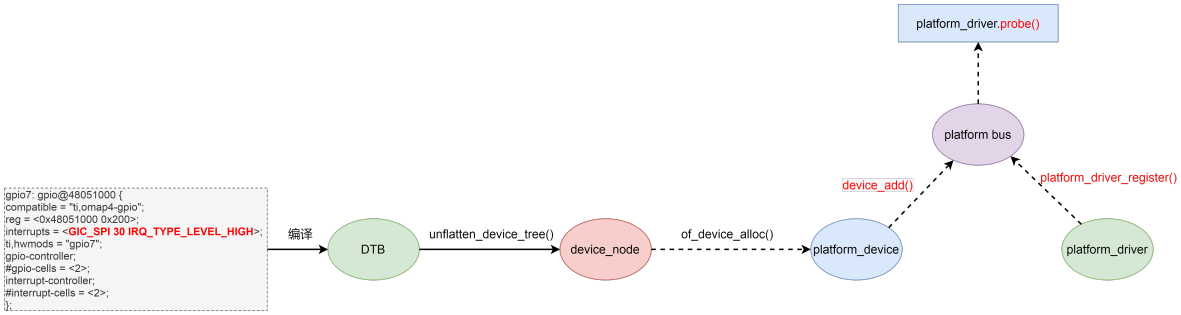
- 对于`gpio@48051000`中的其他普通的GPIO来说，它们产生中断后，并没有直接通知GIC，而是先通知`gpio@48051000`，然后`gpio@48051000`再通过SPI-30通知GIC，然后GIC会通过irq或者firq触发某个CPU中断。
- 每一个irq_domain都对应一个irq_chip，irq_chip是kernel对中断控制器的软件抽象

```
/ {
    #address-cells = <2>;
    #size-cells = <2>;

    compatible = "ti,dra7xx";
    interrupt-parent = <&crossbar_mpu>;
    chosen { };

    gic: interrupt-controller@48211000 {
        compatible = "arm,cortex-a15-gic";
        interrupt-controller;
        #interrupt-cells = <3>;
        reg = <0x0 0x48211000 0x0 0x1000>,
              <0x0 0x48212000 0x0 0x2000>,
              <0x0 0x48214000 0x0 0x2000>,
              <0x0 0x48216000 0x0 0x2000>;
        interrupts = <GIC_PPI 9 (GIC_CPU_MASK_SIMPLE(2) |
IRQ_TYPE_LEVEL_HIGH)>;
        interrupt-parent = <&gic>;
    };

    ocp {
        compatible = "ti,dra7-l3-noc", "simple-bus";
        #address-cells = <1>;
        #size-cells = <1>;
        ranges = <0x0 0x0 0x0 0xc0000000>;
        ti,hwmods = "l3_main_1", "l3_main_2";
        reg = <0x0 0x44000000 0x0 0x1000000>,
              <0x0 0x45000000 0x0 0x1000>;
        interrupts-extended = <&crossbar_mpu GIC_SPI 4
IRQ_TYPE_LEVEL_HIGH>,
                              <&wakeupgen GIC_SPI 10 IRQ_TYPE_LEVEL_HIGH>;

        gpio1: gpio@4ae10000 {
            ......
        };

        gpio2: gpio@48055000 {
            ......
```

```
        };

        gpio3: gpio@48057000 {
            ......
        };

        gpio4: gpio@48059000 {
            ......
        };

        gpio5: gpio@4805b000 {
            ......
        };

        gpio6: gpio@4805d000 {
            ......
        };

        gpio7: gpio@48051000 {
            compatible = "ti,omap4-gpio";
            reg = <0x48051000 0x200>;
            interrupts = <GIC_SPI 30 IRQ_TYPE_LEVEL_HIGH>;
            ti,hwmods = "gpio7";
            gpio-controller;
            #gpio-cells = <2>;
            interrupt-controller;
            #interrupt-cells = <2>;
        };

        gpio8: gpio@48053000 {
            ......
        };
    };
};
```

- root gic就是上面的"arm,cortex-a15-gic"，它的interrupt cells是3, 表示引用gic上的一个中断需要三个参数
- `gpio1: gpio@4ae10000`每组GPIO也是一个中断控制器，它的interrupt parent也是interrupt-interrupt-controller@48211000，interrupt cell是2, 表示引用一个中断需要2个参数。



相关代码：

第一部分： GIC中断控制器的注册。

第二部分：设备树的device node在向platform_device转化的过程中节点的interrupts属性的处理。

第三部分：platform_device注册添加。

第四部分：GPIO控制器驱动的注册，大部分GPIO控制器同时具备interrupt controller的功能。

第五部分：引用GPIO中断的节点的解析。

第一部分 GIC中断控制器的注册

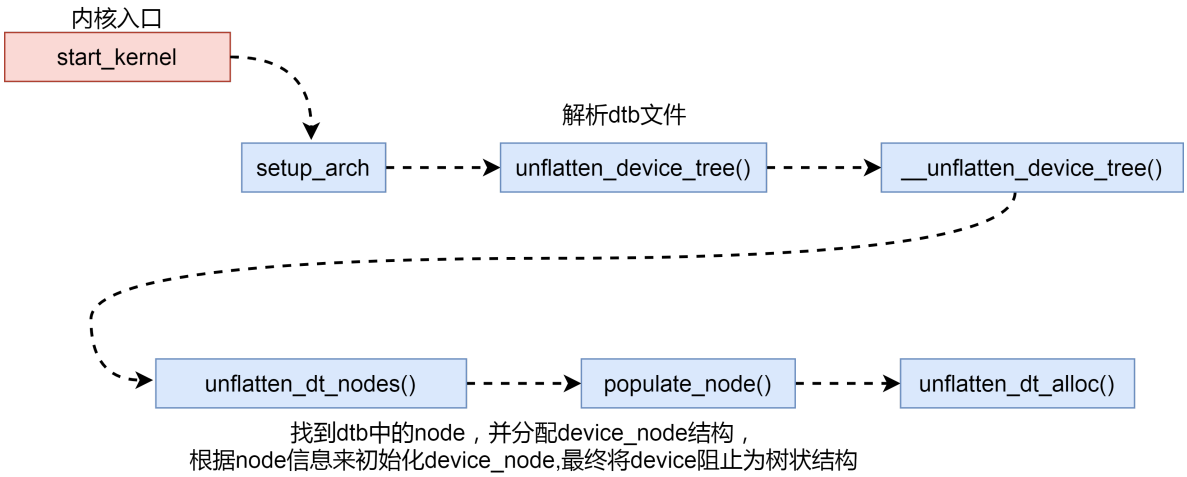
1. GIC驱动分析

ARM平台的设备信息，都是通过`Device Tree`设备树来添加，设备树信息如下

```
/*arch\arm\boot\dts\dra7.dtsi*/
gic: interrupt-controller@48211000 {
        compatible = "arm,cortex-a15-gic";
        interrupt-controller;
        #interrupt-cells = <3>;
        reg = <0x0 0x48211000 0x0 0x1000>,
              <0x0 0x48212000 0x0 0x2000>,
              <0x0 0x48214000 0x0 0x2000>,
              <0x0 0x48216000 0x0 0x2000>;
        interrupts = <GIC_PPI 9 (GIC_CPU_MASK_SIMPLE(2) |
IRQ_TYPE_LEVEL_HIGH)>;
        interrupt-parent = <&gic>;
};
```

- `compatible`字段：用于与具体的驱动来进行匹配，比如图片中`arm,cortex-a15-gic`，可以根据这个名字去匹配对应的驱动程序；
- `interrupt-cells`字段：用于指定编码一个中断源所需要的单元个数，这个值为3。比如在外设在设备树中添加中断信号时，通常能看到类似`interrupts = <0 23 4>;`的信息，第一个单元0，表示的是中断类型（`1：PPI，0：SPI`），第二个单元23表示的是中断号，第三个单元4表示的是中断触发的类型；
- `reg`字段：描述中断控制器的地址信息以及地址范围，比如图片中分别制定了`GIC Distributor(GICD)`和`GIC CPU Interface(GICC)`的地址信息；
- `interrupt-controller`字段：表示该设备是一个中断控制器，外设可以连接在该中断控制器上；
- 关于设备数的各个字段含义，详细可以参考`Documentation/devicetree/bindings`下的对应信息；

