Knocking at your back door (O.H.D.W.M.I.A.C.A.Y.S.)

ARM

Marc Zyngier `<marc.zyngier@arm.com>`

ELCE16

October 13, 2016

Content

- Basics of an interrupt
- Interrupt controllers
- Linux's data structures
- Chained interrupt controllers
- Hierarchical interrupt controllers
- Generic MSIs
- ...
- Profit!

Please interrupt me

ARM

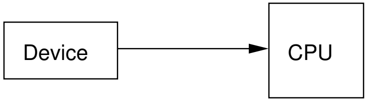
Talk you should not have missed

- IRQs: the Hard, the Soft, the Threaded and the Preemptible
- Alison Chaiken, Peloton Technology
- Took place on Tuesday[1]
- Covers the dynamic aspects of interrupt handling

[1] Use your TARDIS or wait for it to appear on some website

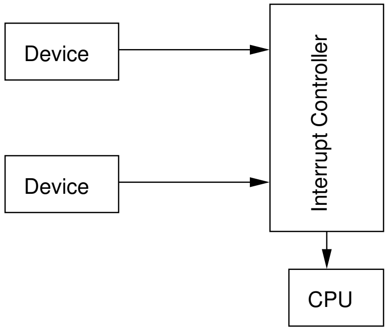
What is an interrupt?

- A hardware signal
- Emited from a peripheral to a CPU
- Indicating that a device-specific condition has been satisfied



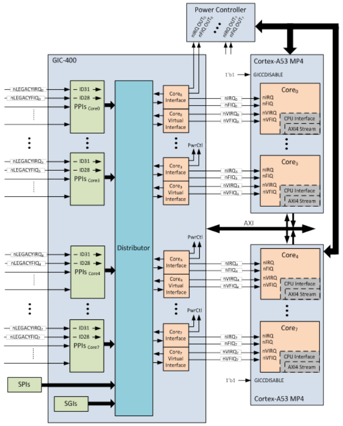
Multiplexing interrupts

- Having a single interrupt for the CPU is usually not enough
- Most systems have tens, hundreds of them
- An interrupt controller allows them to be multiplexed
- Very often architecture or platform specific
- Offers specific facilities
  - Masking/unmasking individual interrupts
  - Setting priorities
  - SMP affinity
  - Exotic things like wake-up interrupts



Multiplexing interrupts

- Having a single interrupt for the CPU is usually not enough
- Most systems have tens, hundreds of them
- An interrupt controller allows them to be multiplexed
- Very often architecture or platform specific
- Offers specific facilities
  - Masking/unmasking individual interrupts
  - Setting priorities
  - SMP affinity
  - Exotic things like wake-up interrupts



GIC-400, simplified view

Interrupt triggers

- Level triggered (high or low)
  - Indicates a persistent condition
  - An action has to be performed on the device to clear the interrupt
- Edge triggered (rising or falling)
  - Indicates an event
  - May have happened once or more…
- Some systems do not expose the trigger type to software
  - Either the interrupt is abstracted (virtualization)
  - Or this is more an exception than an interrupt…

“And now for something completely different...”

