背景：

在request_threaded_irq里面用到的入参unsigned int irq，是一个linux的虚拟/软件中断号。

而soc的中断号是指硬件中断号，和虚拟中断号不一样。

而这个中断号是来自于dts的of_xxx接口的解析。

也就是说of_xxx返回的中断号，已经是虚拟中断号了。

本文就细看下具体硬件中断号是怎么映射到虚拟中断号的。

首先为什么要做映射？

1.早期soc只有一个中断控制器，没必要映射。

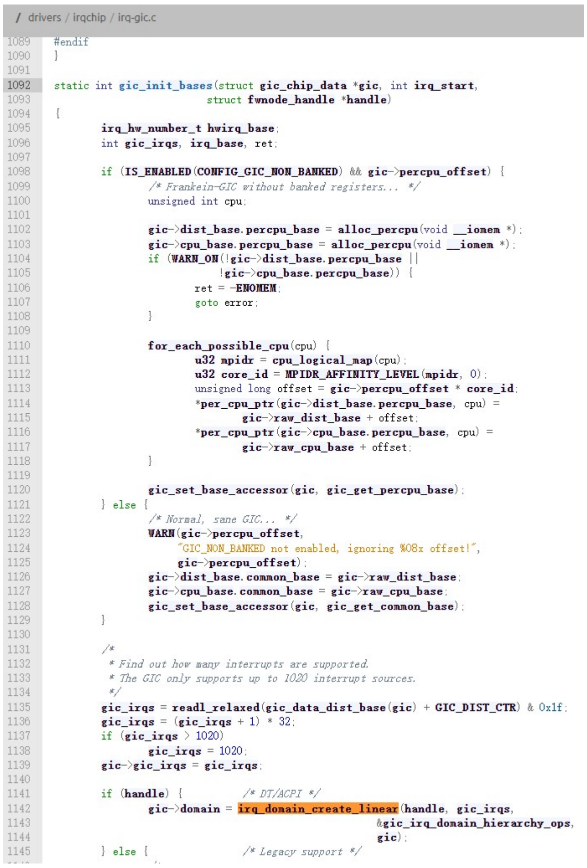
2.后来的soc开始有多个中断控制器，多个控制器之间，除了硬件中断号的编号重复外，控制器之间还有树状结构的关系，不引入虚拟中断号已经没法准确描述和区分了。

在解释映射中断之前，先看下中断控制器的注册过程：

以s32v234.dtsi的中断控制器dts为例：

```
    gic: interrupt-controller@7d001000 {
        compatible = "arm,cortex-a15-gic", "arm,cortex-a9-gic";
        #interrupt-cells = <3>;
        #address-cells = <0>;
        interrupt-controller;
        reg = <0 0x7d001000 0 0x1000>,
              <0 0x7d002000 0 0x2000>,
              <0 0x7d004000 0 0x2000>,
              <0 0x7d006000 0 0x2000>;
        interrupts = <GIC_PPI 9 (GIC_CPU_MASK_SIMPLE(4) |
                     IRQ_TYPE_LEVEL_HIGH)>;
    };
```

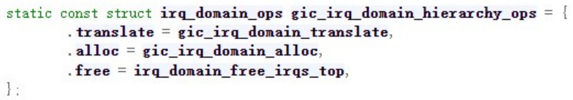
这个是gic-v2标准的控制器，对应drivers/irqchip/irq-gic.c，在gic_init_bases():



/ drivers / irqchip / irq-gic.c

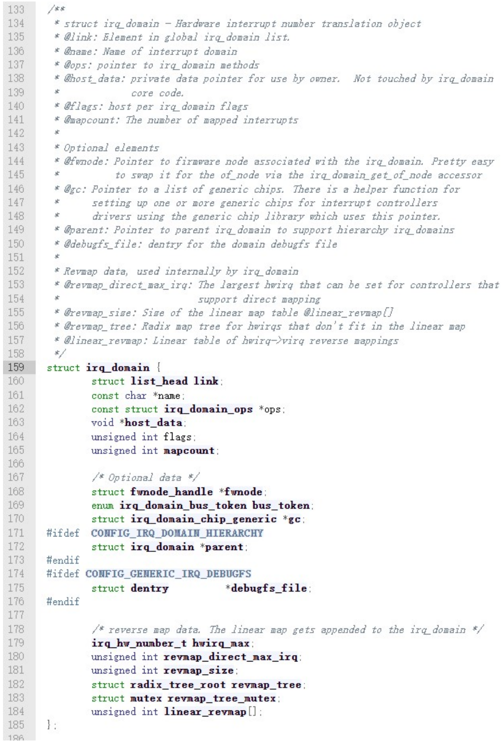
```
1089 #endif
1090 }
1091
1092 static int gic_init_bases(struct gic_chip_data *gic, int irq_start,
1093                           struct fwnode_handle *handle)
1094 {
1095         irq_hw_number_t hwirq_base;
1096         int gic_irqs, irq_base, ret;
1097
1098         if (IS_ENABLED(CONFIG_GIC_NON_BANKED) && gic->percpu_offset) {
1099                 /* Frankein-GIC without banked registers... */
1100                 unsigned int cpu;
1101
1102                 gic->dist_base.percpu_base = alloc_percpu(void __iomem *);
1103                 gic->cpu_base.percpu_base = alloc_percpu(void __iomem *);
1104                 if (WARN_ON(!gic->dist_base.percpu_base ||
1105                             !gic->cpu_base.percpu_base)) {
1106                         ret = -ENOMEM;
1107                         goto error;
1108                 }
1109
1110                 for_each_possible_cpu(cpu) {
1111                         u32 mpidr = cpu_logical_map(cpu);
1112                         u32 core_id = MPIDR_AFFINITY_LEVEL(mpidr, 0);
1113                         unsigned long offset = gic->percpu_offset * core_id;
1114                         *per_cpu_ptr(gic->dist_base.percpu_base, cpu) =
1115                                 gic->raw_dist_base + offset;
1116                         *per_cpu_ptr(gic->cpu_base.percpu_base, cpu) =
1117                                 gic->raw_cpu_base + offset;
1118                 }
1119
1120                 gic_set_base_accessor(gic, gic_get_percpu_base);
1121         } else {
1122                 /* Normal, sane GIC... */
1123                 WARN(gic->percpu_offset,
1124                      "GIC_NON_BANKED not enabled, ignoring %08x offset!",
1125                      gic->percpu_offset);
1126                 gic->dist_base.common_base = gic->raw_dist_base;
1127                 gic->cpu_base.common_base = gic->raw_cpu_base;
1128                 gic_set_base_accessor(gic, gic_get_common_base);
1129         }
1130
1131         /*
1132          * Find out how many interrupts are supported.
1133          * The GIC only supports up to 1020 interrupt sources.
1134          */
1135         gic_irqs = readl_relaxed(gic_data_dist_base(gic) + GIC_DIST_CTR) & 0x1f;
1136         gic_irqs = (gic_irqs + 1) * 32;
1137         if (gic_irqs > 1020)
1138                 gic_irqs = 1020;
1139         gic->gic_irqs = gic_irqs;
1140
1141         if (handle) {           /* DT/ACPI */
1142                 gic->domain = irq_domain_create_linear(handle, gic_irqs,
1143                                                        &gic_irq_domain_hierarchy_ops,
1144                                                        gic);
1145         } else {                /* Legacy support */
```

根据 1135 行寄存器可以读出中断源个数，作为入参调用 irq_domain_create_linear 注册一个 irq_domain 数据结构，另外传入 gic 自己实现的 irq_domain_ops：gic_irq_domain_hierarchy_ops



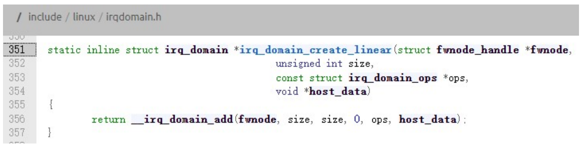
```
static const struct irq_domain_ops gic_irq_domain_hierarchy_ops = {
        .translate = gic_irq_domain_translate,
        .alloc = gic_irq_domain_alloc,
        .free = irq_domain_free_irqs_top,
};
```