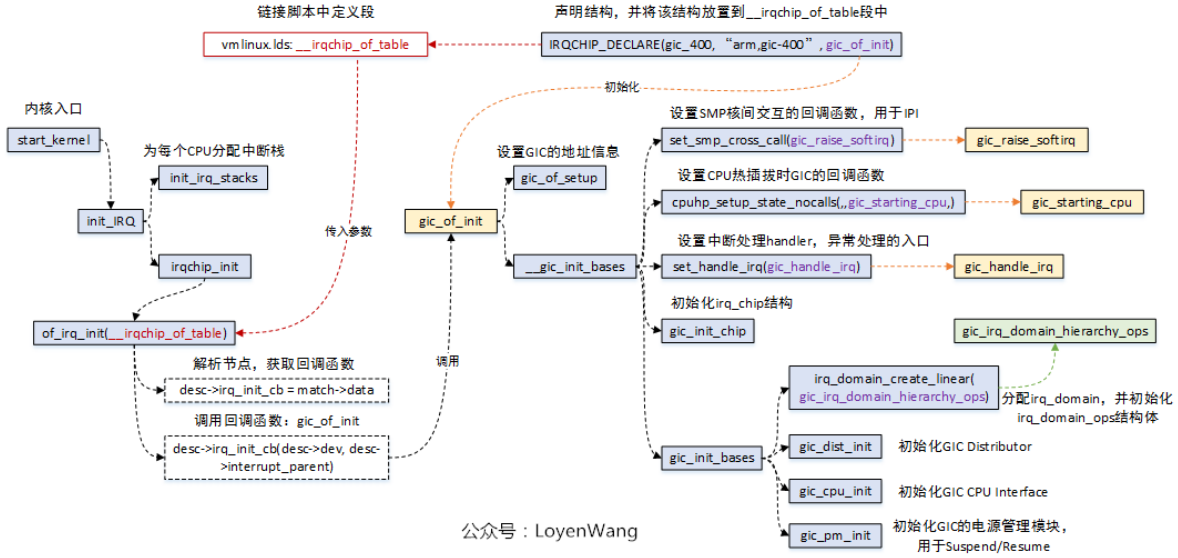
设备树的信息，是怎么添加到系统中的呢？`Device Tree`最终会编译成`dtb`文件，并通过Uboot传递给内核，在内核启动后会将`dtb`文件解析成`device_node`结构。



- 设备树的节点信息，最终会变成`device_node`结构，在内存中维持一个树状结构；

- 设备与驱动，会根据`compatible`字段进行匹配；

2.GIC驱动流程分析



- 首先需要了解一下链接脚本`vmlinux.lds`，脚本中定义了一个`__irqchip_of_table`段，该段用于存放中断控制器信息，用于最终来匹配设备；
- 在GIC驱动程序中，使用`IRQCHIP_DECLARE`宏来声明结构信息，包括`compatible`字段和回调函数，该宏会将这个结构放置到`__irqchip_of_table`字段中；
- 在内核启动初始化中断的函数中，`of_irq_init`函数会去查找设备节点信息，该函数的传入参数就是`__irqchip_of_table`段，由于`IRQCHIP_DECLARE`已经将信息填充好了，`of_irq_init`就会遍历`__irqchip_of_table`，按照interrupt controller的连接关系从root开始，依次初始化每一个interrupt controller,`of_irq_init`函数会根据`arm,gic-400`去查找对应的设备节点，并获取设备的信息。

- `or_irq_init`函数中，最终会回调`IRQCHIP_DECLARE`声明的回调函数，也就是`gic_of_init`，而这个函数就是GIC驱动的初始化入口函数了；

```
IRQCHIP_DECLARE(cortex_a15_gic, "arm,cortex-a15-gic", gic_of_init);
IRQCHIP_DECLARE(cortex_a9_gic, "arm,cortex-a9-gic", gic_of_init);
IRQCHIP_DECLARE(cortex_a7_gic, "arm,cortex-a7-gic", gic_of_init);
```

- GIC的工作，本质上是由中断信号来驱动，因此驱动本身的工作就是完成各类信息的初始化，注册好相应的回调函数，以便能在信号到来之时去执行；
- `set_smp_process_call`设置`__smp_cross_call`函数指向`gic_raise_softirq`，本质上就是通过软件来触发GIC的`SGI中断`，用于核间交互；
- `cpuhp_setup_state_nocalls`函数，设置好CPU进行热插拔时GIC的回调函数，以便在CPU热插拔时做相应处理；
- `set_handle_irq`函数的设置很关键，它将全局函数指针`handle_arch_irq`指向了`gic_handle_irq`，而处理器在进入中断异常时，会跳转到`handle_arch_irq`执行，所以，可以认为它就是中断处理的入口函数了；
- 驱动中完成了各类函数的注册，此外还完成了`irq_chip`，`irq_domain`等结构体的初始化,计算这个GIC模块所支持的中断个数gic_irqs，然后创建一个linear irq domain。此时尚未分配virq，也没有建立hwirq跟virq的映射；

```
    gic_irqs = readl_relaxed(gic_data_dist_base(gic) + GIC_DIST_CTR) & 0x1f;
    gic_irqs = (gic_irqs + 1) * 32;
    if (gic_irqs > 1020)
        gic_irqs = 1020;
    gic->gic_irqs = gic_irqs;

    gic->domain = irq_domain_create_linear(handle, gic_irqs,
                                &gic_irq_domain_hierarchy_ops,
                                gic);
```

在初始化的时候既没有给hwirq分配对应的virq，也没有建立二者之间的映射，这部分工作会到后面有人引用GIC上的某个中断时再分配和建立。

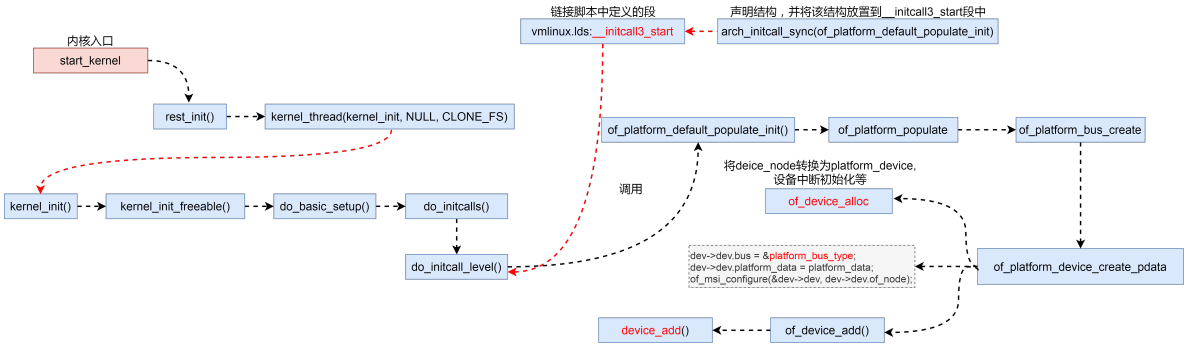
- 最后，完成GIC硬件模块的初始化设置，以及电源管理相关的注册等工作；

第二部分 device node转化为platform_device

相关代码：

drivers/of/platform.c

这个转化过程是调用`of_platform_populate`开始的。以`gpio1: gpio@4ae10000`为例，暂时只关心interrupts属性的处理，函数调用关系：




```
struct platform_device *of_device_alloc(struct device_node *np,
                  const char *bus_id,
                  struct device *parent)
{
    struct platform_device *dev;
    int rc, i, num_reg = 0, num_irq;
    struct resource *res, temp_res;

    dev = platform_device_alloc("", PLATFORM_DEVID_NONE);
    if (!dev)
        return NULL;

    /* count the io and irq resources */
    while (of_address_to_resource(np, num_reg, &temp_res) == 0)
        num_reg++;
    num_irq = of_irq_count(np);/* 统计这个节点的interrupts属性中描述了几个中断*/

    /* Populate the resource table */
    if (num_irq || num_reg) {
        res = kzalloc(sizeof(*res) * (num_irq + num_reg), GFP_KERNEL);
        if (!res) {
```

```
            platform_device_put(dev);
            return NULL;
        }

        dev->num_resources = num_reg + num_irq;
        dev->resource = res;
        for (i = 0; i < num_reg; i++, res++) {
            rc = of_address_to_resource(np, i, res);
            WARN_ON(rc);
        }
        /*解析interrupts属性，将每一个中断转化为resource结构体*/
        if (of_irq_to_resource_table(np, res, num_irq) != num_irq)
            pr_debug("not all legacy IRQ resources mapped for %s\n",
                 np->name);
    }

    dev->dev.of_node = of_node_get(np);
    dev->dev.fwnode = &np->fwnode;
    dev->dev.parent = parent ? : &platform_bus;

    if (bus_id)
        dev_set_name(&dev->dev, "%s", bus_id);
    else
        of_device_make_bus_id(&dev->dev);

    return dev;
}
```

这里主要涉及到两个函数`of_irq_count`和`of_irq_to_resource_table`，传入的np就是`gpio1: gpio@4ae10000`节点。