Monty Python’s Flying Circus

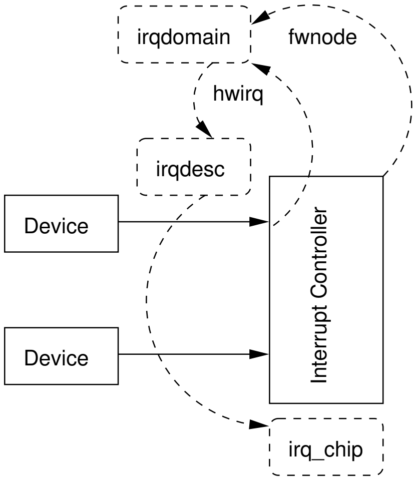
ARM

How does Linux deal with interrupts

- `struct irq_chip`
  - A set of methods describing how to drive the interrupt controller
  - Directly called by core IRQ code
- `struct irqdomain`
  - A pointer to the firmware node for a given interrupt controller (`fwnode`)
  - A method to convert a firmware description of an IRQ into an ID local to this interrupt controller (`hwirq`)
  - A way to retrieve the Linux view of an IRQ from the `hwirq`
- `struct irq_desc`
  - Linux's view of an interrupt
  - Contains all the core stuff
  - 1:1 mapping to the Linux interrupt number
- `struct irq_data`
  - Contains the data that is relevant to the `irq_chip` managing this interrupt
    - Both the Linux IRQ number and the `hwirq`
    - A pointer to the `irq_chip`
    - Embedded in `irq_desc` (for now)

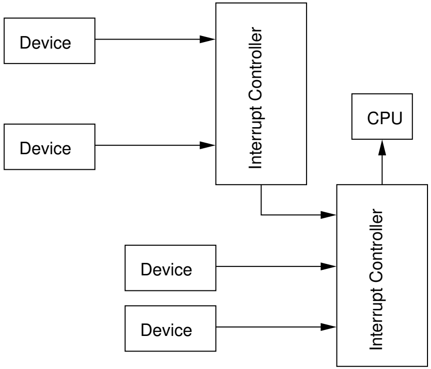
In a nutshell

- CPU gets an interrupt
- Find out the `hwirq` from the interrupt controller
  - Usually involves reading some HW register
- Look-up the `irq_desc` into the `irqdomain` using the `hwirq`
  - Actually returns an IRQ number, which is equivalent to the `irq_desc`
- The core kernel then handles the interrupt



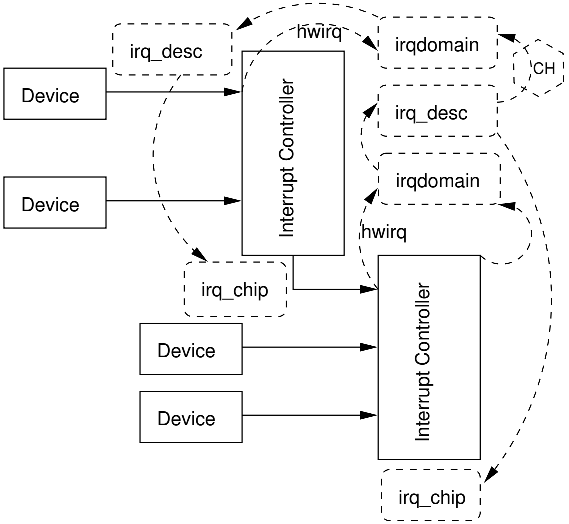
Multiplexing more interrupts

- Not enough interrupts lines?
  - Dedicate a single line for a secondary interrupt controller
  - And add more stuff to it!
- Requires two level handling
  - First handle the interrupt on the primary interrupt controller
  - Then at the secondary one to find out which device has caused the interrupt
  - See `irq_set_chained_handler_and_data`, `chained_irq_enter`, `chained_irq_exit`
  - Never treat this as a normal interrupt handler
- Used in each and every x86 system
  - The infamous i8259 cascade
- You can also share a single interrupt between devices
  - And that really stinks. Please avoid doing it if possible.



Chained irqchips, the irqdomain view

- Each interrupt controller has its own `irqdomain`
- The kernel deals with two interrupts
  - and two interrupt handlers
  - the first one being a `chained` handler
  - convention is to stash a pointer to the secondary domain inside the top-level `irq_desc`
- We walk the interrupt chain in reverse order
- Once we reach the last level `irq_desc`, we can process the actual interrupt handler