这里面的 translate 方法，在后续映射的时候会用到。



```
/**
 * struct irq_domain - Hardware interrupt number translation object
 * @link: Element in global irq_domain list.
 * @name: Name of interrupt domain
 * @ops: pointer to irq_domain methods
 * @host_data: private data pointer for use by owner.  Not touched by irq_domain
 *             core code.
 * @flags: host per irq_domain flags
 * @mapcount: The number of mapped interrupts
 *
 * Optional elements
 * @fwnode: Pointer to firmware node associated with the irq_domain. Pretty easy
 *          to swap it for the of_node via the irq_domain_get_of_node accessor
 * @gc: Pointer to a list of generic chips. There is a helper function for
 *      setting up one or more generic chips for interrupt controllers
 *      drivers using the generic chip library which uses this pointer.
 * @parent: Pointer to parent irq_domain to support hierarchy irq_domains
 * @debugfs_file: dentry for the domain debugfs file
 *
 * Revmap data, used internally by irq_domain
 * @revmap_direct_max_irq: The largest hwirq that can be set for controllers that
 *                         support direct mapping
 * @revmap_size: Size of the linear map table @linear_revmap[]
 * @revmap_tree: Radix map tree for hwirqs that don't fit in the linear map
 * @linear_revmap: Linear table of hwirq->virq reverse mappings
 */
struct irq_domain {
	struct list_head link;
	const char *name;
	const struct irq_domain_ops *ops;
	void *host_data;
	unsigned int flags;
	unsigned int mapcount;

	/* Optional data */
	struct fwnode_handle *fwnode;
	enum irq_domain_bus_token bus_token;
	struct irq_domain_chip_generic *gc;
#ifdef	CONFIG_IRQ_DOMAIN_HIERARCHY
	struct irq_domain *parent;
#endif
#ifdef CONFIG_GENERIC_IRQ_DEBUGFS
	struct dentry		*debugfs_file;
#endif

	/* reverse map data. The linear map gets appended to the irq_domain */
	irq_hw_number_t hwirq_max;
	unsigned int revmap_direct_max_irq;
	unsigned int revmap_size;
	struct radix_tree_root revmap_tree;
	struct mutex revmap_tree_mutex;
	unsigned int linear_revmap[];
};
```

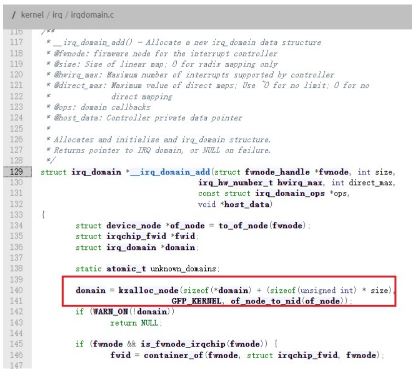
irq_domain_create_linear：



/ include / linux / irqdomain.h

```
static inline struct irq_domain *irq_domain_create_linear(struct fwnode_handle *fwnode,
					 unsigned int size,
					 const struct irq_domain_ops *ops,
					 void *host_data)
{
	return __irq_domain_add(fwnode, size, size, 0, ops, host_data);
}
```

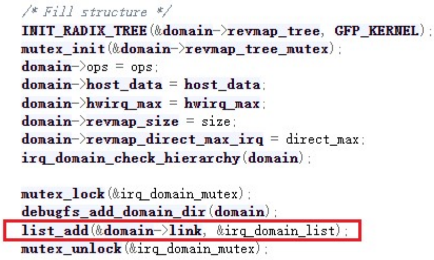
由__irq_domain_add 分配一个 irq_domain 结构体：



/ kernel / irq / irqdomain.c

```
/**
 * __irq_domain_add() - Allocate a new irq_domain data structure
 * @fwnode: firmware node for the interrupt controller
 * @size: Size of linear map; 0 for radix mapping only
 * @hwirq_max: Maximum number of interrupts supported by controller
 * @direct_max: Maximum value of direct maps; Use ~0 for no limit; 0 for no
 *              direct mapping
 * @ops: domain callbacks
 * @host_data: Controller private data pointer
 *
 * Allocates and initialize and irq_domain structure.
 * Returns pointer to IRQ domain, or NULL on failure.
 */
struct irq_domain *__irq_domain_add(struct fwnode_handle *fwnode, int size,
				    irq_hw_number_t hwirq_max, int direct_max,
				    const struct irq_domain_ops *ops,
				    void *host_data)
{
	struct device_node *of_node = to_of_node(fwnode);
	struct irqchip_fwid *fwid;
	struct irq_domain *domain;

	static atomic_t unknown_domains;

	domain = kzalloc_node(sizeof(*domain) + (sizeof(unsigned int) * size),
			      GFP_KERNEL, of_node_to_nid(of_node));
	if (WARN_ON(!domain))
		return NULL;

	if (fwnode && is_fwnode_irqchip(fwnode)) {
		fwid = container_of(fwnode, struct irqchip_fwid, fwnode);
```

在填充完内容物后，加入全局链表 irq_domain_list 中

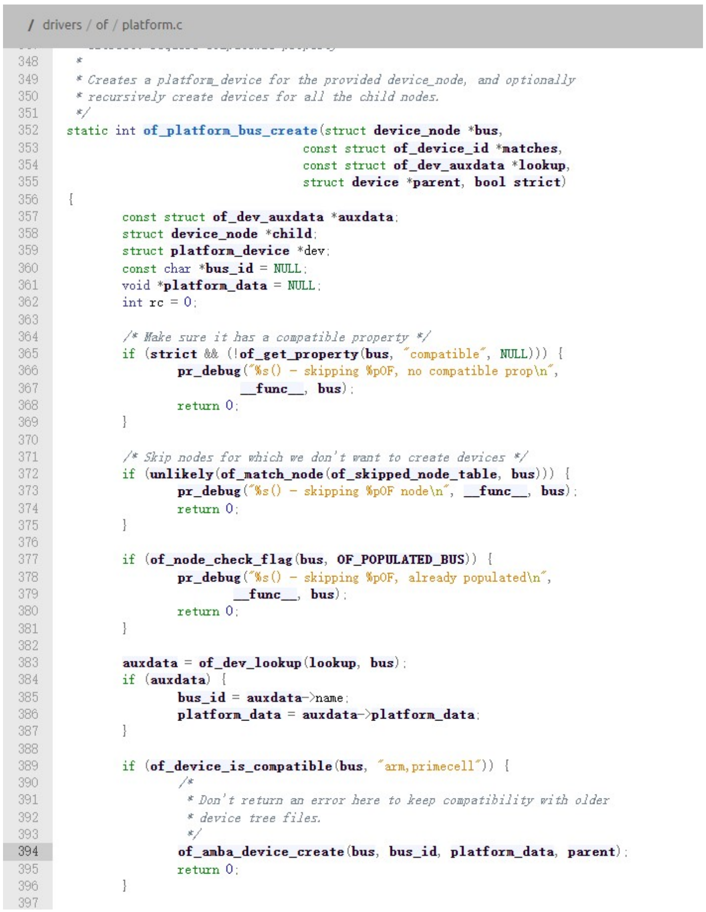


```
/* Fill structure */
INIT_RADIX_TREE(&domain->revmap_tree, GFP_KERNEL);
mutex_init(&domain->revmap_tree_mutex);
domain->ops = ops;
domain->host_data = host_data;
domain->hwirq_max = hwirq_max;
domain->revmap_size = size;
domain->revmap_direct_max_irq = direct_max;
irq_domain_check_hierarchy(domain);

mutex_lock(&irq_domain_mutex);
debugfs_add_domain_dir(domain);
list_add(&domain->link, &irq_domain_list);
mutex_unlock(&irq_domain_mutex);
```