- of_irq_count

这个函数会解析interrupts属性，并统计其中描述了几个中断。

简化如下：找到`gpio1: gpio@4ae10000`节点的所隶属的interrupt-controller，即interrupt-controller@10490000节点，然后获得其#interrupt-cells属性的值，因为只要知道了这个值，也就知道了在interrupts属性中描述一个中断需要几个参数，也就很容易知道interrupts所描述的中断个数。这里关键的函数是`of_irq_parse_one`：

```
int of_irq_count(struct device_node *dev)
{
    struct of_phandle_args irq;
    int nr = 0;

    while (of_irq_parse_one(dev, nr, &irq) == 0)
        nr++;

    return nr;
}
```

nr表示的是index，of_irq_parse_one每次成功返回，都表示成功从interrupts属性中解析到了第nr个中断，同时将关于这个中断的信息存放到irq中，struct of_phandle_args的含义如下：

```
#define MAX_PHANDLE_ARGS 16
struct of_phandle_args {
    struct device_node *np; // 用于存放赋值处理这个中断的中断控制器的节点
    int args_count;// 就是interrupt-controller的#interrupt-cells的值
    uint32_t args[MAX_PHANDLE_ARGS];// 用于存放具体描述某一个中断的参数的值
};
```

最后将解析到的中断个数返回。

- **of_irq_to_resource_table**

知道interrupts中描述了几个中断后，这个函数开始将这些中断转换为resource，这个是由of_irq_to_resource函数完成。

```
int of_irq_to_resource_table(struct device_node *dev, struct resource *res,
        int nr_irqs)
{
    int i;

    for (i = 0; i < nr_irqs; i++, res++)
        if (of_irq_to_resource(dev, i, res) <= 0)//将这些中断转换为resource
            break;

    return i;
}
```

第二个参数i表示的是index，即interrupts属性中的第i个中断。

```
int of_irq_to_resource(struct device_node *dev, int index, struct resource
*r)
{
    int irq = of_irq_get(dev, index);// 返回interrupts中第index个hwirq中断映射
到的virq

    if (irq < 0)
        return irq;

    /* Only dereference the resource if both the
     * resource and the irq are valid. */
    if (r && irq) { // 将这个irq封装成resource
        const char *name = NULL;

        memset(r, 0, sizeof(*r));
        /*
         * Get optional "interrupt-names" property to add a name
         * to the resource.
         获取可选的“中断名称”属性，以向资源添加名称。*/
        of_property_read_string_index(dev, "interrupt-names", index,
                      &name);

        r->start = r->end = irq;    // 全局唯一的virq
        r->flags = IORESOURCE_IRQ |
irqd_get_trigger_type(irq_get_irq_data(irq));// 这个中断的属性，如上升沿还是下降
沿触发
        r->name = name ? name : of_node_full_name(dev);
    }
```

```
25
26      return irq;
27  }
```

所以，分析重点是irq_of_parse_and_map，这个函数会获得`gpio@4ae10000`节点的interrupts属性的第index个中断的参数，这是通过`of_irq_parse_one`完成的，然后获得该中断所隶属的interrupt-controller的irq domain，也就是前面GIC注册的那个irq domain，利用该domain的`of_xlate`函数从前面表示第index个中断的参数中解析出hwirq和中断类型，最后从系统中为该hwriq分配一个全局唯一的virq，并将映射关系存放到中断控制器的irq domain中，也就是gic的irq domain。

下面结合kernel代码分析一下：

```
1  int of_irq_get(struct device_node *dev, int index)
2  {
3      int rc;
4      struct of_phandle_args oirq;
5      struct irq_domain *domain;
6
7      rc = of_irq_parse_one(dev, index, &oirq);// 获得interrupts的第index个中断参数，并封装到oirq中
8      if (rc)
9          return rc;
10
11     domain = irq_find_host(oirq.np);
12     if (!domain)
13         return -EPROBE_DEFER;
14
15     return irq_create_of_mapping(&oirq); //返回映射到的virq
16 }
```

获取设备数据中的参数，然后调用irq_create_of_mapping映射hwirq到virq，这个过程中先分配virq、分配irq_desc，然后调用domain的map函数建立hwirq到该virq的映射，最后以virq为索引将irq_desc插入基数树。

```
1  unsigned int irq_create_of_mapping(struct of_phandle_args *irq_data)
2  {
3      struct irq_fwspec fwspec;
4
5      of_phandle_args_to_fwspec(irq_data, &fwspec);// 将irq_data中的数据转存到fwspec
6      return irq_create_fwspec_mapping(&fwspec);
7  }
```

```
1  unsigned int irq_create_fwspec_mapping(struct irq_fwspec *fwspec)
2  {
3      struct irq_domain *domain;
4      struct irq_data *irq_data;
5      irq_hw_number_t hwirq;
6      unsigned int type = IRQ_TYPE_NONE;
7      int virq;
8
9      if (fwspec->fwnode) {
10         /*这里的代码主要是找到irq domain。这是根据上一个函数传递进来的参数irq_data的np成员来寻找的*/
11         domain = irq_find_matching_fwspec(fwspec, DOMAIN_BUS_WIRED);
```

```
        if (!domain)
            domain = irq_find_matching_fwspec(fwspec, DOMAIN_BUS_ANY);
    } else {
        domain = irq_default_domain;
    }

    ......
    /*如果没有定义xlate函数，那么取interrupts属性的第一个cell作为HW interrupt ID。
*/
    if (irq_domain_translate(domain, fwspec, &hwirq, &type))
        return 0;

    ......
    /*
     解析完了，最终还是要调用irq_create_mapping函数来创建HW interrupt ID和IRQ
number的映射关系。*/
    virq = irq_find_mapping(domain, hwirq);
    if (virq) {
        if (type == IRQ_TYPE_NONE || type == irq_get_trigger_type(virq))
            return virq;

        if (irq_get_trigger_type(virq) == IRQ_TYPE_NONE) {
            irq_data = irq_get_irq_data(virq);
            if (!irq_data)
                return 0;
            /*如果有需要，调用irq_set_irq_type函数设定trigger type*/
            irqd_set_trigger_type(irq_data, type);
            return virq;
        }

        pr_warn("type mismatch, failed to map hwirq-%lu for %s!\n",
            hwirq, of_node_full_name(to_of_node(fwspec->fwnode)));
        return 0;
    }

    if (irq_domain_is_hierarchy(domain)) {
        // 对于GIC的irq domain这样定义了alloc的domain来说，走这个分支
        virq = irq_domain_alloc_irqs(domain, 1, NUMA_NO_NODE, fwspec);
        if (virq <= 0)
            return 0;
    } else {
        /* Create mapping
        建立HW interrupt ID和IRQ number的映射关系。 */
        virq = irq_create_mapping(domain, hwirq);
        if (!virq)
            return virq;
    }

    irq_data = irq_get_irq_data(virq);
    if (!irq_data) {
        if (irq_domain_is_hierarchy(domain))
            irq_domain_free_irqs(virq, 1);
        else
            irq_dispose_mapping(virq);
        return 0;
    }

    /* Store trigger type */
```

```
    irqd_set_trigger_type(irq_data, type);

    return virq;    //返回映射到的virq
}
```

看一下gic irq domain的translate的过程：

```
static int gic_irq_domain_translate(struct irq_domain *d,
                    struct irq_fwspec *fwspec,
                    unsigned long *hwirq,
                    unsigned int *type)
{
    if (is_of_node(fwspec->fwnode)) {
        if (fwspec->param_count < 3)// 检查描述中断的参数个数是否合法
            return -EINVAL;
    /* 这里加16的目的是跳过SGI中断，因为SGI用于CPU之间通信，不归中断子系统管
10;GIC支持的中断中从0-15号属于SGI，16-32属于PPI，32-1020属于SPI*/
        *hwirq = fwspec->param[1] + 16;

/*从这里可以看到，描述GIC中断的三个参数中第一个表示中断种类，0表示的是SPI，非0表示PPI;
这里加16的意思是跳过PPI;
同时我们也知道了，第二个参数表示某种类型的中断（PPI or SPI）中的第几个（从0开始）*/
        if (!fwspec->param[0])
            *hwirq += 16;
    // 第三个参数表示的中断的类型，如上升沿、下降沿或者高低电平触发
        *type = fwspec->param[2] & IRQ_TYPE_SENSE_MASK;
        return 0;
    }

    ......
    return -EINVAL;
}
```

通过这个函数，我们就获得了fwspec所表示的hwirq和type

接着看一下irq_find_mapping，如果hwirq之前跟virq之间发生过映射，会存放到irq domain中，这个函数就是查询irq domain，以hwirq为索引，寻找virq;

```
unsigned int irq_find_mapping(struct irq_domain *domain,
                  irq_hw_number_t hwirq)
{
    struct irq_data *data;
......
    if (hwirq < domain->revmap_direct_max_irq) {
        data = irq_domain_get_irq_data(domain, hwirq);
        if (data && data->hwirq == hwirq)
            return hwirq;
    }

    /* Check if the hwirq is in the linear revmap. */
    if (hwirq < domain->revmap_size)//如果是线性映射irq domain的条件，hwirq作为
数字下标
        return domain->linear_revmap[hwirq];
......
    data = radix_tree_lookup(&domain->revmap_tree, hwirq);// hwirq作为key
    return data ? data->irq : 0;
```

```
}
```

