Chapter 27. Plug’n’Play (15) Tom Lees

PnP Hardware Implementations

In the beginning, all PC hardware was either built-in to the motherboard, or on ISA expansion cards. All cards that required an IRQ had to have the IRQ number they used hard-wired (or switchable using jumpers), and similarly for IO addresses and DMA channels.

This situation meant that conflicts between different cards could often appear, and one would have to open the case up and change numerous jumpers to fix them, if the cards allowed non-conflicting addresses to be used.

The first attempts to fix these problems were made by the MCA and EISA bus architectures, both of which included some method of configuration of the individual ports and interrupts used by the hardware, along with many other modernising features. However, EISA and MCA didn’t take off seriously, partly because of the cost of them, and partly because there was a chicken-and-egg situation: it wasn’t worth buying EISA unless you had EISA hardware, and you couldn’t use the hardware if you didn’t have the motherboard. MCA was faced with the problem of not being a very “open” standard: only IBM PS/2’s ever had an MCA bus, and so it never took off.

Consequently, EISA was limited mainly to high-end servers, where its other advantages in throughput were the overriding factor.

The next generation of fixes to the configuration problem were soft-configurable cards. These cards could be configured using a software interface, however, all these interfaces were different. Each card needed a different vendor’s utility, and it was still possible to set conflicting addresses.

However, configuration of hardware is not the only problem associated with hard-wiring the configuration of a device. If the system wishes to automatically detect this device, it must scan all possible IO port ranges to find where the card responds (and possibly also do the same with interrupts and DMA channels), which may possibly involve stepping on another card’s resources. In the cases of certain cards (mainly SCSI controllers and 8390-based Ethernet cards), this could cause a crash of the system.

Finally, the PCI bus and ISA Plug’n’Play standards were drawn up (separately), which addressed the issues: ISA PnP hardware didn’t need a new bus, but simply extended the old one, to allow relatively pain-free card detection and configuration. PCI (and also PC Card, USB, and almost all other newer bus standards) supports PnP from the start, but there was real momentum behind this specification – support from Intel and other major hardware manufacturers made it the standard.

This chapter will concentrate mainly on PnP-ISA, since PCI and other busses are treated in other chapters.

ISA

The PnP ISA standard is described in PnPISA94. The essential points are:

- A hardware interface to which all PnP-ISA cards respond
- The division of hardware into "cards" and "devices"
- A data format describing how to configure each device
- An interface to configure hardware if need be, or alternatively disable it

Every PnP ISA device is identified by a so-called EISA ID [1] — three upper-case letters, followed by a 4-digit hexadecimal number. The three letters uniquely identify the manufacturer of the device — for example, `CTL` is Creative Technology.

The `PNP` manufacturer identifier is used for standard devices — serial ports and the like. A list of these standard identifiers can be found in the last section of this chapter. Any device which provides compatibility with a standard device should indicate this in its resource information (which the PnP-ISA kernel driver retrieves automatically when probing for devices).

ISA PnP uses a write port, fixed at `0x279`, and a read port, variable from `0x200` to `0x3ff` (set by writing its value to a register via a sequence of writes to the write port). In order to prevent accidental reconfiguration or misconfiguration of PnP-ISA devices, the write port `0x279` is protected by requiring a 32-byte long "key" to be written before PnP-ISA functions become active.

After the key has been written, the cards on the system must be "isolated": reading from the isolation register each card returns the next bit of its identifier, followed by its serial number, lower-number cards "chickening out" if they detect another card signalling a high bit when they have a low bit. Each card as it is isolated should be assigned an identifier so the kernel PnP-ISA driver can keep track of it.

Part of the information returned by the card is a list of configurations for each subdevice of the card, each tagged with a level of preference — if no IRQs are available, for example, the card may specify a lower-priority configuration with no IRQs used. These groups of resource settings are termed sets of "dependent resources" by the specification.

For more details on how the PnP-ISA hardware works, see PnPISA94.

BIOS

Along with the specification of PnP-ISA, the PnP BIOS specification was also created. The primary reasons for this were to allow resources built-in to the motherboard to also be configured and detected automatically. Microsoft's Windows 95 and up use the PnP BIOS to detect ALL hardware in the system, if it is present; the PnP BIOS includes support for returning the read port used by the BIOS to configure

PnP ISA devices, and for revealing the existence of a PCI bus. However, Linux does not take this approach. Instead, each bus system is handled separately.