接下来看下具体的某个中断（例如 uarto）的映射过程：

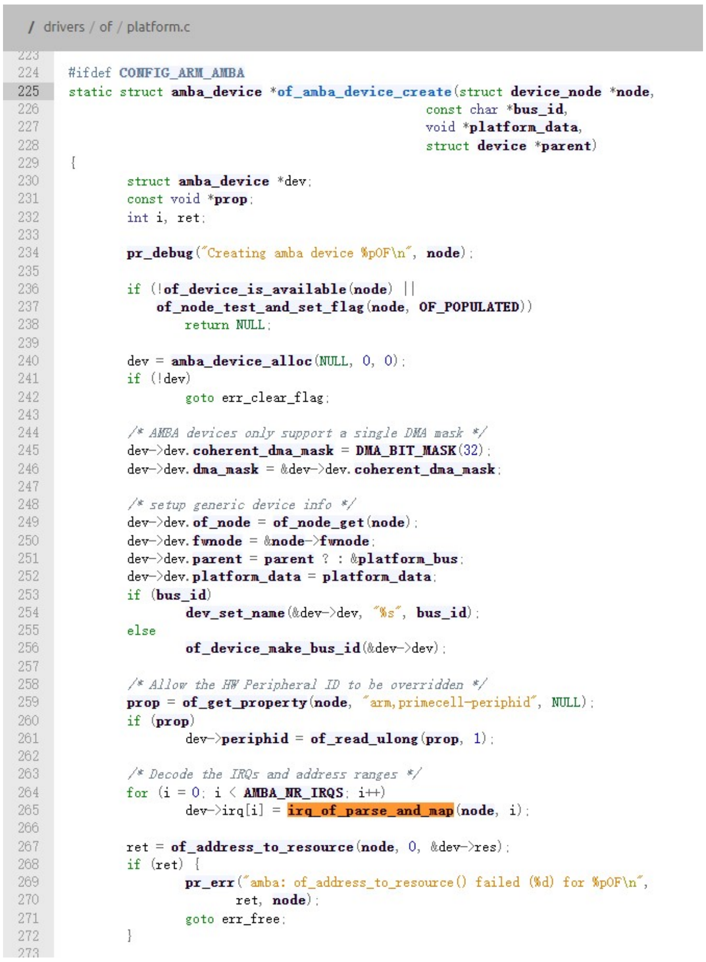
那么映射中断的内核 API 入口函数是 irq_of_parse_and_map。

调用点在：do_one_initcall()->of_platform_default_populate_init->of_platform_default_populate->of_platform_bus_create-> of_amba_device_create-> irq_of_parse_and_map



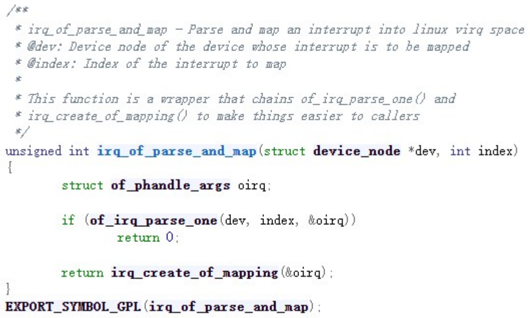
/ drivers / of / platform.c

```c
 *
 * Creates a platform_device for the provided device_node, and optionally
 * recursively create devices for all the child nodes.
 */
static int of_platform_bus_create(struct device_node *bus,
                                  const struct of_device_id *matches,
                                  const struct of_dev_auxdata *lookup,
                                  struct device *parent, bool strict)
{
        const struct of_dev_auxdata *auxdata;
        struct device_node *child;
        struct platform_device *dev;
        const char *bus_id = NULL;
        void *platform_data = NULL;
        int rc = 0;

        /* Make sure it has a compatible property */
        if (strict && (!of_get_property(bus, "compatible", NULL))) {
                pr_debug("%s() - skipping %pOF, no compatible prop\n",
                         __func__, bus);
                return 0;
        }

        /* Skip nodes for which we don't want to create devices */
        if (unlikely(of_match_node(of_skipped_node_table, bus))) {
                pr_debug("%s() - skipping %pOF node\n", __func__, bus);
                return 0;
        }

        if (of_node_check_flag(bus, OF_POPULATED_BUS)) {
                pr_debug("%s() - skipping %pOF, already populated\n",
                        __func__, bus);
                return 0;
        }

        auxdata = of_dev_lookup(lookup, bus);
        if (auxdata) {
                bus_id = auxdata->name;
                platform_data = auxdata->platform_data;
        }

        if (of_device_is_compatible(bus, "arm,primecell")) {
                /*
                 * Don't return an error here to keep compatibility with older
                 * device tree files.
                 */
                of_amba_device_create(bus, bus_id, platform_data, parent);
                return 0;
        }

```



/ drivers / of / platform.c

```
#ifdef CONFIG_ARM_AMBA
static struct amba_device *of_amba_device_create(struct device_node *node,
                                                 const char *bus_id,
                                                 void *platform_data,
                                                 struct device *parent)
{
        struct amba_device *dev;
        const void *prop;
        int i, ret;

        pr_debug("Creating amba device %pOF\n", node);

        if (!of_device_is_available(node) ||
            of_node_test_and_set_flag(node, OF_POPULATED))
                return NULL;

        dev = amba_device_alloc(NULL, 0, 0);
        if (!dev)
                goto err_clear_flag;

        /* AMBA devices only support a single DMA mask */
        dev->dev.coherent_dma_mask = DMA_BIT_MASK(32);
        dev->dev.dma_mask = &dev->dev.coherent_dma_mask;

        /* setup generic device info */
        dev->dev.of_node = of_node_get(node);
        dev->dev.fwnode = &node->fwnode;
        dev->dev.parent = parent ? : &platform_bus;
        dev->dev.platform_data = platform_data;
        if (bus_id)
                dev_set_name(&dev->dev, "%s", bus_id);
        else
                of_device_make_bus_id(&dev->dev);

        /* Allow the HW Peripheral ID to be overridden */
        prop = of_get_property(node, "arm,primecell-periphid", NULL);
        if (prop)
                dev->periphid = of_read_ulong(prop, 1);

        /* Decode the IRQs and address ranges */
        for (i = 0; i < AMBA_NR_IRQS; i++)
                dev->irq[i] = irq_of_parse_and_map(node, i);

        ret = of_address_to_resource(node, 0, &dev->res);
        if (ret) {
                pr_err("amba: of_address_to_resource() failed (%d) for %pOF\n",
                       ret, node);
                goto err_free;
        }
```

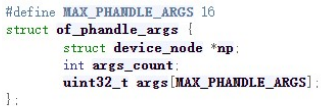


```
/**
 * irq_of_parse_and_map - Parse and map an interrupt into linux virq space
 * @dev: Device node of the device whose interrupt is to be mapped
 * @index: Index of the interrupt to map
 *
 * This function is a wrapper that chains of_irq_parse_one() and
 * irq_create_of_mapping() to make things easier to callers
 */
unsigned int irq_of_parse_and_map(struct device_node *dev, int index)
{
        struct of_phandle_args oirq;

        if (of_irq_parse_one(dev, index, &oirq))
                return 0;

        return irq_create_of_mapping(&oirq);
}
EXPORT_SYMBOL_GPL(irq_of_parse_and_map);
```

此处 irq_create_of_mapping 除了返回 dts 的硬件中断号给 dev-irq[i]以外，在内部会完成硬件-虚拟中断的映射。下面细看下过程：

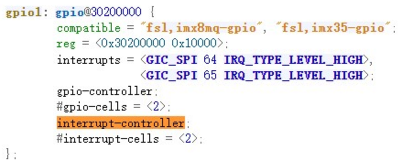
首先 irq_of_parse_and_map 会调用 AMBA_NR_IRQS 次数。#define AMBA_NR_IRQS 9

irq_of_parse_and_map 首先声明一个 of_phandle_args 的结构体 oirq：



```
#define MAX_PHANDLE_ARGS 16
struct of_phandle_args {
        struct device_node *np;
        int args_count;
        uint32_t args[MAX_PHANDLE_ARGS];
};
```