下面分析virq的分配以及映射，对于GIC irq domain，由于其ops定义了alloc，在注册irq domain的时候会执行`domain->flags |= IRQ_DOMAIN_FLAG_HIERARCHY`

```
int __irq_domain_alloc_irqs(struct irq_domain *domain, int irq_base,
                unsigned int nr_irqs, int node, void *arg,
                bool realloc, const struct cpumask *affinity)
{
    int i, ret, virq;
 // 下面这个函数会从系统中一个唯一的virq，其实就是全局变量allocated_irqs从低位到高位第一个为0的位的位号
// 然后将allocated_irqs的第virq位置为1，然后会为这个virq分配一个irq_desc，virq会存放到irq_desc的irq_data.irq中
// 最后将这个irq_desc存放到irq_desc_tree中，以virq为key，函数irq_to_desc就是以virq为key，查询irq_desc_tree
// 迅速定位到irq_desc
        virq = irq_domain_alloc_descs(irq_base, nr_irqs, 0, node,
                              affinity);

    irq_domain_alloc_irq_data(domain, virq, nr_irqs);
......
    ret = irq_domain_alloc_irqs_hierarchy(domain, virq, nr_irqs, arg);
......
    for (i = 0; i < nr_irqs; i++)
        // 将virq跟hwirq的映射关系存放到irq domain中，这样就可以通过hwirq在该irq_domain中快速找到virq
        irq_domain_insert_irq(virq + i);
.....
    return virq;
}
```

`irq_domain_alloc_irq_data`会根据virq获得对应的`irq_desc`，然后将`domain`赋值给`irq_desc->irq_data->domain`.

`irq_domain_alloc_irqs_recursive`这个函数会调用gic irq domain的`domain->ops->alloc`，即`gic_irq_domain_alloc`

下面分析irq_create_mapping，对于irq domain的ops中没有定义alloc的domain，会执行这个函数

---> irq_create_mapping 为hwirq分配virq，并存放映射到irq domain中

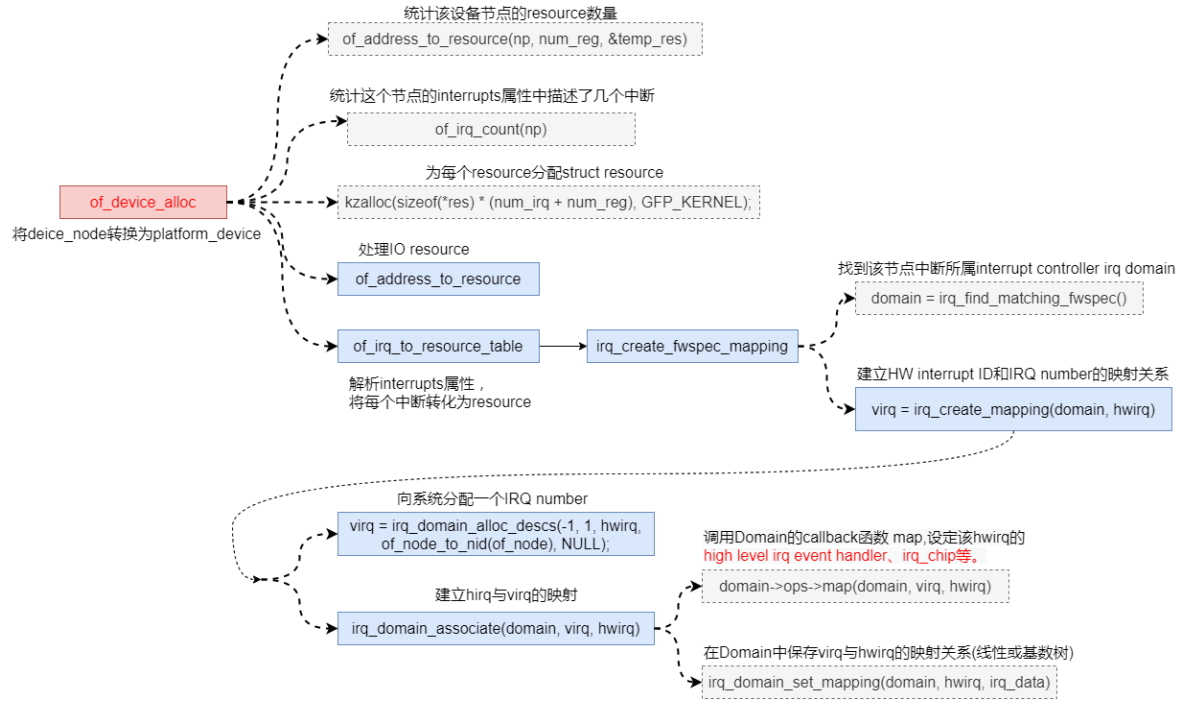
```
unsigned int irq_create_mapping(struct irq_domain *domain,
                irq_hw_number_t hwirq)
{
    struct device_node *of_node;
    int virq;
    ......
    of_node = irq_domain_get_of_node(domain);

    /* Check if mapping already exists
    如果映射已经存在，那么不需要映射，直接返回 */
    virq = irq_find_mapping(domain, hwirq);
    if (virq) {
        pr_debug("-> existing mapping on virq %d\n", virq);
        return virq;
    }
```

```
16
17      /* Allocate a virtual interrupt number 分配虚拟中断号*/
18      virq = irq_domain_alloc_descs(-1, 1, hwirq, of_node_to_nid(of_node),
    NULL);
19  .....
20
21      if (irq_domain_associate(domain, virq, hwirq)) {//建立mapping
22          irq_free_desc(virq);
23          return 0;
24      }
25  .....
26      return virq;
27  }
```

至此，device node在转化为platform_device过程中的interrupts属性的处理就暂时分析完毕，后面会调用`device_add()`注册该platform_device，然后匹配到的platform_driver的probe就会被调用。

通过打印信息可知GPIO7的hwirq与virq的映射关系：

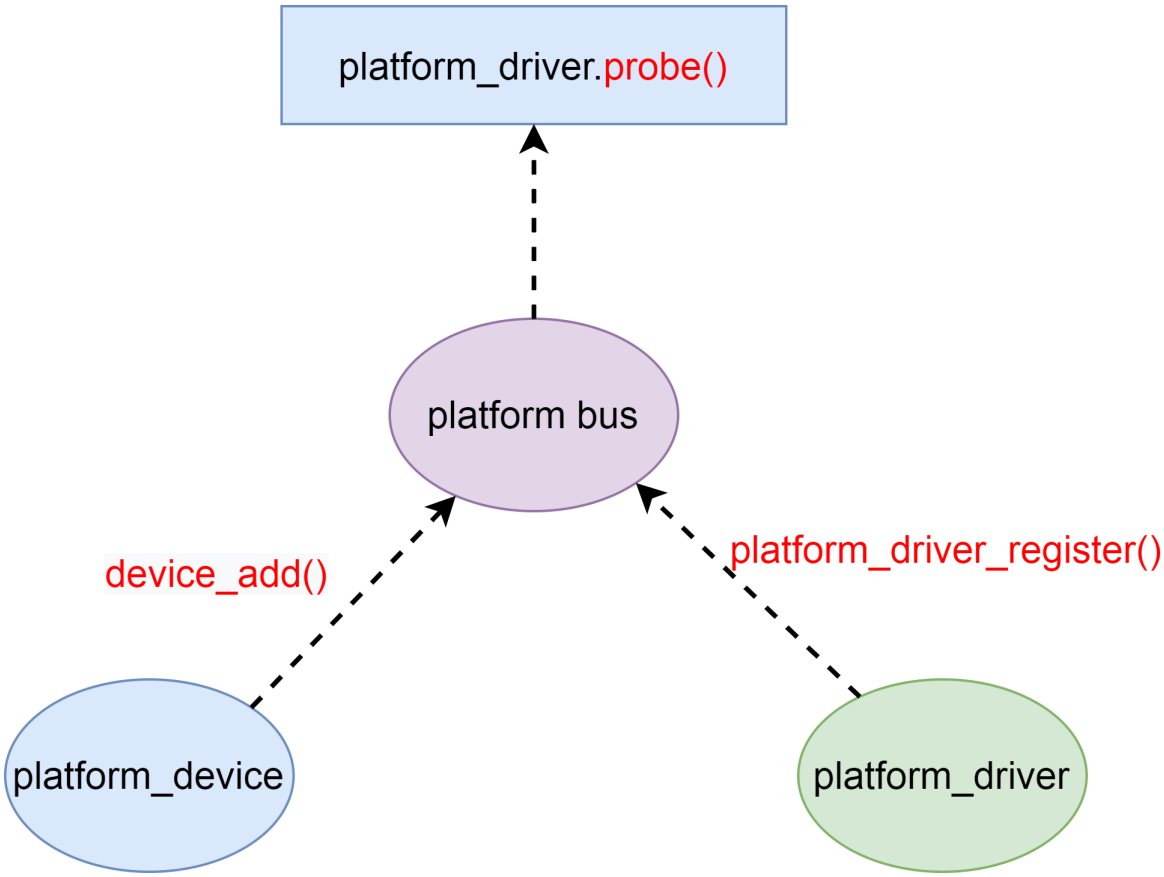
```
1  [19491.235350] virq is 43,hwirq is 30
```



