# The irq_domain interrupt number mapping library

The current design of the Linux kernel uses a single large number space where each separate IRQ source is assigned a different number. This is simple when there is only one interrupt controller, but in systems with multiple interrupt controllers the kernel must ensure that each one gets assigned non-overlapping allocations of Linux IRQ numbers.

The number of interrupt controllers registered as unique irqchips show a rising tendency: for example subdrivers of different kinds such as GPIO controllers avoid reimplementing identical callback mechanisms as the IRQ core system by modelling their interrupt handlers as irqchips, i.e. in effect cascading interrupt controllers.

Here the interrupt number loose all kind of correspondence to hardware interrupt numbers: whereas in the past, IRQ numbers could be chosen so they matched the hardware IRQ line into the root interrupt controller (i.e. the component actually fireing the interrupt line to the CPU) nowadays this number is just a number.

For this reason we need a mechanism to separate controller-local interrupt numbers, called hardware irq's, from Linux IRQ numbers.

The irq_alloc_desc*() and irq_free_desc*() APIs provide allocation of irq numbers, but they don't provide any support for reverse mapping of the controller-local IRQ (hwirq) number into the Linux IRQ number space.

The irq_domain library adds mapping between hwirq and IRQ numbers on top of the irq_alloc_desc*() API. An irq_domain to manage mapping is preferred over interrupt controller drivers open coding their own reverse mapping scheme.

irq_domain also implements translation from an abstract irq_fwspec structure to hwirq numbers (Device Tree and ACPI GSI so far), and can be easily extended to support other IRQ topology data sources.

## irq_domain usage

An interrupt controller driver creates and registers an irq_domain by calling one of the irq_domain_add_*() or irq_domain_create_*() functions (each mapping method has a different allocator function, more on that later). The function will return a pointer to the irq_domain on success. The caller must provide the allocator function with an irq_domain_ops structure.

In most cases, the irq_domain will begin empty without any mappings between hwirq and IRQ numbers. Mappings are added to the irq_domain by calling irq_create_mapping() which accepts the irq_domain and a hwirq number as arguments. If a mapping for the hwirq doesn't already exist then it will allocate a new Linux irq_desc, associate it with the hwirq, and call the .map() callback so the driver can perform any required hardware setup.

Once a mapping has been established, it can be retrieved or used via a variety of methods:

- irq_resolve_mapping() returns a pointer to the irq_desc structure for a given domain and hwirq number, and NULL if there was no mapping.
- irq_find_mapping() returns a Linux IRQ number for a given domain and hwirq number, and 0 if there was no mapping
- irq_linear_revmap() is now identical to irq_find_mapping(), and is deprecated
- generic_handle_domain_irq() handles an interrupt described by a domain and a hwirq number

Note that irq domain lookups must happen in contexts that are compatible with a RCU read-side critical section.

The irq_create_mapping() function must be called *atleast once* before any call to irq_find_mapping(), lest the descriptor will not be allocated.

If the driver has the Linux IRQ number or the irq_data pointer, and needs to know the associated hwirq number (such as in the irq_chip callbacks) then it can be directly obtained from irq_data->hwirq.

## Types of irq_domain mappings

There are several mechanisms available for reverse mapping from hwirq to Linux irq, and each mechanism uses a different allocation function. Which reverse map type should be used depends on the use case. Each of the reverse map types are described below:

### Linear

```
irq_domain_add_linear()
irq_domain_create_linear()
```

The linear reverse map maintains a fixed size table indexed by the hwirq number. When a hwirq is mapped, an irq_desc is allocated for the hwirq, and the IRQ number is stored in the table.

The Linear map is a good choice when the maximum number of hwirqs is fixed and a relatively small number (~ < 256). The advantages of this map are fixed time lookup for IRQ numbers, and irq_descs are only allocated for in-use IRQs. The disadvantage is that the table must be as large as the largest possible hwirq number.

irq_domain_add_linear() and irq_domain_create_linear() are functionally equivalent, except for the first argument is different - the former accepts an Open Firmware specific 'struct device_node', while the latter accepts a more general abstraction 'struct fwnode_handle'.

The majority of drivers should use the linear map.

### Tree

```
irq_domain_add_tree()
irq_domain_create_tree()
```

The irq_domain maintains a radix tree map from hwirq numbers to Linux IRQs. When an hwirq is mapped, an irq_desc is allocated and the hwirq is used as the lookup key for the radix tree.

The tree map is a good choice if the hwirq number can be very large since it doesn't need to allocate a table as large as the largest hwirq number. The disadvantage is that hwirq to IRQ number lookup is dependent on how many entries are in the table.

irq_domain_add_tree() and irq_domain_create_tree() are functionally equivalent, except for the first argument is different - the former accepts an Open Firmware specific 'struct device_node', while the latter accepts a more general abstraction 'struct fwnode_handle'.

Very few drivers should need this mapping.

### No Map

```
irq_domain_add_nomap()
```

The No Map mapping is to be used when the hwirq number is programmable in the hardware. In this case it is best to program the Linux IRQ number into the hardware itself so that no mapping is required. Calling irq_create_direct_mapping() will allocate a Linux IRQ number and call the .map() callback so that driver can program the Linux IRQ number into the hardware.

Most drivers cannot use this mapping, and it is now gated on the CONFIG_IRQ_DOMAIN_NOMAP option. Please refrain from introducing new users of this API.