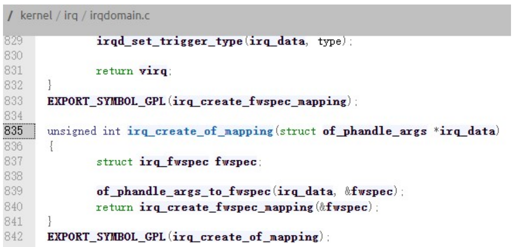
args[]会用来存放中断的信息，例如 imx8 的 gpio 中断定义是



```
gpio1: gpio@30200000 {
        compatible = "fsl,imx8mq-gpio", "fsl,imx35-gpio";
        reg = <0x30200000 0x10000>;
        interrupts = <GIC_SPI 64 IRQ_TYPE_LEVEL_HIGH>,
                     <GIC_SPI 65 IRQ_TYPE_LEVEL_HIGH>;
        gpio-controller;
        #gpio-cells = <2>;
        interrupt-controller;
        #interrupt-cells = <2>;
};
```

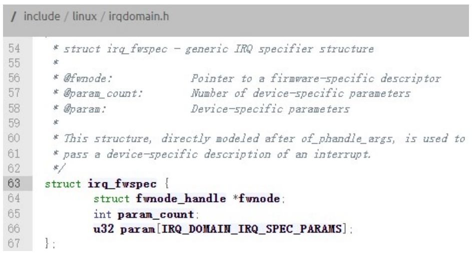
所以 args[0]就是 GIC_SPI,args[1]是 64 也就是硬件中断号，args[2]是 IRQ_TYPE_LEVEL_HIGH

继续看：



```c
	irqd_set_trigger_type(irq_data, type);

	return virq;
}
EXPORT_SYMBOL_GPL(irq_create_fwspec_mapping);

unsigned int irq_create_of_mapping(struct of_phandle_args *irq_data)
{
	struct irq_fwspec fwspec;

	of_phandle_args_to_fwspec(irq_data, &fwspec);
	return irq_create_fwspec_mapping(&fwspec);
}
EXPORT_SYMBOL_GPL(irq_create_of_mapping);
```

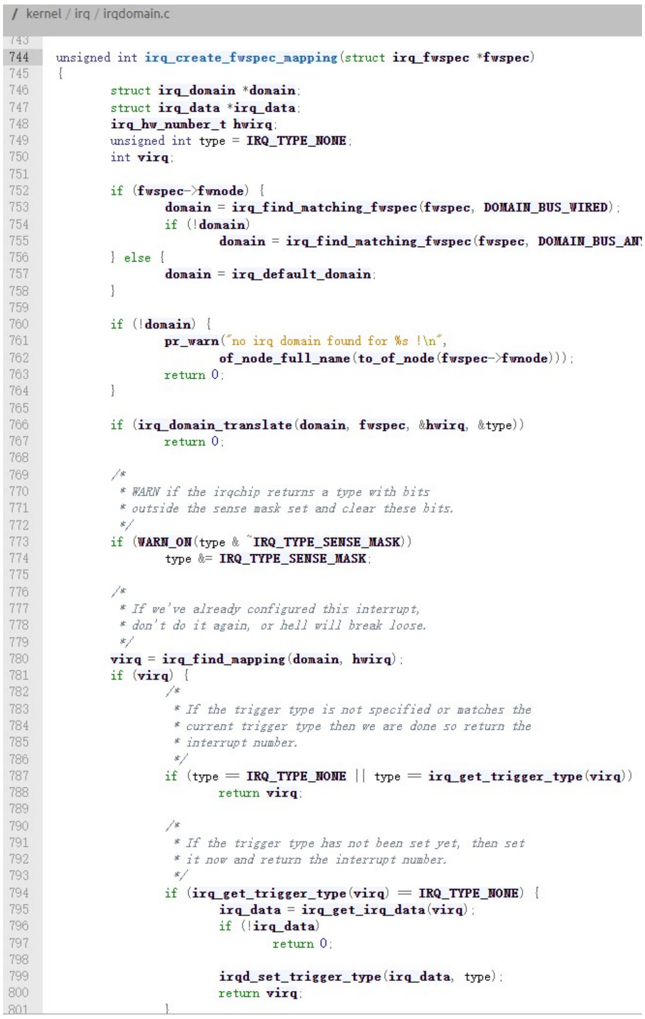
这里声明了一个 struct irq_fwspec fwspec;



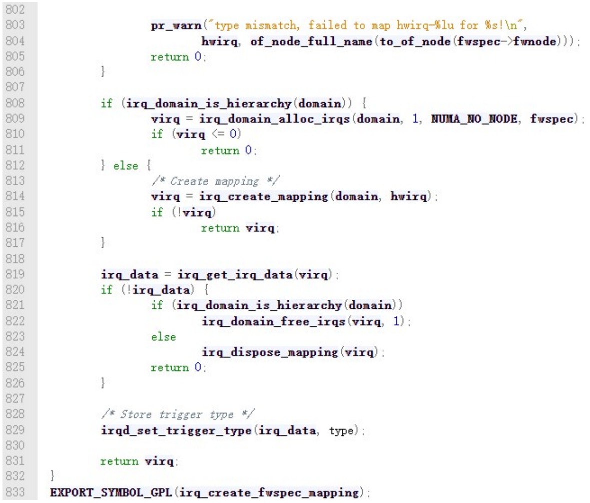
```c
 * struct irq_fwspec - generic IRQ specifier structure
 *
 * @fwnode:		Pointer to a firmware-specific descriptor
 * @param_count:	Number of device-specific parameters
 * @param:		Device-specific parameters
 *
 * This structure, directly modeled after of_phandle_args, is used to
 * pass a device-specific description of an interrupt.
 */
struct irq_fwspec {
	struct fwnode_handle *fwnode;
	int param_count;
	u32 param[IRQ_DOMAIN_IRQ_SPEC_PARAMS];
};
```

并且用 of_phandle_args_to_fwspec 把之前的 args 翻译到 irq_fwspec.

irq_create_fwspec_mapping：

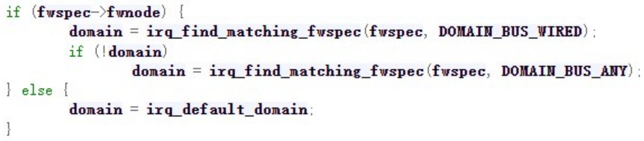


```c
unsigned int irq_create_fwspec_mapping(struct irq_fwspec *fwspec)
{
	struct irq_domain *domain;
	struct irq_data *irq_data;
	irq_hw_number_t hwirq;
	unsigned int type = IRQ_TYPE_NONE;
	int virq;

	if (fwspec->fwnode) {
		domain = irq_find_matching_fwspec(fwspec, DOMAIN_BUS_WIRED);
		if (!domain)
			domain = irq_find_matching_fwspec(fwspec, DOMAIN_BUS_AN
	} else {
		domain = irq_default_domain;
	}

	if (!domain) {
		pr_warn("no irq domain found for %s !\n",
			of_node_full_name(to_of_node(fwspec->fwnode)));
		return 0;
	}

	if (irq_domain_translate(domain, fwspec, &hwirq, &type))
		return 0;

	/*
	 * WARN if the irqchip returns a type with bits
	 * outside the sense mask set and clear these bits.
	 */
	if (WARN_ON(type & ~IRQ_TYPE_SENSE_MASK))
		type &= IRQ_TYPE_SENSE_MASK;

	/*
	 * If we've already configured this interrupt,
	 * don't do it again, or hell will break loose.
	 */
	virq = irq_find_mapping(domain, hwirq);
	if (virq) {
		/*
		 * If the trigger type is not specified or matches the
		 * current trigger type then we are done so return the
		 * interrupt number.
		 */
		if (type == IRQ_TYPE_NONE || type == irq_get_trigger_type(virq))
			return virq;

		/*
		 * If the trigger type has not been set yet, then set
		 * it now and return the interrupt number.
		 */
		if (irq_get_trigger_type(virq) == IRQ_TYPE_NONE) {
			irq_data = irq_get_irq_data(virq);
			if (!irq_data)
				return 0;

			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)) {
                virq = irq_domain_alloc_irqs(domain, 1, NUMA_NO_NODE, fwspec);
                if (virq <= 0)
                        return 0;
        } else {
                /* Create mapping */
                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;
}
EXPORT_SYMBOL_GPL(irq_create_fwspec_mapping);
```

