

{

struct gic\_chip\_data \*gic = domain->host\_data;

if(hw<32) {

}else {

irq\_domain\_set\_info(domain,virq,hwirq,&gic->chip,gic,handle\_fasteoi\_irq,NULL,NULL);

}  
}

irq\_domain\_set\_info(domain,virq,hwirq,&gic->chip,gic,handle\_fasteoi\_irq,NULL,NULL)

{

irq\_domain\_set\_hwirq\_and\_chip(domain,virq,hwirq,&gic->chip,gic);

\_\_irq\_set\_handler(virq,handler,0,NULL);

}

irq\_domain\_set\_hwirq\_and\_chip(domain,virq,hwirq,&gic->chip,gic)

{

struct irq\_data \*irq\_data = irq\_domain\_get\_irq\_data(domain,virq);

irq\_data->hwirq = hwirq;

irq\_data->chip =&gic->chip;

irq\_data->chip\_data = chip\_data;

}

\_\_irq\_set\_handler(virq,handler,0,NULL)

{

unsigned long flags;

struct irq\_desc \*desc = irq\_get\_desc\_buslock(irq,&flags,0);

\_\_irq\_do\_set\_handler(desc,handler,0,NULL);

}

\_\_irq\_do\_set\_handler(desc,handler,0,NULL)

{

if(!handle) {

handle = handle\_bad\_irq;

}else {

struct irq\_data \*irq\_data = &desc->irq\_data;

while(irq\_data) {

if(irq\_data->chip !=&no\_irq\_chip)

break;

}

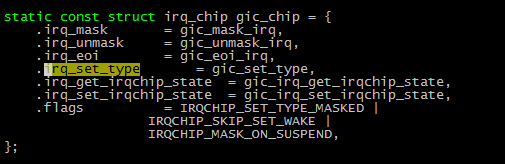
}

desc->handle\_irq =handler;

desc->name = name;

}

## chained



irq\_set\_chained\_handler\_data(irq,handle,data)

{

unsigned long flags;

struct irq\_desc \*desc = irq\_get\_desc\_busblock(irq,&flags,0);

desc->irq\_common\_data.handler\_data = data;

\_\_irq\_do\_set\_handler(desc,handle,1,NULL);

}

\_\_irq\_do\_set\_handler(desc,handle,1,NULL)

{

struct irq\_data \*irq\_data = &desc->irq\_data;

desc->handle\_irq = handle;

desc->name = name;

if(handle!=handle\_bad\_irq&&1)

{

unsigned int type = irqd\_get\_trigger\_type(&desc->irq\_data);

if(chip->flags &IRQCHIP\_SET\_TYPE\_MASKED) {

}

if(type!=IRQ\_TYPE\_NONE)

{

\_\_irq\_set\_trigger(desc,type);

desc->handle\_irq = handle;

}

switch(ret) {

case IRQ\_SET\_MASK\_OK:

case IRQ\_SET\_MASK\_OK\_DONE:

irqd\_clear(&desc->irq\_data,IRQD\_TRIGGER\_MASK);

irqd\_set(&desc->irq\_data,flags);

case IRQ\_SET\_MASK\_OK\_NOCOPY:

flags = irqd\_get\_trigger\_type(&desc->irq\_data);

irq\_settings\_set\_trigger\_mask(desc,flags);

irqd\_clear(&desc->irq\_data,IRQD\_LEVEL);

irq\_settings\_clr\_level(desc);

if(flags &IRQ\_TYPE\_LEVEL\_MASK) {

irq\_settings\_set\_level(desc);

irqd\_set(&desc->irq\_data,IRQD\_LEVEL);

}

ret = 0;

break;

}

}

}

\_\_irq\_set\_trigger(desc,type)

{

struct irq\_chip \*chip = desc->irq\_data.chip;

if(chip->flags &IRQCHIP\_SET\_TYPE\_MASKED)

{

}

flags &=IRQ\_TYPE\_SENSE\_MASK;

ret = chip->irq\_set\_type(&desc->irq\_data,flags);

}

## gic

<https://www.realdigital.org/doc/87be521204ed2c4447d567b666961ce8>

<http://blog.chinaaet.com/weiqi7777/p/5100057557>

### priority

ICCPMR (priority mask register):设置cpu的优先级level，比该优先级低的不会送给cpu。

ICCHPRIO:读该寄存器会返回当前pending的最高优先级的interrupt id

ICDIPR：中断优先级寄存器，设置每个interrupt id的优先级

### affinity

ICDIPTR:设置每个中断的cpu id

### distributor

GICD\_作为前缀

### cpu interface

GICC作为前缀

### spi/sgi/ppi