需要关注的是`domain->ops->map()`，该函数中户设置该中断的`desc->handle_irq()`，对于GIC来说，map函数为`gic_irq_domain_map`,SPI中断handle_irq()设置为handle_fasteoi_irq。

第三部分：platform_device注册添加

platform_driver的probe就会被调用。



第四部分 GPIO控制器驱动

相关代码：

drivers\gpio\gpio-omap.c

在`gpio@48051000`节点转化成的platform_device被注册的时候,`omap_gpio_probe()`会被调用。这个函数目前我们先只分析跟中断相关的。

```
1  static int omap_gpio_probe(struct platform_device *pdev)
2  {
3      struct device *dev = &pdev->dev;
4      struct device_node *node = dev->of_node;
```

```
    const struct of_device_id *match;
    const struct omap_gpio_platform_data *pdata;
    struct resource *res;
    struct gpio_bank *bank;
    struct irq_chip *irqc;
    int ret;

    match = of_match_device(of_match_ptr(omap_gpio_match), dev);
    ......
    pdata = match ? match->data : dev_get_platdata(dev);
    ......
    bank = devm_kzalloc(dev, sizeof(struct gpio_bank), GFP_KERNEL);
    ......
    /*irq_chip用于抽象该GPIO中断控制器*/
    irqc = devm_kzalloc(dev, sizeof(*irqc), GFP_KERNEL);
    ......
    irqc->irq_startup = omap_gpio_irq_startup,
    irqc->irq_shutdown = omap_gpio_irq_shutdown,
    irqc->irq_ack = omap_gpio_ack_irq,
    irqc->irq_mask = omap_gpio_mask_irq,
    irqc->irq_mask_ack = omap_gpio_mask_ack_irq,
    irqc->irq_unmask = omap_gpio_unmask_irq,
    irqc->irq_set_type = omap_gpio_irq_type,
    irqc->irq_set_wake = omap_gpio_wake_enable,
    irqc->irq_bus_lock = omap_gpio_irq_bus_lock,
    irqc->irq_bus_sync_unlock = gpio_irq_bus_sync_unlock,
    irqc->name = dev_name(&pdev->dev);
    irqc->flags = IRQCHIP_MASK_ON_SUSPEND | IRQCHIP_PIPELINE_SAFE;

    bank->irq = platform_get_irq(pdev, 0);/*该irq已经是虚拟的了 详见
of_irq_to_resource*/
    ......
    bank->chip.parent = dev;
    bank->chip.owner = THIS_MODULE;
    bank->dbck_flag = pdata->dbck_flag;
    bank->stride = pdata->bank_stride;
    bank->width = pdata->bank_width;/*该bank GPIO数*/
    bank->is_mpuio = pdata->is_mpuio;
    bank->non_wakeup_gpios = pdata->non_wakeup_gpios;
    bank->regs = pdata->regs;
#ifdef CONFIG_OF_GPIO
    bank->chip.of_node = of_node_get(node);
#endif
    ......
    platform_set_drvdata(pdev, bank);

    ......
    ret = omap_gpio_chip_init(bank, irqc);/*完成GPIO中断控制器注册*/
    ......

    omap_gpio_show_rev(bank);
    ......
    list_add_tail(&bank->node, &omap_gpio_list);

    return 0;
}
```

需要注意的是，通过`platform_get_irq(pdev, 0)`获取该bank对应的中断时，已经是virq了，不是设备树里指定的GIC hwirq。

`omap_gpio_chip_init`为该bank注册GPIO中断控制器。

```
static int omap_gpio_chip_init(struct gpio_bank *bank, struct irq_chip
*irqc)
{
    struct gpio_irq_chip *irq;
    static int gpio;
    const char *label;
    int irq_base = 0;
    int ret;

    /*GPIO操作回调函数*/
    bank->chip.request = omap_gpio_request;
    bank->chip.free = omap_gpio_free;
    bank->chip.get_direction = omap_gpio_get_direction;
    bank->chip.direction_input = omap_gpio_input;
    bank->chip.get = omap_gpio_get;
    bank->chip.get_multiple = omap_gpio_get_multiple;
    bank->chip.direction_output = omap_gpio_output;
    bank->chip.set_config = omap_gpio_set_config;
    bank->chip.set = omap_gpio_set;
    bank->chip.set_multiple = omap_gpio_set_multiple;

    label = devm_kasprintf(bank->chip.parent, GFP_KERNEL, "gpio-%d-%d",
                           gpio, gpio + bank->width - 1);

    bank->chip.label = label;
    bank->chip.base = gpio;//该bank中的第一个gpio的逻辑gpio号

    bank->chip.ngpio = bank->width;//该bank GPIO数

    irq = &bank->chip.irq;
    irq->chip = irqc;  //设置该bank 的irq_chip
    irq->handler = handle_bad_irq; //该中断控制器默认中断处理函数
    irq->default_type = IRQ_TYPE_NONE;  //中断默认触发方式
    irq->num_parents = 1;
    irq->parents = &bank->irq;
    irq->first = irq_base;//该GPIO中断控制器的起始中断号0

    ret = gpiochip_add_data(&bank->chip, bank);
// 这里的d->irq是节点gpio@48051000的interrupts属性所映射到的virq，对应的hwirq就是
SPI-30
// 这里申请了中断，在中断处理函数omap_gpio_irq_handler中会获得发生中断的引脚，转化为
该GPIO控制器的hwirq，再进行一步处理
    ret = devm_request_irq(bank->chip.parent, bank->irq,
                   omap_gpio_irq_handler,
                   0, dev_name(bank->chip.parent), bank);
ank->width;

    return ret;
}
```

`gpiochip_add_data`

```c
#define gpiochip_add_data(chip, data) gpiochip_add_data_with_key(chip,
data, NULL, NULL)
int gpiochip_add_data_with_key(struct gpio_chip *chip, void *data,
                   struct lock_class_key *lock_key,
                   struct lock_class_key *request_key)
{
    unsigned long   flags;
    int     status = 0;
    unsigned    i;
    int     base = chip->base;
    struct gpio_device *gdev;

     // 每一个bank都都应一个唯一的gpio_device和gpio_chip
    gdev = kzalloc(sizeof(*gdev), GFP_KERNEL);

    gdev->dev.bus = &gpio_bus_type;
    gdev->chip = chip;
    chip->gpiodev = gdev;
    if (chip->parent) {
        gdev->dev.parent = chip->parent;
        gdev->dev.of_node = chip->parent->of_node;
    }

#ifdef CONFIG_OF_GPIO
    /* If the gpiochip has an assigned OF node this takes precedence */
    if (chip->of_node)
        gdev->dev.of_node = chip->of_node;
    else
        chip->of_node = gdev->dev.of_node;
#endif
    // 分配一个唯一的id
    gdev->id = ida_simple_get(&gpio_ida, 0, 0, GFP_KERNEL);

    dev_set_name(&gdev->dev, "gpiochip%d", gdev->id);
    device_initialize(&gdev->dev);
    dev_set_drvdata(&gdev->dev, gdev);
    if (chip->parent && chip->parent->driver)
        gdev->owner = chip->parent->driver->owner;
    else if (chip->owner)
        /* TODO: remove chip->owner */
        gdev->owner = chip->owner;
    else
        gdev->owner = THIS_MODULE;

    // 为这个chip下的每一个gpio都要分配一个gpio_desc结构体
    gdev->descs = kcalloc(chip->ngpio, sizeof(gdev->descs[0]), GFP_KERNEL);

    gdev->label = kstrdup_const(chip->label ?: "unknown", GFP_KERNEL);

    // 这个chip中含有的gpio的个数
    gdev->ngpio = chip->ngpio;
    //gdev->data代表这个bank
    gdev->data = data;

    spin_lock_irqsave(&gpio_lock, flags);

    // base表示的是这个bank在系统中的逻辑gpio号
    gdev->base = base;
```

```
58
59      // 将这个bank对应的gpio_device添加到全局链表gpio_devices中
60      // 在添加的时候会根据gdev->base和ngpio在gpio_devices链表中找到合适的位置
61      status = gpiodev_add_to_list(gdev);
62
63      spin_unlock_irqrestore(&gpio_lock, flags);
64
65      /*为每个GPIO分配gpio_desc，建立与gdev的联系*/
66      for (i = 0; i < chip->ngpio; i++) {
67          struct gpio_desc *desc = &gdev->descs[i];
68
69          desc->gdev = gdev;
70
71          desc->flags = !chip->direction_input ? (1 << FLAG_IS_OUT) : 0;
72      }
73
74      // 默认这个chip下的所有gpio都是可以产生中断
75      status = gpiochip_irqchip_init_valid_mask(chip);
76
77      status = gpiochip_init_valid_mask(chip);
78      /*为该bank添加irq_chip,并创建一个irq_domain
79      只是创建了irq domain，还没有存放任何中断映射关系，在需要的时候才会映射。*/
80      status = gpiochip_add_irqchip(chip, lock_key, request_key);
81
82      status = of_gpiochip_add(chip);
83
84      acpi_gpiochip_add(chip);
85
86      machine_gpiochip_add(chip);
87
88      if (gpiolib_initialized) {
89          status = gpiochip_setup_dev(gdev);
90      }
91      return 0;
92  }
```