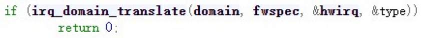
这其中先要找到对应的 irq_domain：



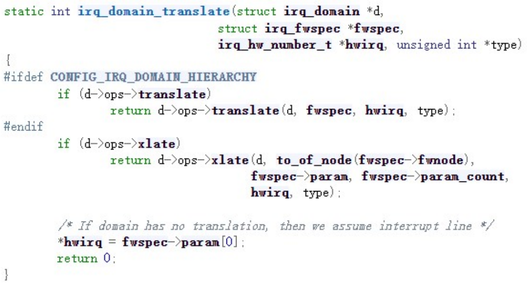
```
if (fwspec->fwnode) {
        domain = irq_find_matching_fwspec(fwspec, DOMAIN_BUS_WIRED);
        if (!domain)
                domain = irq_find_matching_fwspec(fwspec, DOMAIN_BUS_ANY);
} else {
        domain = irq_default_domain;
}
```

这个 domain 就是之前中断控制器注册的 irq_domain。

然后用 gic 的 translate 来翻译：

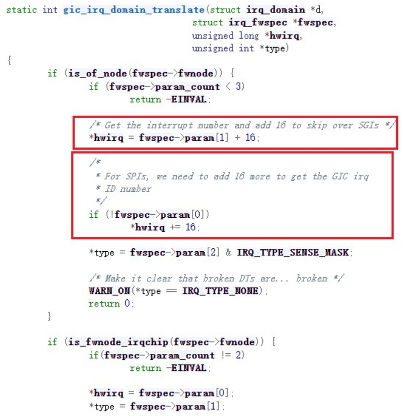


```
if (irq_domain_translate(domain, fwspec, &hwirq, &type))
        return 0;
```



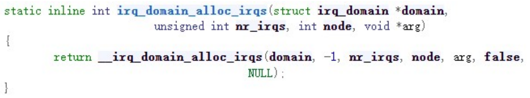
```
static int irq_domain_translate(struct irq_domain *d,
                                struct irq_fwspec *fwspec,
                                irq_hw_number_t *hwirq, unsigned int *type)
{
#ifdef CONFIG_IRQ_DOMAIN_HIERARCHY
        if (d->ops->translate)
                return d->ops->translate(d, fwspec, hwirq, type);
#endif
        if (d->ops->xlate)
                return d->ops->xlate(d, to_of_node(fwspec->fwnode),
                                     fwspec->param, fwspec->param_count,
                                     hwirq, type);

        /* If domain has no translation, then we assume interrupt line */
        *hwirq = fwspec->param[0];
        return 0;
}
```

对于 gic-v2 来说，0~31 是预留给 SGI 和 PPI 用的，所以外设的中断号在 translate 后要加上 32 的偏移。

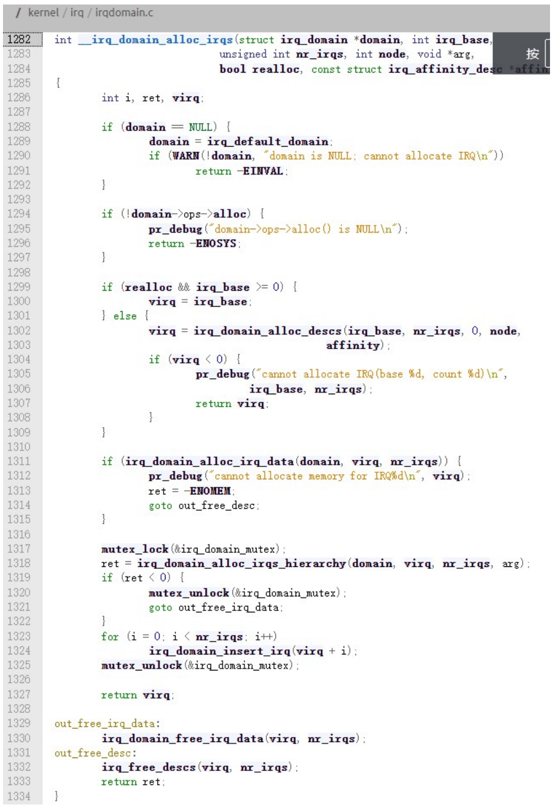


```
static int gic_irq_domain_translate(struct irq_domain *d,
                                    struct irq_fwspec *fwspec,
                                    unsigned long *hwirq,
                                    unsigned int *type)
{
        if (is_of_node(fwspec->fwnode)) {
                if (fwspec->param_count < 3)
                        return -EINVAL;

                /* Get the interrupt number and add 16 to skip over SGIs */
                *hwirq = fwspec->param[1] + 16;

                /*
                 * For SPIs, we need to add 16 more to get the GIC irq
                 * ID number
                 */
                if (!fwspec->param[0])
                        *hwirq += 16;

                *type = fwspec->param[2] & IRQ_TYPE_SENSE_MASK;

                /* Make it clear that broken DTs are... broken */
                WARN_ON(*type == IRQ_TYPE_NONE);
                return 0;
        }

        if (is_fwnode_irqchip(fwspec->fwnode)) {
                if(fwspec->param_count != 2)
                        return -EINVAL;

                *hwirq = fwspec->param[0];
                *type = fwspec->param[1];
```

接下来 of_irq_find_mapping 是尝试寻找现成的映射号的虚拟中断号，如果找到就配置下中断属性，我们属于找不到的情形，继续往下到了重点：irq_domain_alloc_irqs



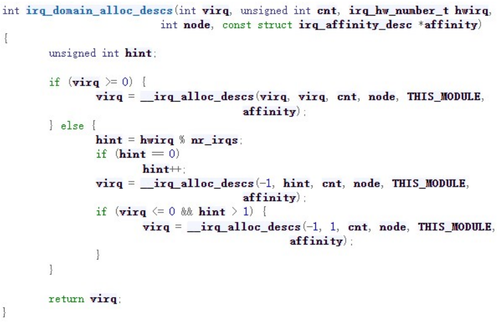
```
static inline int irq_domain_alloc_irqs(struct irq_domain *domain,
                        unsigned int nr_irqs, int node, void *arg)
{
        return __irq_domain_alloc_irqs(domain, -1, nr_irqs, node, arg, false,
                                       NULL);
}
```