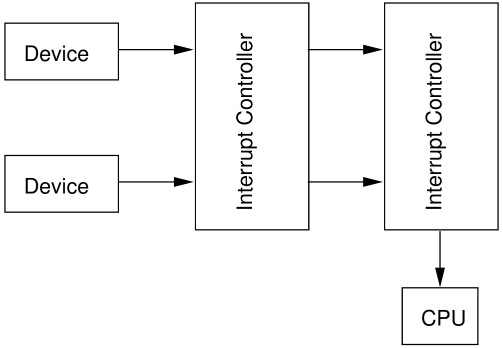
The DT view

- A secondary irqchip points to the one implementing the first level
  - Use `interrupts` to describe the signal path between irqchips
  - The secondary chip owns the cascade interrupt
  - It doesn't appear in `/proc/interrupts`
- Use `interrupt-parent` to point the device at the right interrupt controller

```
interrupt-parent = <&gic>;

gic: interrupt-controller@01c81000 {
        compatible = "arm,cortex-a7-gic", "arm,cortex-a15-gic";
        interrupt-controller;
        #interrupt-cells = <3>;
        interrupts = <GIC_PPI 9 (GIC_CPU_MASK_SIMPLE(4) |
                                 IRQ_TYPE_LEVEL_HIGH)>;
};

nmi_intc: interrupt-controller@01c00030 {
        compatible = "allwinner,sun7i-a20-sc-nmi";
        interrupt-controller;
        #interrupt-cells = <2>;
        interrupts = <GIC_SPI 0 IRQ_TYPE_LEVEL_HIGH>;
};

axp209: pmic@34 {
        interrupt-parent = <&nmi_intc>;
        interrupts = <0 IRQ_TYPE_LEVEL_LOW>;
};
```

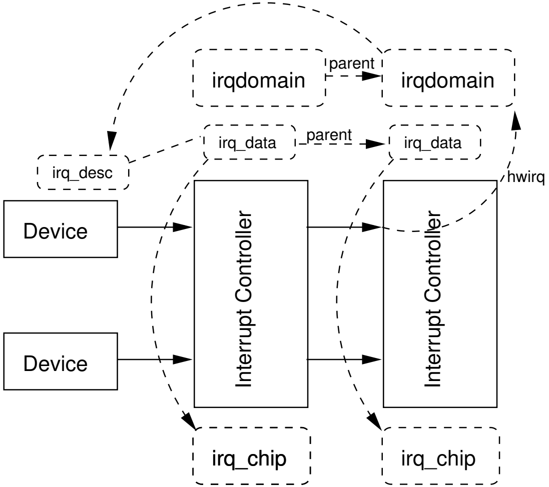
When multiplexing doesn't fit

- There is more than just cascading irqchips
- Some setups have a 1:1 mapping between input and output
  - Interrupt routers
  - Wake-up controllers
  - Programmable line inverters
- Most of them are not interrupt controllers
  - Still, they do impact the interrupt delivery
  - We choose to represent them as `irq_chip`
- This is a hierarchical/stacked configuration
- The chained irqchip paradigm doesn't match it



Hierarchical (stacked) IRQ domains

- We want the same `irq_desc` to be valid across all irqchips
  - This ensures that the Linux IRQ number is unique for a given signal path
- For a given `irq_desc`, each irqchip should be responsible for the `hwirq`
  - This fits the `irq_data` properties
- Most of the data structures now have a `parent` field representing the hierarchy
- The handling is done by walking the signal path in delivery order
  - A given irqchip can perform some local action before forwarding the request to its parent
  - Or even terminate the handling early



Hierarchical domains, the DT view

- Each intermediate irqchip points to its parent
  - Do not use `interrupts` to describe the signal path between irqchips
  - Use a device-specific property to decribe an interrupt range/space if necessary
- The root irqchip points to itself
  - A DT oddity...
- Devices can point to any element of the stack
  - The device interrupt specifiers must match the first irqchip in the signal path

```
interrupt-parent = <&sysirq>;

sysirq: intpol-controller@10200620 {
        interrupt-controller;
        #interrupt-cells = <3>;
        interrupt-parent = <&gic>;
};

gic: interrupt-controller@10231000 {
        #interrupt-cells = <3>;
        interrupt-parent = <&gic>;
        interrupt-controller;
};

uart0: serial@11002000 {
        interrupts = <GIC_SPI 91 IRQ_TYPE_LEVEL_LOW>;
};
```

“Message in a bottle”