### Legacy

```
irq_domain_add_simple()
irq_domain_add_legacy()
irq_domain_create_simple()
irq_domain_create_legacy()
```

The Legacy mapping is a special case for drivers that already have a range of irq_descs allocated for the hwirqs. It is used when the driver cannot be immediately converted to use the linear mapping. For example, many embedded system board support files use a set of #defines for IRQ numbers that are passed to struct device registrations. In that case the Linux IRQ numbers cannot be dynamically assigned and the legacy mapping should be used.

As the name implies, the *_legacy() functions are deprecated and only exist to ease the support of ancient platforms. No new users should be added. Same goes for the *_simple() functions when their use results in the legacy behaviour.

The legacy map assumes a contiguous range of IRQ numbers has already been allocated for the controller and that the IRQ number can be calculated by adding a fixed offset to the hwirq number, and visa-versa. The disadvantage is that it requires the interrupt controller to manage IRQ allocations and it requires an irq_desc to be allocated for every hwirq, even if it is unused.

The legacy map should only be used if fixed IRQ mappings must be supported. For example, ISA controllers would use the legacy map for mapping Linux IRQs 0-15 so that existing ISA drivers get the correct IRQ numbers.

Most users of legacy mappings should use irq_domain_add_simple() or irq_domain_create_simple() which will use a legacy domain only if an IRQ range is supplied by the system and will otherwise use a linear domain mapping. The semantics of this call are such that if an IRQ range is specified then descriptors will be allocated on-the-fly for it, and if no range is specified it will fall through to irq_domain_add_linear() or irq_domain_create_linear() which means *no* irq descriptors will be allocated.

A typical use case for simple domains is where an irqchip provider is supporting both dynamic and static IRQ assignments.

In order to avoid ending up in a situation where a linear domain is used and no descriptor gets allocated it is very important to make sure that the driver using the simple domain call irq_create_mapping() before any irq_find_mapping() since the latter will actually work for the static IRQ assignment case.

irq_domain_add_simple() and irq_domain_create_simple() as well as irq_domain_add_legacy() and irq_domain_create_legacy() are functionally equivalent, except for the first argument is different - the former accepts an Open Firmware specific 'struct device_node', while the latter accepts a more general abstraction 'struct fwnode_handle'.

### Hierarchy IRQ domain

On some architectures, there may be multiple interrupt controllers involved in delivering an interrupt from the device to the target CPU. Let's look at a typical interrupt delivering path on x86 platforms:

```
Device --> IOAPIC -> Interrupt remapping Controller -> Local APIC -> CPU
```

There are three interrupt controllers involved:

1. IOAPIC controller
2. Interrupt remapping controller

3. Local APIC controller

To support such a hardware topology and make software architecture match hardware architecture, an irq_domain data structure is built for each interrupt controller and those irq_domains are organized into hierarchy. When building irq_domain hierarchy, the irq_domain near to the device is child and the irq_domain near to CPU is parent. So a hierarchy structure as below will be built for the example above:

```
CPU Vector irq_domain (root irq_domain to manage CPU vectors)
        ^
        |
Interrupt Remapping irq_domain (manage irq_remapping entries)
        ^
        |
IOAPIC irq_domain (manage IOAPIC delivery entries/pins)
```

There are four major interfaces to use hierarchy irq_domain:

1. irq_domain_alloc_irqs(): allocate IRQ descriptors and interrupt controller related resources to deliver these interrupts.
2. irq_domain_free_irqs(): free IRQ descriptors and interrupt controller related resources associated with these interrupts.
3. irq_domain_activate_irq(): activate interrupt controller hardware to deliver the interrupt.
4. irq_domain_deactivate_irq(): deactivate interrupt controller hardware to stop delivering the interrupt.

Following changes are needed to support hierarchy irq_domain:

1. a new field 'parent' is added to struct irq_domain; it's used to maintain irq_domain hierarchy information.
2. a new field 'parent_data' is added to struct irq_data; it's used to build hierarchy irq_data to match hierarchy irq_domains. The irq_data is used to store irq_domain pointer and hardware irq number.
3. new callbacks are added to struct irq_domain_ops to support hierarchy irq_domain operations.

With support of hierarchy irq_domain and hierarchy irq_data ready, an irq_domain structure is built for each interrupt controller, and an irq_data structure is allocated for each irq_domain associated with an IRQ. Now we could go one step further to support stacked(hierarchy) irq_chip. That is, an irq_chip is associated with each irq_data along the hierarchy. A child irq_chip may implement a required action by itself or by cooperating with its parent irq_chip.

With stacked irq_chip, interrupt controller driver only needs to deal with the hardware managed by itself and may ask for services from its parent irq_chip when needed. So we could achieve a much cleaner software architecture.

For an interrupt controller driver to support hierarchy irq_domain, it needs to:

1. Implement irq_domain_ops.alloc and irq_domain_ops.free
2. Optionally implement irq_domain_ops.activate and irq_domain_ops.deactivate.
3. Optionally implement an irq_chip to manage the interrupt controller hardware.
4. No need to implement irq_domain_ops.map and irq_domain_ops.unmap, they are unused with hierarchy irq_domain.

Hierarchy irq_domain is in no way x86 specific, and is heavily used to support other architectures, such as ARM, ARM64 etc.

## Debugging

Most of the internals of the IRQ subsystem are exposed in debugfs by turning CONFIG_GENERIC_IRQ_DEBUGFS on.