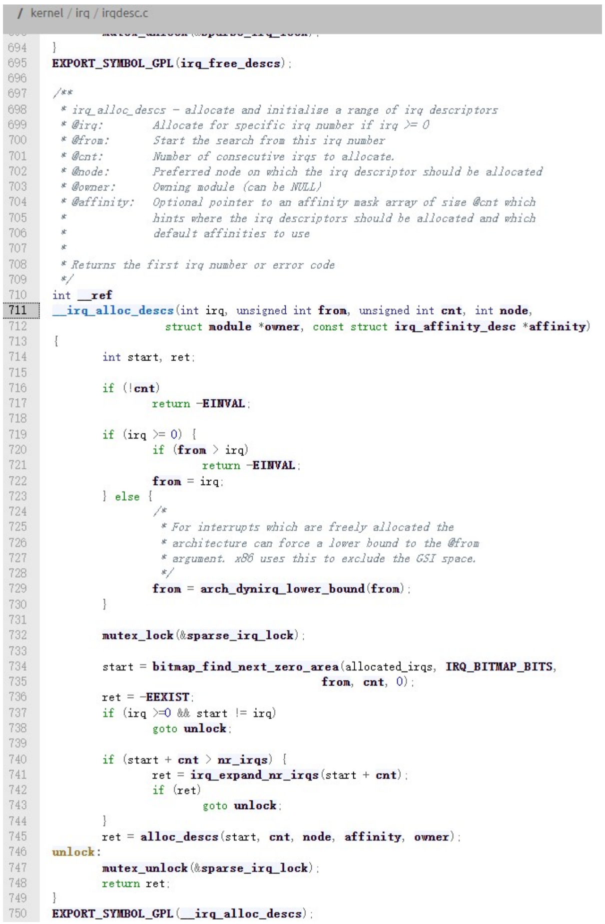
/ kernel / irq / irqdomain.c

```
1282  int __irq_domain_alloc_irqs(struct irq_domain *domain, int irq_base,
1283                              unsigned int nr_irqs, int node, void *arg,
1284                              bool realloc, const struct irq_affinity_desc *affin
1285  {
1286          int i, ret, virq;
1287
1288          if (domain == NULL) {
1289                  domain = irq_default_domain;
1290                  if (WARN(!domain, "domain is NULL; cannot allocate IRQ\n"))
1291                          return -EINVAL;
1292          }
1293
1294          if (!domain->ops->alloc) {
1295                  pr_debug("domain->ops->alloc() is NULL\n");
1296                  return -ENOSYS;
1297          }
1298
1299          if (realloc && irq_base >= 0) {
1300                  virq = irq_base;
1301          } else {
1302                  virq = irq_domain_alloc_descs(irq_base, nr_irqs, 0, node,
1303                                                affinity);
1304                  if (virq < 0) {
1305                          pr_debug("cannot allocate IRQ(base %d, count %d)\n",
1306                                   irq_base, nr_irqs);
1307                          return virq;
1308                  }
1309          }
1310
1311          if (irq_domain_alloc_irq_data(domain, virq, nr_irqs)) {
1312                  pr_debug("cannot allocate memory for IRQ%d\n", virq);
1313                  ret = -ENOMEM;
1314                  goto out_free_desc;
1315          }
1316
1317          mutex_lock(&irq_domain_mutex);
1318          ret = irq_domain_alloc_irqs_hierarchy(domain, virq, nr_irqs, arg);
1319          if (ret < 0) {
1320                  mutex_unlock(&irq_domain_mutex);
1321                  goto out_free_irq_data;
1322          }
1323          for (i = 0; i < nr_irqs; i++)
1324                  irq_domain_insert_irq(virq + i);
1325          mutex_unlock(&irq_domain_mutex);
1326
1327          return virq;
1328
1329  out_free_irq_data:
1330          irq_domain_free_irq_data(virq, nr_irqs);
1331  out_free_desc:
1332          irq_free_descs(virq, nr_irqs);
1333          return ret;
1334  }
```

看 1302 行 irq_domain_alloc_descs，也就是__irq_alloc_descs



```
int irq_domain_alloc_descs(int virq, unsigned int cnt, irq_hw_number_t hwirq,
                           int node, const struct irq_affinity_desc *affinity)
{
        unsigned int hint;

        if (virq >= 0) {
                virq = __irq_alloc_descs(virq, virq, cnt, node, THIS_MODULE,
                                         affinity);
        } else {
                hint = hwirq % nr_irqs;
                if (hint == 0)
                        hint++;
                virq = __irq_alloc_descs(-1, hint, cnt, node, THIS_MODULE,
                                         affinity);
                if (virq <= 0 && hint > 1) {
                        virq = __irq_alloc_descs(-1, 1, cnt, node, THIS_MODULE,
                                                 affinity);
                }
        }

        return virq;
}
```



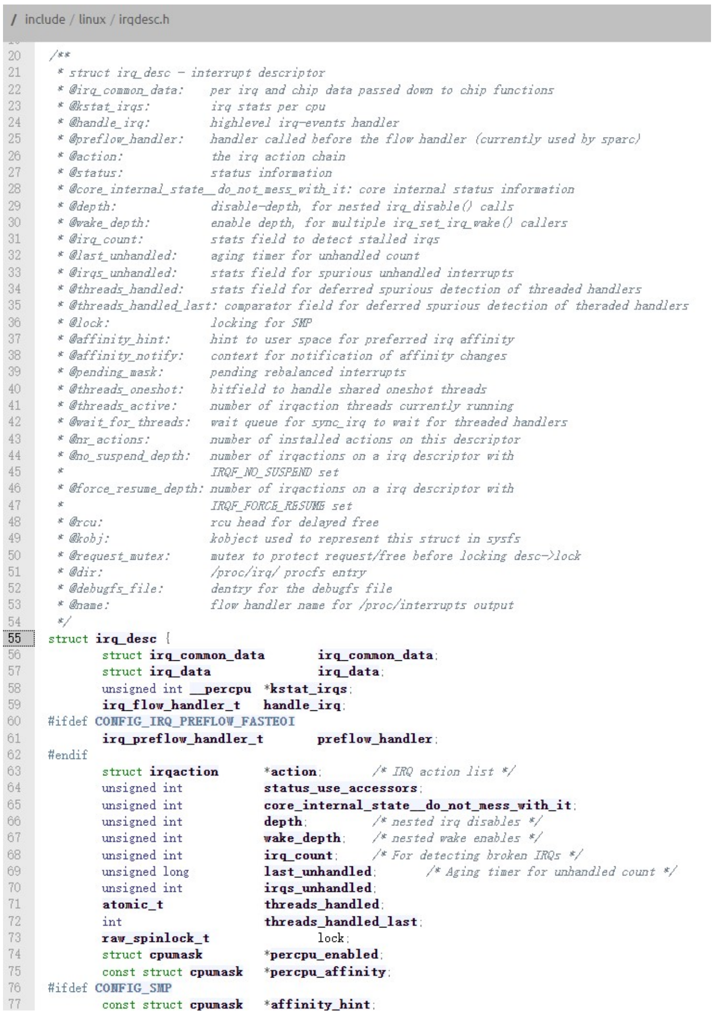
/ kernel / irq / irqdesc.c

```
694 }
695 EXPORT_SYMBOL_GPL(irq_free_descs);
696 
697 /**
698  * irq_alloc_descs - allocate and initialize a range of irq descriptors
699  * @irq:        Allocate for specific irq number if irq >= 0
700  * @from:       Start the search from this irq number
701  * @cnt:        Number of consecutive irqs to allocate.
702  * @node:       Preferred node on which the irq descriptor should be allocated
703  * @owner:      Owning module (can be NULL)
704  * @affinity:   Optional pointer to an affinity mask array of size @cnt which
705  *              hints where the irq descriptors should be allocated and which
706  *              default affinities to use
707  *
708  * Returns the first irq number or error code
709  */
710 int __ref
711 __irq_alloc_descs(int irq, unsigned int from, unsigned int cnt, int node,
712                   struct module *owner, const struct irq_affinity_desc *affinity)
713 {
714         int start, ret;
715 
716         if (!cnt)
717                 return -EINVAL;
718 
719         if (irq >= 0) {
720                 if (from > irq)
721                         return -EINVAL;
722                 from = irq;
723         } else {
724                 /*
725                  * For interrupts which are freely allocated the
726                  * architecture can force a lower bound to the @from
727                  * argument. x86 uses this to exclude the GSI space.
728                  */
729                 from = arch_dynirq_lower_bound(from);
730         }
731 
732         mutex_lock(&sparse_irq_lock);
733 
734         start = bitmap_find_next_zero_area(allocated_irqs, IRQ_BITMAP_BITS,
735                                            from, cnt, 0);
736         ret = -EEXIST;
737         if (irq >=0 && start != irq)
738                 goto unlock;
739 
740         if (start + cnt > nr_irqs) {
741                 ret = irq_expand_nr_irqs(start + cnt);
742                 if (ret)
743                         goto unlock;
744         }
745         ret = alloc_descs(start, cnt, node, affinity, owner);
746 unlock:
747         mutex_unlock(&sparse_irq_lock);
748         return ret;
749 }
750 EXPORT_SYMBOL_GPL(__irq_alloc_descs);
```

734 行 bitmap_find_next_zero_area 是在 allocated_irqs 全局变量位图中查找第一个包含连续 cnt 个 0 的位域。allocated_irqs 会保存所有分配过的虚拟中断号。