The Police, Reggatta de Blanc

More than wired interrupts: MSIs

Message Signaled Interrupts are an essential part of the interrupt infrastructure

- A simple 32bit write (the message) from the device to a doorbell
  - The doorbell is usually the interrupt controller itself
  - The generated interrupt depends on the data being written
  - By definition edge triggered
- Avoid the spider web syndrome
  - Routing interrupts to the periphery of a SoC is a constraint
  - MSIs allows the use of the same busses as the data
  - Having multiple interrupts per device costs nothing
- Acts as a memory barrier w.r.t DMA
  - Avoid the “got an interrupt but data is not there yet” problem
- Bus agnostic
  - Historically tied to PCI(e)
  - Now implemented on all kinds of busses…

The goals of supporting MSIs in a generic way

- We'd like to support MSIs on any bus
- We want to cater for the weird and wonderful stuff
  - Intel's DMAR
  - ARM's GICv3 ITS
  - Freescale's MC bus
  - Platform devices
  - Hisilicon's MBIGEN
- Must nicely cohabit with the current PCI/MSI implementation
- Hierarchical domains are a good solution for this[2]
  - Entirely implemented as part of the core IRQ code (`kernel/irq/msi.c`)
  - Per-bus front-ends
    - `drivers/pci/msi.c`
    - `drivers/base/platform-msi.c`
    - `drivers/staging/fsl-mc/bus/mc-msi.c`

[2]Please trust me on that one...

Generic MSI

- irq_chip grows two new methods
  - irq_compose_msi_msg: populate a msi_msg
    - Address of the doorbell + data to be written
    - Implemented by the MSI controller, bus agnostic
  - irq_write_msi_msg
    - Write the content of the msi_msg to a given device
    - Implemented by the bus front-end, bus specific
- msi_domain_info to describe a MSI domain
  - A struct irq_chip
    - Must at least contain a irq_write_msi_msg method
  - A struct msi_domain_ops
    - A set of functions used to build an irqdomain
  - A set of flags (some bus specific), and allowing most of the above to get sensible defaults
- Bus specific irqdomain creation functions

```
/*
 * PCI/MSI setup
 */
static struct irq_chip my_msi_irq_chip = {
        .name              = "MSI",
        .irq_eoi           = irq_chip_eoi_parent,
        .irq_write_msi_msg = pci_msi_domain_write_msg,
};

static struct msi_domain_info my_msi_dom_info = {
        .flags        = (MSI_FLAG_USE_DEF_DOM_OPS |
                         MSI_FLAG_USE_DEF_CHIP_OPS |
                         MSI_FLAG_PCI_MSIX),
        .chip         = &my_msi_irq_chip,
};

[...]
/*
 * Build the PCI/MSI domain on top of the IRQ domain
 * representing the MSI hardware
 */
pci_domain = pci_msi_create_irq_domain(fwnode,
                                        &my_msi_dom_info,
                                        irq_domain);
```