Another related standard is the ESCD standard (PnPESCD). This defines a standard for storing the configuration of certain devices in the system, and the PnP BIOS defines methods for accessing and modifying this data. This data is typically used by the BIOS during system startup to configure the hardware — thus, if the OS modifies this data, the BIOS might configure the hardware how the OS wants next time the machine is restarted. This is particularly useful in principle for some hardware which can only be configured once, on system startup. The normal PnP BIOS interface as defined in PnPBIOS also allows these devices to be reconfigured, but only if they are built-in to the motherboard (which is usually the case). In practice, such hardware is rare, and so the standard is somewhat duplicating the interfaces

PCI

PCI adds several features which greatly improve the situation when considering limited resources. For a start, the PCI IO space extends way beyond the traditional ISA IO resource space. And with PCI, devices should be expected to be at ANY IO address by the OS, so there isn't any problem with "standard locations" being shared, as there is with ISA. PCI also allows devices to (if necessary) share IRQs safely (under ISA, simultaneous IRQs from two devices on the same channel could potentially cause hardware problems).

However, almost all PCI setup should be done by the system's PCI BIOS. After system startup, the OS is limited almost entirely to simply finding out where the BIOS has put things.

PCI is covered in more detail in Chapter 20.

Other systems

There are several other device connection systems which could be described as "Plug'n'Play" in one sense or another. Among these are USB (the section called *USB Devices (here or own chapter?)* in Chapter 17), and PC Card (PCMCIA).

Resources

The main site on the web where you can find information about Plug-n-Play hardware is on Microsoft's FTP (ftp://ftp.microsoft.com/pub/developr/drg/plug-and-play/) site

Kernel ISA support

As of version 2.4, the Linux kernel includes basic support for PnP-ISA devices. The sources for the implementation of this support may be found in the `drivers/pnp` subtree of the kernel source. The C source files are specifically `drivers/pnp/isapnp.c`, `drivers/pnp/isapnp_proc.c`, `drivers/pnp/quirks.c`, and `include/linux/isapnp.h`. There is also a small README file in `Documentation/isapnp.txt`.

Using the ISA driver

The Linux PnP driver architecture in the 2.4 kernel is relatively simple. The basic steps to making your hardware driver use the kernel PnP services are as follows:

1. Locate PnP-ISA cards with particular identifiers we want to drive.
2. Probe these cards for devices the driver wants to use
3. Initialise the driver normally with these devices
4. When the driver stops using these devices, deactivate them

The first action you would want to take is to try and locate any cards your driver handles. This is done by calling one of a number of functions, depending on exactly what you want to do. Typically, most drivers want to search for particular devices. This is done using one of the functions `isapnp_find_dev` or `isapnp_probe_cards`. The respective definitions of these functions are:

```
struct pci_dev *isapnp_find_dev(struct pci_bus *card,
unsigned short vendor,
unsigned short function,
struct pci_dev *from);
int isapnp_probe_cards(const struct isapnp_card_id *ids,
       int (*probe)(struct pci_bus *card,
    const struct isapnp_card_id *id));
```

These definitions are in the file `include/linux/isapnp.h`, and the functions themselves are defined in `drivers/pnp/isapnp.c`. Deciding which of these functions to use is more a matter of taste than of function; the `isapnp_find_dev` function will search for the particular device you want, whereas `isapnp_probe_cards` will look for any of a list of cards.

`isapnp_find_dev()` searches the "bus tree" stored in memory by the PnP-ISA driver. The card argument specifies the top of the tree you want to search from: if you want to search for any device in the system, pass `NULL` for this argument, but if you want to search on a particular card (which you have previously located), pass the address of that cards' structure instead. Similarly, the dev argument also controls where the search starts from — if you want to search for more than one card, you need to use this argument to stop the first card from being returned each time you call the search. To start searching

from the beginning, pass `NULL` as the from argument; to find the second instance of a particular device (if it exists), pass the pointer returned for the first instance in the from argument.

The vendor and function arguments specify the device type. As mentioned above, every PnP-ISA device and card type has its own unique device identifier, three letters and a 16-bit number. However, in addition, each device or card can provide a list of device IDs it is backwards- or sideways-compatible with, so that if manufacturers and driver-writers do their jobs properly, most devices should not require modifications to be made by the driver. [2]

In order to search for a card or device, you need to know the identifier or set of identifiers which are used by the card to identify itself. There are a couple of ways you can do this; you can find out from the manufacturer, use the list of standard identifiers in the last section of this chapter, or plug the device in and find out what it calls itself. This last method is probably the most effective. You could use the kernel PnP-ISA subsystem's `/proc` interface, documented below, to achieve this. Once you have the identifier or identifiers of the device or card, use the `ISAPNP_VENDOR` and `ISAPNP_FUNCTION` macros to convert the identifier into the binary form used by the kernel PnP-ISA subsystem.

Caution

Do not forget to use the `ISAPNP_FUNCTION` macro: It converts the endianness of the identifier so that the identifier is the same on all systems.

For example, you might use the following code fragment to find the first Creative Labs Sound Blaster 16 compatible device (identifier `PNPB003`) in the system:

```
int find_card (void)
{
struct pci_dev *d;

d = isapnp_find_dev (NULL, ISAPNP_VENDOR ('P','N','P'),
ISAPNP_FUNCTION (0xB003), NULL);
if (!d) return -ENODEV;
printk ("found an SB16-compatible device...\n");
[...]
}
```

The alternative way to find devices is to use the isapnp_find_card function. This function allows you to pass a list of identifiers you support, so is useful for example if several compatible devices don't have a common compatible device identifier, or if your driver supports multiple variants of the same device which use different identifiers.