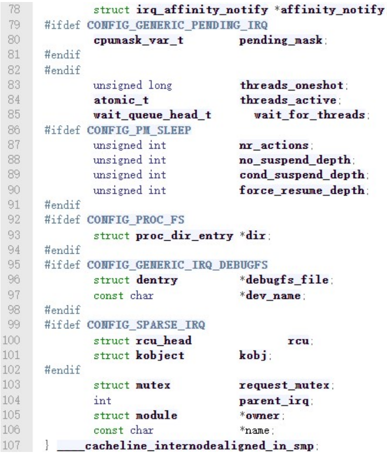
static DECLARE_BITMAP(allocated_irqs, IRQ_BITMAP_BITS);

745 行 alloc_descs 用于根据刚刚分得的 virq，分配一个 irq_desc 结构，即中断描述符。



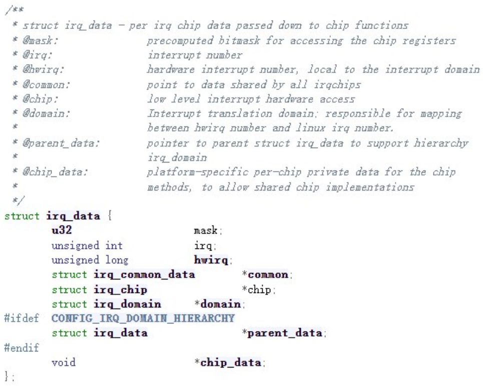
/ include / linux / irqdesc.h

```c
/**
 * struct irq_desc - interrupt descriptor
 * @irq_common_data:   per irq and chip data passed down to chip functions
 * @kstat_irqs:        irq stats per cpu
 * @handle_irq:        highlevel irq-events handler
 * @preflow_handler:   handler called before the flow handler (currently used by sparc)
 * @action:            the irq action chain
 * @status:            status information
 * @core_internal_state__do_not_mess_with_it: core internal status information
 * @depth:             disable-depth, for nested irq_disable() calls
 * @wake_depth:        enable depth, for multiple irq_set_irq_wake() callers
 * @irq_count:         stats field to detect stalled irqs
 * @last_unhandled:    aging timer for unhandled count
 * @irqs_unhandled:    stats field for spurious unhandled interrupts
 * @threads_handled:   stats field for deferred spurious detection of threaded handlers
 * @threads_handled_last: comparator field for deferred spurious detection of theraded handlers
 * @lock:              locking for SMP
 * @affinity_hint:     hint to user space for preferred irq affinity
 * @affinity_notify:   context for notification of affinity changes
 * @pending_mask:      pending rebalanced interrupts
 * @threads_oneshot:   bitfield to handle shared oneshot threads
 * @threads_active:    number of irqaction threads currently running
 * @wait_for_threads:  wait queue for sync_irq to wait for threaded handlers
 * @nr_actions:        number of installed actions on this descriptor
 * @no_suspend_depth:  number of irqactions on a irq descriptor with
 *                     IRQF_NO_SUSPEND set
 * @force_resume_depth: number of irqactions on a irq descriptor with
 *                     IRQF_FORCE_RESUME set
 * @rcu:               rcu head for delayed free
 * @kobj:              kobject used to represent this struct in sysfs
 * @request_mutex:     mutex to protect request/free before locking desc->lock
 * @dir:               /proc/irq/ procfs entry
 * @debugfs_file:      dentry for the debugfs file
 * @name:              flow handler name for /proc/interrupts output
 */
struct irq_desc {
	struct irq_common_data	irq_common_data;
	struct irq_data		irq_data;
	unsigned int __percpu	*kstat_irqs;
	irq_flow_handler_t	handle_irq;
#ifdef CONFIG_IRQ_PREFLOW_FASTEOI
	irq_preflow_handler_t	preflow_handler;
#endif
	struct irqaction	*action;	/* IRQ action list */
	unsigned int		status_use_accessors;
	unsigned int		core_internal_state__do_not_mess_with_it;
	unsigned int		depth;		/* nested irq disables */
	unsigned int		wake_depth;	/* nested wake enables */
	unsigned int		irq_count;	/* For detecting broken IRQs */
	unsigned long		last_unhandled;	/* Aging timer for unhandled count */
	unsigned int		irqs_unhandled;
	atomic_t		threads_handled;
	int			threads_handled_last;
	raw_spinlock_t		lock;
	struct cpumask		*percpu_enabled;
	const struct cpumask	*percpu_affinity;
#ifdef CONFIG_SMP
	const struct cpumask	*affinity_hint;
```



```c
	struct irq_affinity_notify *affinity_notify;
#ifdef CONFIG_GENERIC_PENDING_IRQ
	cpumask_var_t		pending_mask;
#endif
#endif
	unsigned long		threads_oneshot;
	atomic_t		threads_active;
	wait_queue_head_t	wait_for_threads;
#ifdef CONFIG_PM_SLEEP
	unsigned int		nr_actions;
	unsigned int		no_suspend_depth;
	unsigned int		cond_suspend_depth;
	unsigned int		force_resume_depth;
#endif
#ifdef CONFIG_PROC_FS
	struct proc_dir_entry	*dir;
#endif
#ifdef CONFIG_GENERIC_IRQ_DEBUGFS
	struct dentry		*debugfs_file;
	const char		*dev_name;
#endif
#ifdef CONFIG_SPARSE_IRQ
	struct rcu_head		rcu;
	struct kobject		kobj;
#endif
	struct mutex		request_mutex;
	int			parent_irq;
	struct module		*owner;
	const char		*name;
} ____cacheline_internodealigned_in_smp;
```

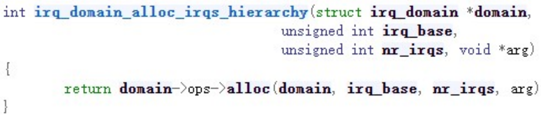
重点看下第二个成员 `struct irq_data		irq_data;`



```
/**
 * struct irq_data - per irq chip data passed down to chip functions
 * @mask:              precomputed bitmask for accessing the chip registers
 * @irq:               interrupt number
 * @hwirq:             hardware interrupt number, local to the interrupt domain
 * @common:            point to data shared by all irqchips
 * @chip:              low level interrupt hardware access
 * @domain:            Interrupt translation domain; responsible for mapping
 *                     between hwirq number and linux irq number.
 * @parent_data:       pointer to parent struct irq_data to support hierarchy
 *                     irq_domain
 * @chip_data:         platform-specific per-chip private data for the chip
 *                     methods, to allow shared chip implementations
 */
struct irq_data {
        u32                     mask;
        unsigned int            irq;
        unsigned long           hwirq;
        struct irq_common_data          *common;
        struct irq_chip                 *chip;
        struct irq_domain       *domain;
#ifdef  CONFIG_IRQ_DOMAIN_HIERARCHY
        struct irq_data                 *parent_data;
#endif
        void                    *chip_data;
};
```

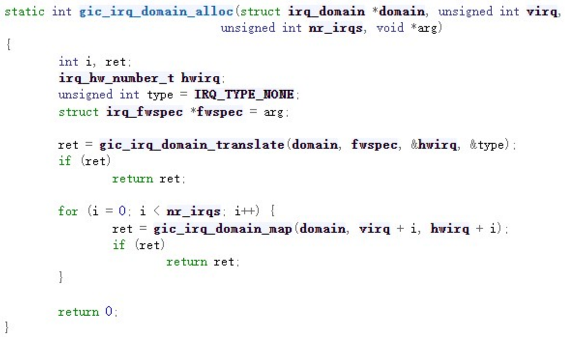
这里面 irq 是会被填上虚拟中断号，hwirq 后续要填硬件中断号。

回到__irq_domain_alloc_irqs 后续的 irq_domain_alloc_irqs_hierarchy，



```
int irq_domain_alloc_irqs_hierarchy(struct irq_domain *domain,
                                    unsigned int irq_base,
                                    unsigned int nr_irqs, void *arg)
{
        return domain->ops->alloc(domain, irq_base, nr_irqs, arg);
}
```