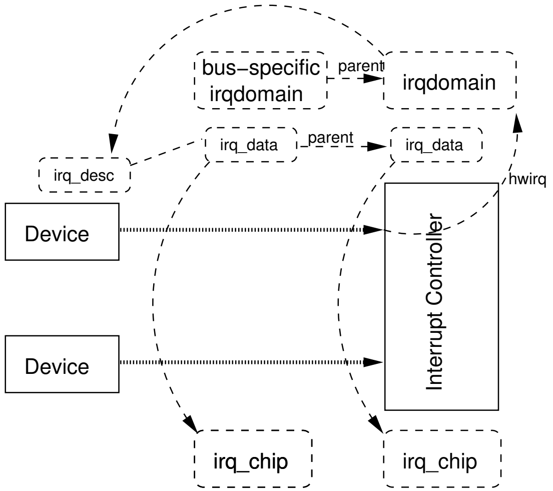
Generic MSI

- irq_chip grows two new methods
  - irq_compose_msi_msg: populate a msi_msg
    - Address of the doorbell + data to be written
    - Implemented by the MSI controller, bus agnostic
  - irq_write_msi_msg
    - Write the content of the msi_msg to a given device
    - Implemented by the bus front-end, bus specific
- msi_domain_info to describe a MSI domain
  - A struct irq_chip
    - Must at least contain a irq_write_msi_msg method
  - A struct msi_domain_ops
    - A set of functions used to build an irqdomain
  - A set of flags (some bus specific), and allowing most of the above to get sensible defaults
- Bus specific irqdomain creation functions

```
/*
 * platform-msi setup
 */
static struct irq_chip my_pmsi_irq_chip = {
        .name    = "pMSI",
};

static struct msi_domain_ops my_pmsi_ops = {
};

static struct msi_domain_info my_pmsi_dom_info = {
        .flags   = (MSI_FLAG_USE_DEF_DOM_OPS |
                    MSI_FLAG_USE_DEF_CHIP_OPS),
        .ops     = &my_pmsi_ops,
        .chip    = &my_pmsi_irq_chip,
};

[...]
/*
 * Build the platform-msi domain on top of the IRQ domain
 * representing the MSI hardware
 */
plat_domain = platform_msi_create_irq_domain(fwnode,
                                             &my_pmsi_dom_info,
                                             irq_domain);
```

Generic MSI in pictures

- At configuration time
  - The MSI controller irqchip composes the message
  - The bus-specific irqchip programs the device
- Everything is just like the stacked irqchip scenario
  - The only notable difference is that we have a bus-specific irqdomain that doesn't correspond to any HW
  - Its main function is to cater for different programing interfaces at the device level



A platform MSI special

- There is no such thing as a “standard” platform device
- No way to implement a `irq_write_msi_msg` in a standard way
- Worked around by providing it at allocation time
    - The function is per-device
    - Allows for any crazy stuff

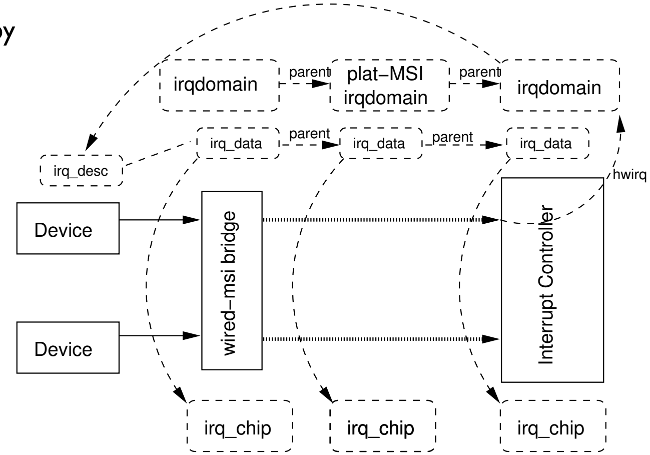
```
static void arm_smmu_write_msi_msg(struct msi_desc *desc,
                                   struct msi_msg *msg)
{
  doorbell = (((u64)msg->address_hi) << 32) | msg->address_lo;

  writeq_relaxed(doorbell, smmu->base + cfg[0]);
  writel_relaxed(msg->data, smmu->base + cfg[1]);
}

static void arm_smmu_setup_msis(struct arm_smmu_device *smmu)
{
[...]
  ret = platform_msi_domain_alloc_irqs(dev, nvec,
                                       arm_smmu_write_msi_msg);
[...]
  for_each_msi_entry(desc, dev) {
          switch (desc->platform.msi_index) {
          /* request desc->irq */
          }
  }
}
```

“I’m going slightly mad”

Queen, Innuendo

The interrupt strikes back

- Just as we thought we had fixed the world by giving MSIs to everyone...
- People now build wired interrupt controllers...
- ... that use MSI as their transport
  - Allows wired devices to be placed far away from the irqchip
  - Conveniently, one MSI per wire
- Stacked domains to the rescue!
  - The irqchip is a MSI-capable device
  - We can give it its own irqdomain