---> of_gpiochip_add(struct gpio_chip *chip)

```
1   int of_gpiochip_add(struct gpio_chip *chip)
2   {
3       int status;
4   ......
5       if (!chip->of_xlate) {
6           /*pio_chip的of_gpio_n_cells被赋值为2，表示引用一个gpio资源需要两个参数，
7           负责解析这两个参数函数以的of_xlate函数为of_gpio_simple_xlate，
8           其中第一个参数表示gpio号(在对应的bank中)，第二个表示flag*/
9           chip->of_gpio_n_cells = 2;
10          chip->of_xlate = of_gpio_simple_xlate;
11      }
```

这里需要看一下of_gpio_simple_xlate的实现，这个在下面的分析中会被回调.

```
int of_gpio_simple_xlate(struct gpio_chip *gc,
             const struct of_phandle_args *gpiospec, u32 *flags)
{
    ......
    if (flags)      // 第二个参数表示的是flag
        *flags = gpiospec->args[1];
    // 第一个参数表示的是gpio号
    return gpiospec->args[0];
}
```

下看创建domain流程：

```
static int gpiochip_add_irqchip(struct gpio_chip *gpiochip,
                struct lock_class_key *lock_key,
                struct lock_class_key *request_key)
{
    struct irq_chip *irqchip = gpiochip->irq.chip;
    const struct irq_domain_ops *ops;
    struct device_node *np;
    unsigned int type;
    unsigned int i;
......
    np = gpiochip->gpiodev->dev.of_node;
    type = gpiochip->irq.default_type;      //默认触发类型
.....
    gpiochip->to_irq = gpiochip_to_irq;   /*驱动request irq时调用*/
    gpiochip->irq.default_type = type;
    gpiochip->irq.lock_key = lock_key;
    gpiochip->irq.request_key = request_key;

    if (gpiochip->irq.domain_ops)
        ops = gpiochip->irq.domain_ops;
    else
        ops = &gpiochip_domain_ops;

    /* 创建一个linear irq domain，从这里看到，每一个bank都会有一个irq domain，
ngpio是这个bank含有的gpio的个数，也是这个irq domain支持的中断的个数*/
    gpiochip->irq.domain = irq_domain_add_simple(np, gpiochip->ngpio,
                             gpiochip->irq.first,
                             ops, gpiochip);
    ......
    return 0;
}

```

上面也只是创建了irq domain，还没有存放任何中断映射关系，在需要的时候才会映射。

该irq domain的irq_domain_ops为`gpiochip_domain_ops`;

```
static const struct irq_domain_ops gpiochip_domain_ops = {
    .map    = gpiochip_irq_map,
    .unmap  = gpiochip_irq_unmap,
    /* Virtually all GPIO irqchips are twocell:ed */
    .xlate  = irq_domain_xlate_twocell,
};
```

gpio7这个中断在GIC级的处理函数注册为`omap_gpio_irq_handler`;

```
ret = devm_request_irq(bank->chip.parent, bank->irq,
                       omap_gpio_irq_handler,
                       0, dev_name(bank->chip.parent), bank);
```

为`bank->irq`创建一个action，设置该`action->handler`为`omap_gpio_irq_handler`,将该action添加到`bank->irq`对应的irq_desc的actions链表。

第五部分 引用GPIO中断的节点的解析

从上面的分析中我们知道了如下几点：

1. 每一个bank都对应一个gpio_chip和gpio_device
2. 这个bank下的每一个gpio都会对应一个唯一的gpio_desc结构体，这些结构提的首地址存放在gpio_device的desc中
3. 上面的gpio_device会加入到全局gpio_devices链表中
4. gpio_chip的of_gpio_n_cells被赋值为2，表示引用一个gpio资源需要两个参数，负责解析这两个参数函数以的of_xlate函数为of_gpio_simple_xlate，其中第一个参数表示gpio号(在对应的bank中)，第二个表示flag

掉电保护功能设备树节点入下：

```
powerdown_protect__pins_default: powerdown_protect__pins_default {
    pinctrl-single,pins = <
        DRA7XX_CORE_IOPAD(0x37a4, (PIN_INPUT_PULLUP | MUX_MODE14)) /*
gpio7_7 */
        DRA7XX_CORE_IOPAD(0x34fc, (PIN_OUTPUT | MUX_MODE14)) /* gpio3_6 */
        >;
};
    powerdown_protect {
        compatible = "greerobot,powerdown_protect";
        pinctrl-names = "default";
        pinctrl-0 = <&powerdown_protect__pins_default>;
        powerdown_detect_gpio  = <&gpio7 7 GPIO_ACTIVE_HIGH>;
        powerdown_ssd_en  = <&gpio3 6 GPIO_ACTIVE_HIGH>;
    };
```

上面的节点powerdown_protect中引用了gpio3、gpio7，而且在驱动中打算将这个gpio当作中断引脚来使用。

下面是掉电检测的驱动：

```
.....
int gpio_id = -1;
int ssd_en  = -1;
int irq_num = -1;

......
static int powerdown_protect_probe(struct platform_device *pdev)
{
    struct device *dev = &pdev->dev;
    struct device_node *node = dev->of_node;
    int ret = -1;

```

```
    gpio_id = of_get_named_gpio(node, "powerdown_detect_gpio", 0);
.....
    ret = gpio_request(gpio_id, "powerdown_detect");
.....
    irq_num = gpio_to_irq(gpio_id);
.....
    ret = request_irq(irq_num, powerdown_detect_irq, IRQFLAGS, IRQDESC,
pdev);
.....

    ret = misc_register(&pwd_miscdev);
    ....

    return 0;

fail:
    gpio_free(gpio_id);
    return ret;
}

static int powerdown_protect_remove(struct platform_device *pdev)
{
    free_irq(irq_num, pdev);
    gpio_free(gpio_id);
    return 0;
}

static const struct of_device_id powerdown_protect_match[] = {
    { .compatible = "greerobot,powerdown_protect", },
    {}
};

static struct platform_driver powerdown_protect_driver = {
    .probe = powerdown_protect_probe,
    .remove = powerdown_protect_remove,
    .driver = {
        .name = "greerobot_powerdown_protect",
        .owner = THIS_MODULE,
        .of_match_table = powerdown_protect_match,
    },
};

static __init int powerdown_protect_init(void)
{
    return platform_driver_register(&powerdown_protect_driver);
}

module_init(powerdown_protect_init);
```

其中我们只需要分析两个关键的函数：`of_get_named_gpio` 和 `gpio_to_irq`.

of_get_named_gpio

这个函数的作用是根据传递的属性的name和索引号，得到一个gpio号

```
int of_get_named_gpio_flags(struct device_node *np, const char *list_name,
                int index, enum of_gpio_flags *flags)
{
    struct gpio_desc *desc;

    desc = of_get_named_gpiod_flags(np, list_name, index, flags);
.....
    return desc_to_gpio(desc);
}
```

```
struct gpio_desc *of_get_named_gpiod_flags(struct device_node *np,
             const char *propname, int index, enum of_gpio_flags *flags)
{
    struct of_phandle_args gpiospec;
    struct gpio_chip *chip;
    struct gpio_desc *desc;
    int ret;

    /* 解析"powerdown_detect_gpio"属性中第index字段，将解析结果存放到gpiospec中
struct of_phandle_args {
    struct device_node *np;  // int-gpio属性所引用的gpio-controller的node--
gpio7
    int args_count;  // gpio7这个gpio-controller的#gpio-cells属性的值
    uint32_t args[MAX_PHANDLE_ARGS];  // 具体描述这个gpio属性的每一个参数
};
*/
    ret = of_parse_phandle_with_args_map(np, propname, "gpio", index,
                          &gpiospec);

// 上面gpiospec的np存放的索引用的gpio-controller的node,
// 遍历gpio_devices链表，找到对应的gpio_device，也就找到了gpio_chip
    chip = of_find_gpiochip_by_xlate(&gpiospec);
// 调用chip->of_xlate解析gpiospec，返回gpiospec的args中的第一个参数args[0]，
// 也就是前面分析的在bank中的逻辑gpio号
// 知道了gpio号，就可以在gpio_device->desc中索引到对应的gpio_desc
    desc = of_xlate_and_get_gpiod_flags(chip, &gpiospec, flags);
.....
    return desc;
}
```

```
int desc_to_gpio(const struct gpio_desc *desc)
{
    // 获得这个gpio_desc对应的gpio在系统中的逻辑gpio号
    return desc->gdev->base + (desc - &desc->gdev->descs[0]);
}
```

gpio_to_irq

将这个gpio转换成对应的virq

gpio_to_irq(irq_gpio)

