

EISA bus support

Author: Marc Zyngier <maz@wild-wind.fr.eu.org>

This document groups random notes about porting EISA drivers to the new EISA/sysfs API.

Starting from version 2.5.59, the EISA bus is almost given the same status as other much more mainstream busses such as PCI or USB. This has been possible through sysfs, which defines a nice enough set of abstractions to manage busses, devices and drivers.

Although the new API is quite simple to use, converting existing drivers to the new infrastructure is not an easy task (mostly because detection code is generally also used to probe ISA cards). Moreover, most EISA drivers are among the oldest Linux drivers so, as you can imagine, some dust has settled here over the years.

The EISA infrastructure is made up of three parts:

- The bus code implements most of the generic code. It is shared among all the architectures that the EISA code runs on. It implements bus probing (detecting EISA cards available on the bus), allocates I/O resources, allows fancy naming through sysfs, and offers interfaces for driver to register.
- The bus root driver implements the glue between the bus hardware and the generic bus code. It is responsible for discovering the device implementing the bus, and setting it up to be latter probed by the bus code. This can go from something as simple as reserving an I/O region on x86, to the rather more complex, like the hppa EISA code. This is the part to implement in order to have EISA running on an "new" platform.
- The driver offers the bus a list of devices that it manages, and implements the necessary callbacks to probe and release devices whenever told to.

Every function/structure below lives in <linux/eisa.h>, which depends heavily on <linux/device.h>.

Bus root driver

```
int eisa_root_register (struct eisa_root_device *root);
```

The `eisa_root_register` function is used to declare a device as the root of an EISA bus. The `eisa_root_device` structure holds a reference to this device, as well as some parameters for probing purposes:

```
struct eisa_root_device {
    struct device *dev;      /* Pointer to bridge device */
    struct resource *res;
    unsigned long   bus_base_addr;
    int             slots;   /* Max slot number */
    int             force_probe; /* Probe even when no slot 0 */
    u64             dma_mask; /* from bridge device */
    int             bus_nr;  /* Set by eisa_root_register */
    struct resource eisa_root_res; /* ditto */
};
```

node	used for <code>eisa_root_register</code> internal purpose
dev	pointer to the root device
res	root device I/O resource
bus_base_addr	slot 0 address on this bus
slots	max slot number to probe
force_probe	Probe even when slot 0 is empty (no EISA mainboard)
dma_mask	Default DMA mask. Usually the bridge device <code>dma_mask</code> .
bus_nr	unique bus id, set by <code>eisa_root_register</code>

Driver

```
int eisa_driver_register (struct eisa_driver *edrv);
void eisa_driver_unregister (struct eisa_driver *edrv);
```

Clear enough?

```
struct eisa_device_id {
    char sig[EISA_SIG_LEN];
    unsigned long driver_data;
};

struct eisa_driver {
    const struct eisa_device_id *id_table;
    struct device_driver         driver;
};
```

id_table	an array of NULL terminated EISA id strings, followed by an empty string. Each string can optionally be paired with a driver-dependent value (driver_data).
driver	a generic driver, such as described in Documentation/driver-api/driver-model/driver.rst. Only .name, .probe and .remove members are mandatory.

An example is the 3c59x driver:

```
static struct eisa_device_id vortex_eisa_ids[] = {
    { "TCM5920", EISA_3C592_OFFSET },
    { "TCM5970", EISA_3C597_OFFSET },
    { "" }
};

static struct eisa_driver vortex_eisa_driver = {
    .id_table = vortex_eisa_ids,
    .driver = {
        .name = "3c59x",
        .probe = vortex_eisa_probe,
        .remove = vortex_eisa_remove
    }
};
```

Device

The sysfs framework calls .probe and .remove functions upon device discovery and removal (note that the .remove function is only called when driver is built as a module).

Both functions are passed a pointer to a 'struct device', which is encapsulated in a 'struct eisa_device' described as follows:

```
struct eisa_device {
    struct eisa_device_id id;
    int slot;
    int state;
    unsigned long base_addr;
    struct resource res[EISA_MAX_RESOURCES];
    u64 dma_mask;
    struct device dev; /* generic device */
};
```

id	EISA id, as read from device. id.driver_data is set from the matching driver EISA id.
slot	slot number which the device was detected on
state	set of flags indicating the state of the device. Current flags are EISA_CONFIG_ENABLED and EISA_CONFIG_FORCED.
res	set of four 256 bytes I/O regions allocated to this device
dma_mask	DMA mask set from the parent device.
dev	generic device (see Documentation/driver-api/driver-model/device.rst)

You can get the 'struct eisa_device' from 'struct device' using the 'to_eisa_device' macro.

Misc stuff

```
void eisa_set_drvdata (struct eisa_device *edev, void *data);
```

Stores data into the device's driver_data area.

```
void *eisa_get_drvdata (struct eisa_device *edev):
```

Gets the pointer previously stored into the device's driver_data area.

```
int eisa_get_region_index (void *addr);
```

Returns the region number (0 <= x < EISA_MAX_RESOURCES) of a given address.

Kernel parameters

eisa_bus.enable_dev

A comma-separated list of slots to be enabled, even if the firmware set the card as disabled. The driver must be able to properly initialize the device in such conditions.

eisa_bus.disable_dev

A comma-separated list of slots to be enabled, even if the firmware set the card as enabled. The driver won't be called to handle this device.

virtual_root.force_probe

Force the probing code to probe EISA slots even when it cannot find an EISA compliant mainboard (nothing appears on

slot 0). Defaults to 0 (don't force), and set to 1 (force probing) when either CONFIG_ALPHA_JENSEN or CONFIG_EISA_VLB_PRIMING are set.

Random notes

Converting an EISA driver to the new API mostly involves *deleting* code (since probing is now in the core EISA code). Unfortunately, most drivers share their probing routine between ISA, and EISA. Special care must be taken when ripping out the EISA code, so other busses won't suffer from these surgical strikes...

You *must not* expect any EISA device to be detected when returning from eisa_driver_register, since the chances are that the bus has not yet been probed. In fact, that's what happens most of the time (the bus root driver usually kicks in rather late in the boot process). Unfortunately, most drivers are doing the probing by themselves, and expect to have explored the whole machine when they exit their probe routine.

For example, switching your favorite EISA SCSI card to the "hotplug" model is "the right thing"(tm).

Thanks

I'd like to thank the following people for their help:

- Xavier Benigni for lending me a wonderful Alpha Jensen,
- James Bottomley, Jeff Garzik for getting this stuff into the kernel,
- Andries Brouwer for contributing numerous EISA ids,
- Catrin Jones for coping with far too many machines at home.