Wire-MSI bridges, the programatic view

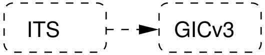
- At probe time, create a device-specific domain
- Automatically attached to the device's `msi-parent`'s own domain
- When allocating its MSIs, place them in that domain
- Dish out wired interrupts as a normal irqchip

```
static struct irq_domain_ops mbigen_domain_ops = {
  [...]
};

static int mbigen_irq_domain_alloc(struct irq_domain *domain,
                                   unsigned int virq,
                                   unsigned int nr_irqs,
                                   void *args)
{
  struct irq_fwspec *fwspec = args;

  mbigen_domain_translate(domain, fwspec, &hwirq, &type);
  platform_msi_domain_alloc(domain, virq, nr_irqs);
  mgn_chip = platform_msi_get_host_data(domain);

  for (i = 0; i < nr_irqs; i++)
    irq_domain_set_hwirq_and_chip(domain, virq + i, hwirq + i,
                                  &mbigen_irq_chip, mgn_chip->base);
}

static struct irq_domain_ops mbigen_domain_ops = {
        .alloc = mbigen_irq_domain_alloc,
};

static int mbigen_device_probe(struct platform_device *pdev)
{
[...]
  domain = platform_msi_create_device_domain(&child->dev,
                                             num_pins,
                                             mbigen_write_msg,
                                             &mbigen_domain_ops,
                                             mgn_chip);
}
```

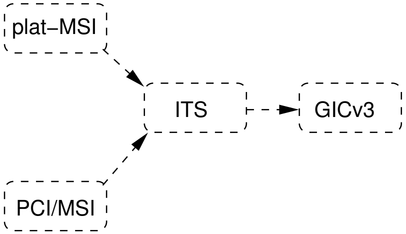
IRQ domains, as seen on `arm64`...

IRQ domains, as seen on `arm64`...

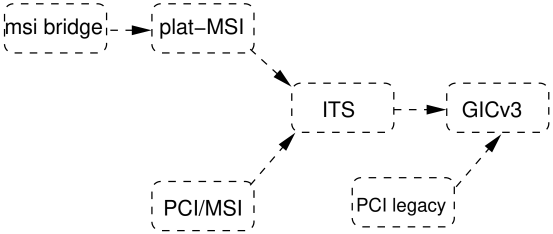
IRQ domains, as seen on `arm64`...



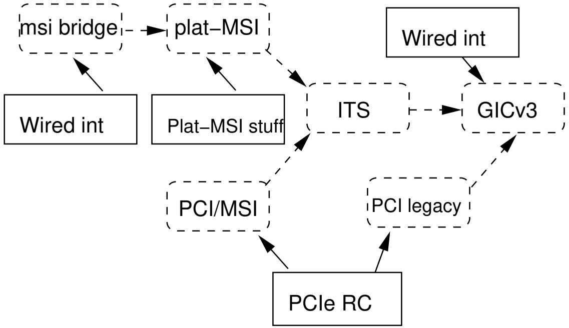
IRQ domains, as seen on `arm64`...



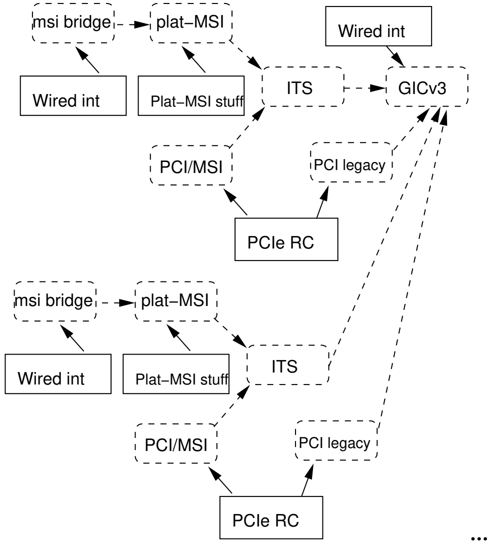
IRQ domains, as seen on `arm64`...



IRQ domains, as seen on `arm64`...



IRQ domains, as seen on `arm64`...



Thank you!

ARM