---> __gpio_to_irq(gpio)

---> gpiod_to_irq(gpio_to_desc(gpio))

这里调用了两个函数，函数`gpio_to_desc`根据传入的全局逻辑gpio号找到对应的`gpio_desc`，原理是：遍历`gpio_devices`链表，根据传入的逻辑gpio号，就可以定位到所属的gpio_device，前面说过，在将gpio_device加入到`gpio_devices`链表的时候，不是乱加的，而是根据gpio_device的base和ngpio找到一个合适的位置。找到了gpio_device，那么通过索引它的desc成员，就可以找到对应的gpio_desc

```
struct gpio_desc *gpio_to_desc(unsigned gpio)
{
    struct gpio_device *gdev;
    unsigned long flags;

    spin_lock_irqsave(&gpio_lock, flags);

    list_for_each_entry(gdev, &gpio_devices, list) {
        if (gdev->base <= gpio &&
            gdev->base + gdev->ngpio > gpio) {
            spin_unlock_irqrestore(&gpio_lock, flags);
            return &gdev->descs[gpio - gdev->base];
        }
    }
......
    return NULL;
}
```

```
int gpiod_to_irq(const struct gpio_desc *desc)
{
    struct gpio_chip *chip;
    int offset;

    .......
    chip = desc->gdev->chip;

    offset = gpio_chip_hwgpio(desc);
    if (chip->to_irq) {
        int retirq = chip->to_irq(chip, offset);
    ...
        return retirq;
    }
    return -ENXIO;
}
```

其to_irq定义如下

```
static int gpiochip_to_irq(struct gpio_chip *chip, unsigned offset)
{
....
    return irq_create_mapping(chip->irq.domain, offset);
}

```

需要注意的是offset，比如对于gpio7.7，那么offset就是7，这里的offset就是GPIO7这个控制器的hwirq，调用irq_create_mapping可以为该hwirq在kernel中分配一个唯一的virq，同时将hwirq和virq的映射关系存放到bank->irq_domain中。

映射过程中会设置该virq的higth level handler函数，上节我们在GPIO控制器驱动中注册了gpio_chip的handler为`handle_bad_irq`,此时我们还没有调用`request_irq()`来设置该中断的处理函数，所以disable该中断，desc->handler_irq也只能是`handle_bad_irq`。

```
void
__irq_do_set_handler(struct irq_desc *desc, irq_flow_handler_t handle,
             int is_chained, const char *name)
{
....
    if (handle == handle_bad_irq) {
....
        irq_state_set_disabled(desc);
        if (is_chained)
            desc->action = NULL;
        desc->depth = 1;
    }
    desc->handle_irq = handle;
    desc->name = name;
....
}
```

最后将注册该中断的中断处理函数:

```
    irq_num = gpio_to_irq(gpio_id);
    .....
    ret = request_irq(irq_num, powerdown_detect_irq, IRQFLAGS, IRQDESC,
pdev);
```

创建一个action，设置该action-handler为`powerdown_detect_irq`,将该action添加到irq_num对应的irq_desc的actions链表，然后`__setup_irq()`,在没调用request_irq前desc->handler_irq是`handle_bad_irq`，现在我们要根据具体的中断触发方式来设置了，最终调用上面gpio中断控制器中注册的函数`omap_gpio_irq_type()`。

```
request_irq
 -->__setup_irq
    -->__irq_set_trigger
      ->ret = chip->irq_set_type(&desc->irq_data, flags);
/*gpio控制器注册的irq_set_type回调函数omap_gpio_irq_type()*/
```

`omap_gpio_irq_type()`根据中断类类型来设置相应的`desc->handler_irq`，即`handle_simple_irq`

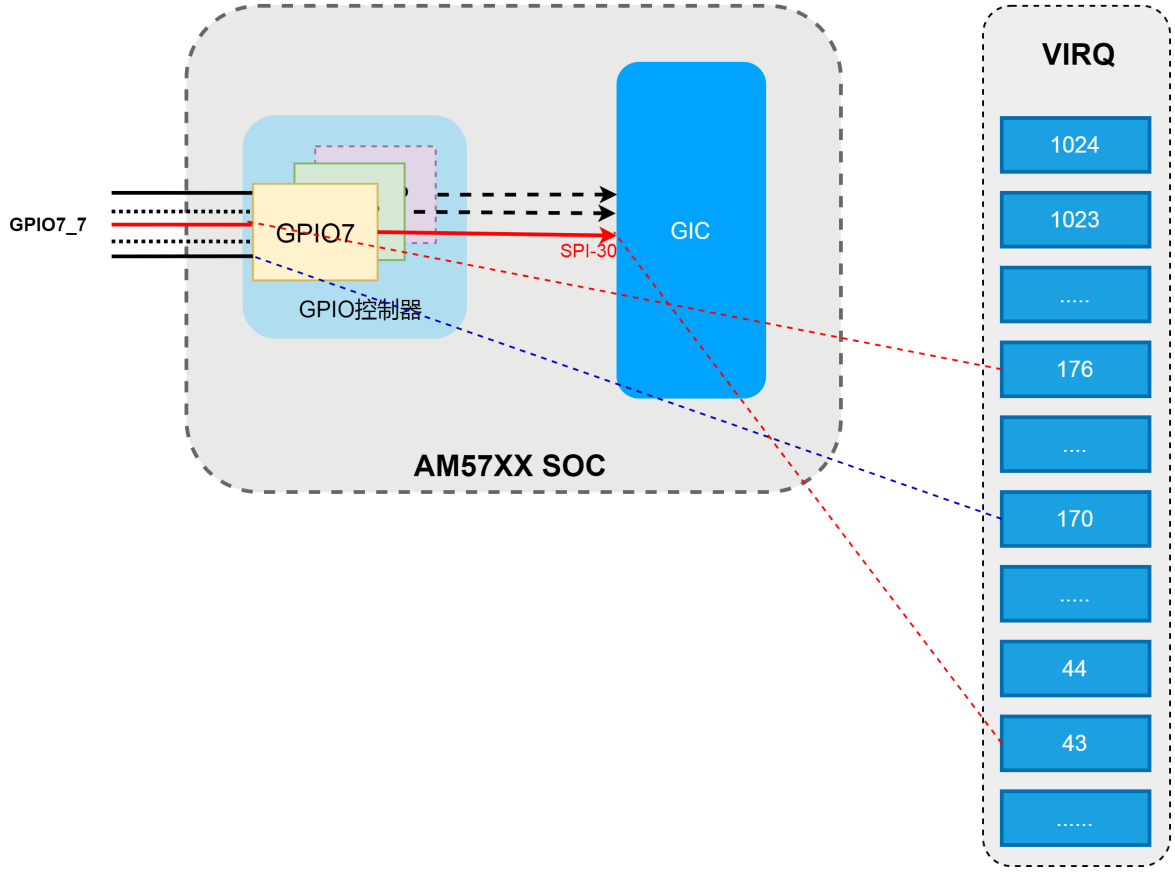
```
static int omap_gpio_irq_type(struct irq_data *d, unsigned type)
{
    struct gpio_bank *bank = omap_irq_data_get_bank(d);
    int retval;
    unsigned long flags;
    unsigned offset = d->hwirq;

    if (type & ~IRQ_TYPE_SENSE_MASK)
        return -EINVAL;

    if (!bank->regs->leveldetect0 &&
        (type & (IRQ_TYPE_LEVEL_LOW|IRQ_TYPE_LEVEL_HIGH)))
        return -EINVAL;
```

```
14
15      raw_spin_lock_irqsave(&bank->lock, flags);
16      retval = omap_set_gpio_triggering(bank, offset, type);
17      if (retval) {
18          raw_spin_unlock_irqrestore(&bank->lock, flags);
19          goto error;
20      }
21      omap_gpio_init_irq(bank, offset);
22      if (!omap_gpio_is_input(bank, offset)) {
23          raw_spin_unlock_irqrestore(&bank->lock, flags);
24          retval = -EINVAL;
25          goto error;
26      }
27      raw_spin_unlock_irqrestore(&bank->lock, flags);
28
29      if (type & (IRQ_TYPE_LEVEL_LOW | IRQ_TYPE_LEVEL_HIGH))
30          irq_set_handler_locked(d, handle_level_irq);
31      else if (type & (IRQ_TYPE_EDGE_FALLING | IRQ_TYPE_EDGE_RISING))
32          /*
33           * Edge IRQs are already cleared/acked in irq_handler and
34           * not need to be masked, as result handle_edge_irq()
35           * logic is excessed here and may cause lose of interrupts.
36           * So just use handle_simple_irq.
37           */
38          irq_set_handler_locked(d, handle_simple_irq);
39
40      return 0;
41
42  error:
43      return retval;
44  }
```

可以看到加载掉电保护驱动后，执行gpio_to_irq时创建了hwirq和virq之间的映射，分配到的virq是176.

```
1  [   12.189454] __irq_alloc_descs: alloc virq: 176, cnt: 1
2  [   12.195729] irq: irq 7 on domain gpio@48051000 mapped to virtual irq 176
3  [   12.195850] powerdown irq is 176
```