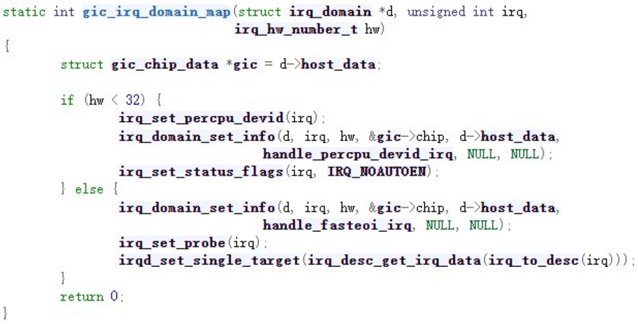
即 gic 的 alloc 回调函数：



```
static int gic_irq_domain_alloc(struct irq_domain *domain, unsigned int virq,
                                unsigned int nr_irqs, void *arg)
{
        int i, ret;
        irq_hw_number_t hwirq;
        unsigned int type = IRQ_TYPE_NONE;
        struct irq_fwspec *fwspec = arg;

        ret = gic_irq_domain_translate(domain, fwspec, &hwirq, &type);
        if (ret)
                return ret;

        for (i = 0; i < nr_irqs; i++) {
                ret = gic_irq_domain_map(domain, virq + i, hwirq + i);
                if (ret)
                        return ret;
        }

        return 0;
}
```

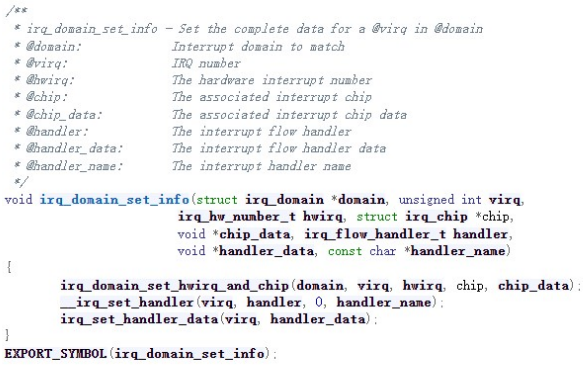
先通过 translate 把硬件中断号放入 hwirq 后，用 gic_irq_domain_map 把 hwirq 放入 irq_desc 中。

gic_irq_domain_map 细看：



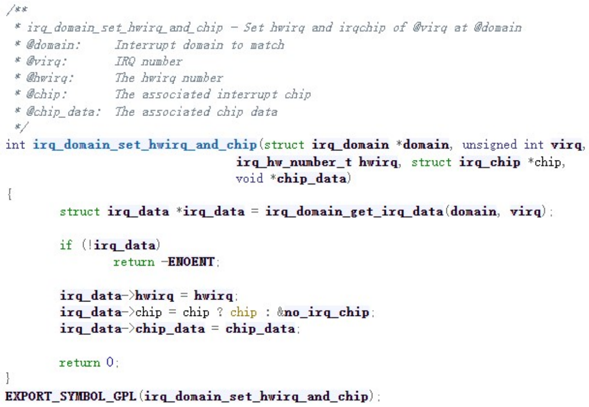
```
static int gic_irq_domain_map(struct irq_domain *d, unsigned int irq,
                              irq_hw_number_t hw)
{
        struct gic_chip_data *gic = d->host_data;

        if (hw < 32) {
                irq_set_percpu_devid(irq);
                irq_domain_set_info(d, irq, hw, &gic->chip, d->host_data,
                                    handle_percpu_devid_irq, NULL, NULL);
                irq_set_status_flags(irq, IRQ_NOAUTOEN);
        } else {
                irq_domain_set_info(d, irq, hw, &gic->chip, d->host_data,
                                    handle_fasteoi_irq, NULL, NULL);
                irq_set_probe(irq);
                irqd_set_single_target(irq_desc_get_irq_data(irq_to_desc(irq)));
        }
        return 0;
}
```

其使用 irq_domain_set_info 设置重要的参数：



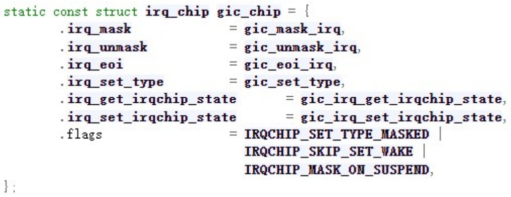
```
/**
 * irq_domain_set_info - Set the complete data for a @virq in @domain
 * @domain:             Interrupt domain to match
 * @virq:               IRQ number
 * @hwirq:              The hardware interrupt number
 * @chip:               The associated interrupt chip
 * @chip_data:          The associated interrupt chip data
 * @handler:            The interrupt flow handler
 * @handler_data:       The interrupt flow handler data
 * @handler_name:       The interrupt handler name
 */
void irq_domain_set_info(struct irq_domain *domain, unsigned int virq,
                         irq_hw_number_t hwirq, struct irq_chip *chip,
                         void *chip_data, irq_flow_handler_t handler,
                         void *handler_data, const char *handler_name)
{
        irq_domain_set_hwirq_and_chip(domain, virq, hwirq, chip, chip_data);
        __irq_set_handler(virq, handler, 0, handler_name);
        irq_set_handler_data(virq, handler_data);
}
EXPORT_SYMBOL(irq_domain_set_info);
```

irq_domain_set_hwirq_and_chip 通过 virq 拿到 irq_data，然后把 hwirq 设置进去。



```
/**
 * irq_domain_set_hwirq_and_chip - Set hwirq and irqchip of @virq at @domain
 * @domain:     Interrupt domain to match
 * @virq:       IRQ number
 * @hwirq:      The hwirq number
 * @chip:       The associated interrupt chip
 * @chip_data:  The associated chip data
 */
int irq_domain_set_hwirq_and_chip(struct irq_domain *domain, unsigned int virq,
                                  irq_hw_number_t hwirq, struct irq_chip *chip,
                                  void *chip_data)
{
        struct irq_data *irq_data = irq_domain_get_irq_data(domain, virq);

        if (!irq_data)
                return -ENOENT;

        irq_data->hwirq = hwirq;
        irq_data->chip = chip ? chip : &no_irq_chip;
        irq_data->chip_data = chip_data;

        return 0;
}
EXPORT_SYMBOL_GPL(irq_domain_set_hwirq_and_chip);
```

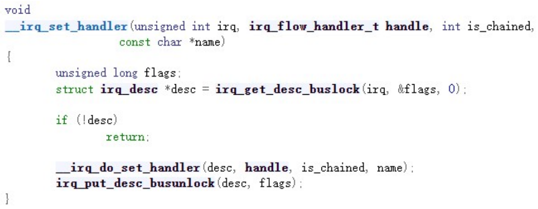
并且会把 chip 也设置到 irq_data->chip，这个 chip 就是 drivers/irqchip/irq-gic.c 里面声明的 gic 中断控制器的 irq_chip：



```
static const struct irq_chip gic_chip = {
        .irq_mask               = gic_mask_irq,
        .irq_unmask             = gic_unmask_irq,
        .irq_eoi                = gic_eoi_irq,
        .irq_set_type           = gic_set_type,
        .irq_get_irqchip_state        = gic_irq_get_irqchip_state,
        .irq_set_irqchip_state        = gic_irq_set_irqchip_state,
        .flags                  = IRQCHIP_SET_TYPE_MASKED |
                                  IRQCHIP_SKIP_SET_WAKE |
                                  IRQCHIP_MASK_ON_SUSPEND,
};
```

这样就把具体的某个中断和中断控制器也挂钩了。

__irq_set_handler 则设置中断处理函数的总入口，对于 SPI 是 handle_fasteoi_irq()



```
void
__irq_set_handler(unsigned int irq, irq_flow_handler_t handle, int is_chained,
                  const char *name)
{
        unsigned long flags;
        struct irq_desc *desc = irq_get_desc_buslock(irq, &flags, 0);

        if (!desc)
                return;

        __irq_do_set_handler(desc, handle, is_chained, name);
        irq_put_desc_busunlock(desc, flags);
}
```