到此可以得到以下映射图：



第六部分 GPIO中断处理流程

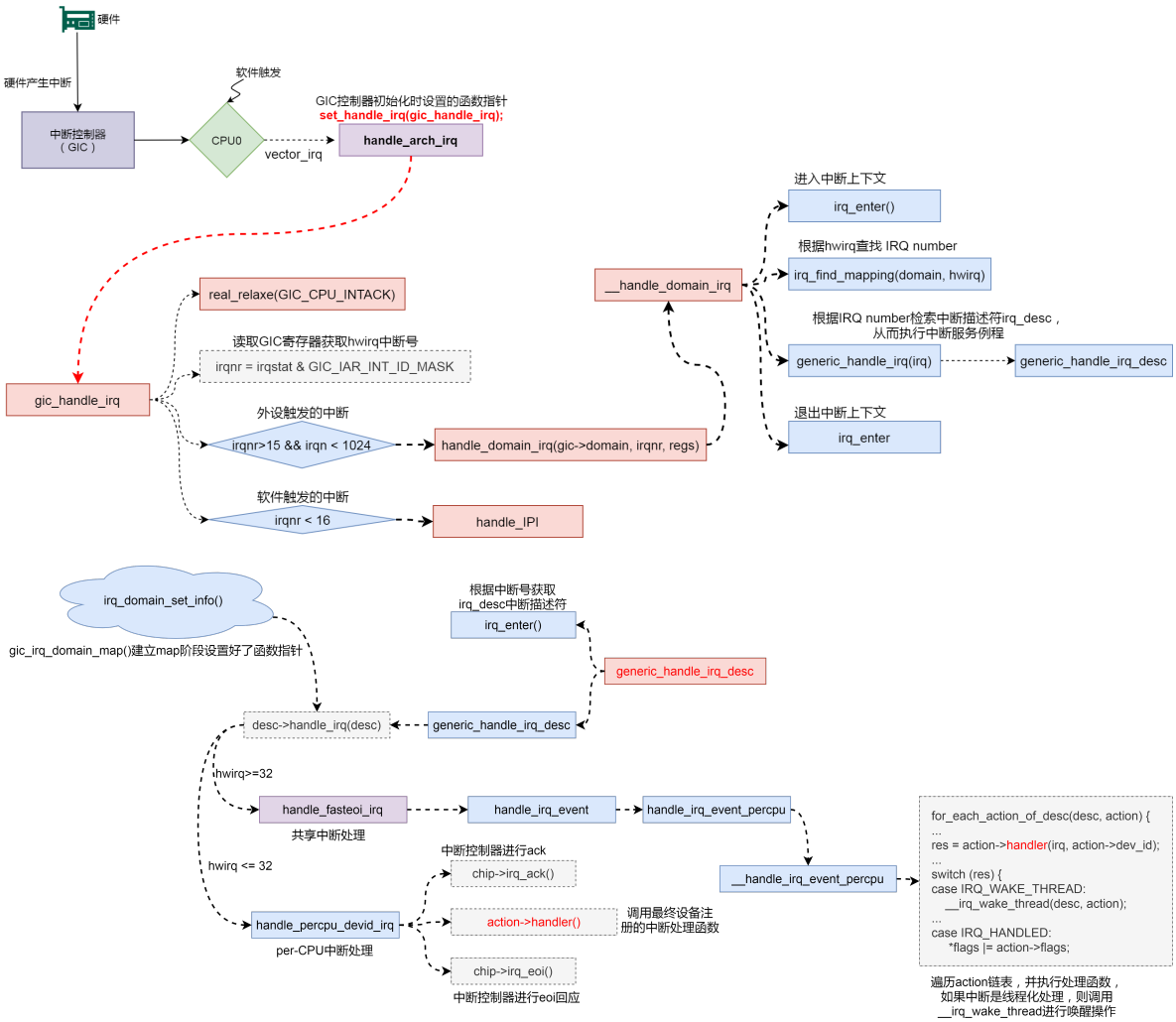
回顾上述分析流程，GPIO7的hwirq：30，其virq：43，维护映射关系的为中断控制器GIC的irq_domain，中断处理函数在GPIO控制器驱动中设置：

```
/*drivers/gpio/gpio-omap.c*/
ret = devm_request_irq(bank->chip.parent, bank->irq,
                       omap_gpio_irq_handler,
                       0, dev_name(bank->chip.parent), bank);
```

掉电检测引脚GPIO7_7，其中断控制器为GPIO7，hwirq：7，其virq为：176,维护映射关系的为GPIO中断控制器的irq_domain，GPIO7_7的中断处理函数在掉电保护驱动中设置：

```
/*drivers/gree/gree_power_down.c*/
ret = request_irq(irq_num, powerdown_detect_irq, IRQFLAGS, IRQDESC, pdev);
```

检测到中断事件后：



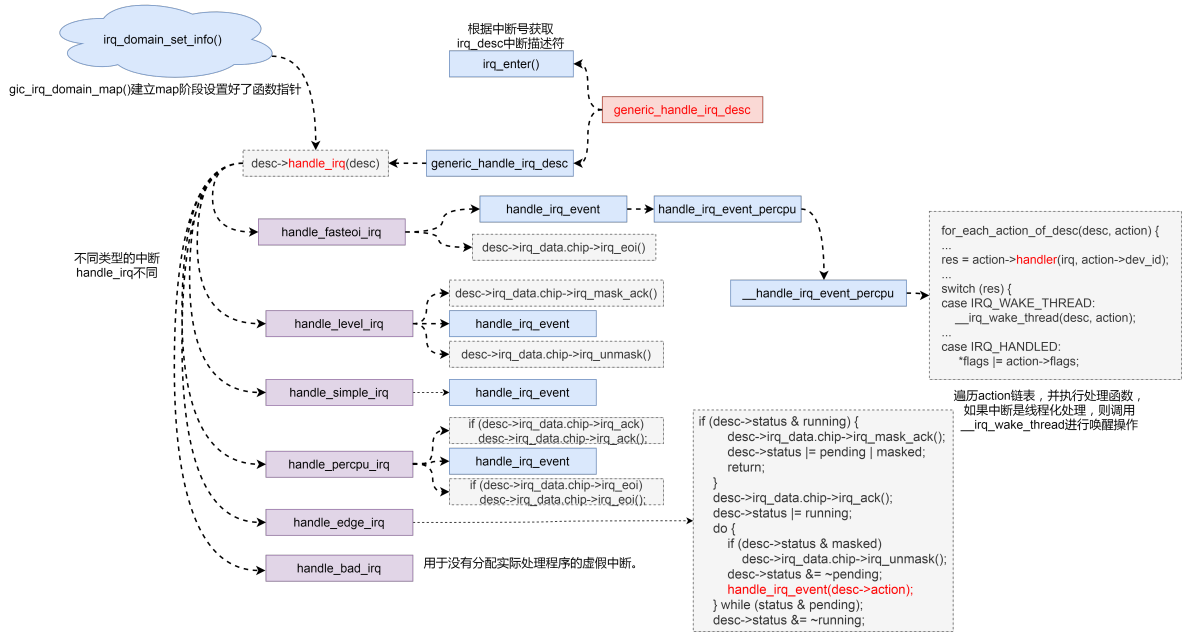
1. 先找到root interrupt controler(GIC)对应的irq_domain；
2. 根据HW寄存器信息和irq_domain信息获取,即30；
3. 调用`handle_IRQ`来处理该hwirq；
4. 调用`irq_find_mapping`找到hwirq对应的IRQ NUMBER 43；
5. 最终调用到`generic_handle_irq`来进行中断处理,即`desc->handle_irq()`。

`desc->handle_irq()`在GPIO7的设备节点转换为platform_device过程中已设置为`handle_fasteoi_irq`，可能是其他函数;

`handle_fasteoi_irq()`进一步得到virq 43对应的irq_desc，并遍历执行链表desc->actions内的action函数，`omap_gpio_irq_handler`得到执行。

以上是GIC中断控制器层处理硬件中断号30的流程，`generic_handle_irq`最终会处理GPIO7 驱动注册的处理函数`omap_gpio_irq_handler`,流程如下：

`omap_gpio_irq_handler`中重复上面`generic_handle_irq`步骤：



1. 找到GPIO7 interrupt controler对应的irq_domain
2. 根据HW寄存器信息和irq_domain信息获取offset,即7；
3. 调用`irq_find_mapping`找到hwirq对应的IRQ Number 176；
4. 调用`generic_handle_irq`处理该IRQ Number 176.
5. 根据virq得到irq_desc，执行`desc->handle_irq()`；

`desc->handle_irq`在掉电检测驱动中request_irq时根据irq来设定，具体为`handle_simple_irq()`,`handle_simple_irq()`中最终遍历action list，调用specific handler，也就是我们掉电检测驱动注册的中断处理函数`powerdown_detect_irq`。