To use `isapnp_probe_card()`, you need to construct a list of structures containing the identifiers you are looking for. The PnP-ISA subsystem allows you to associate a piece of data (an unsigned long) with each device type in addition; you might use this (for example) to indicate to your probing code that the

resources on a particular variant are specified in a different order, or that it needs slightly different handling. The structure in question is the isapnp_card_id structure, defined as follows:

```
struct isapnp_card_id {
unsigned short vendor, device;
struct {
unsigned short vendor, function;
} devs[ISAPNP_CARD_DEVS]; /* logical devices */
unsigned long driver_data; /* data private to the driver */
};
```

The structure members *`vendor`* and *`device`* should be filled with the card identifier, generated using the `ISAPNP_VENDOR` and `ISAPNP_DEVICE` macros. If you don't want to search for a specific card, and care only about the devices, put the value `ISAPNP_ANY_ID` into each of these. You can put it into only one of them, but that's probably only useful in fairly extreme circumstances. The devs array within each structure should then be filled with a list of logical device identifiers you wish to use from that card, terminated with zero in each of the values `devs[n].vendor` and `devs[n].device`. Note that you can search for a maximum of `ISAPNP_CARD_DEVS` devices on any one card (this is currently defined as 8). The list of these structures should be terminated by `ISAPNP_CARD_END`, a macro defined in `include/linux/isapnp.h`.

The `isapnp_probe_cards` function should be called with your list passed in the *`ids`* argument, and a pointer to a callback function which probes a particular card to see if you want to drive it. The probe function should take arguments of the *card* (not device) being probed, and the *`driver_data`* specified in the structure. The function returns the number of cards for which the probe function returned a non-negative result.

Here is an example code fragment using `isapnp_probe_cards()` to find a card which has both SoundBlaster 16 device and an Adlib-compatible device (identifiers `PNPB003` and `PNPB020`).

```
static int card_found;
struct pci_dev *sb, *opl;

static int probe_isapnp_card (struct pnp_bus *card, un-
signed long driver_data)
{
if (card_found)
return -ENOMEM;

sb = isapnp_find_dev (card, ISAPNP_VENDOR ('P','N','P'),
ISAPNP_FUNCTION (0xB003), NULL);
opl = isapnp_find_dev (card, ISAPNP_VENDOR ('P','N','P'),
ISAPNP_FUNCTION (0xB020), NULL);
printk ("found a card!\n");
```

```
[...]
return 0;
}

static struct isapnp_card_id card_ids[] =
{
{
ISAPNP_ANY_ID, ISAPNP_ANY_ID,
{
{ ISAPNP_VENDOR ('P','N','P'), ISAPNP_FUNCTION (0xB003) },
{ ISAPNP_VENDOR ('P','N','P'), ISAPNP_FUNCTION (0xB020) },
{ 0, 0 },
},
1
},
{ ISAPNP_CARD_END }
};

int find_pnp_cards (void)
{
if (!isapnp_probe_cards (card_ids, probe_isapnp_card))
{
sb = opl = NULL;
printk ("found no cards.\n");
}
[...]
}
```

Less commonly to locating particular devices or groupings of devices is locating particular cards by identifier. This is accomplished using `isapnp_find_card()`:

```
struct pci_bus *isapnp_find_card(unsigned short vendor,
 unsigned short device,
 struct pci_bus *from);
```

The vendor and device arguments specify the identifier for the card. Again, you need to use the `ISAPNP_VENDOR` and `ISAPNP_DEVICE` macros to wrap the identifiers. The from argument specifies the position within the list of cards to search from. `NULL` specifies the beginning, or you can pass the previous card returned to find the next.

There are a couple of other high-level functions and macros defined in `isapnp.h` worth mentioning at this point. The first are a pair of macros useful if the above methods for finding devices and cards aren't sufficient for your purposes. These macros are `isapnp_for_each_card` and `isapnp_for_each_dev`. They are defined as follows in `isapnp.h`:

```
#define isapnp_for_each_card(card) \
for(card = pci_bus_b(isapnp_cards.next); \
    card != pci_bus_b(&amp;isapnp_cards); \
    card = pci_bus_b(card->node.next))

#define isapnp_for_each_dev(dev) \
for(dev = pci_dev_g(isapnp_devices.next); \
    dev != pci_dev_g(&amp;isapnp_devices); \
    dev = pci_dev_g(dev->global_list.next))
```

They basically code for you a loop which will run over the global list of cards or devices. For example, you might want to find the first configured device with ID `PNP0600` (hard disk controller). You can do this with the following code:

```
int find_active_ide_pnp_devs (void)
{
struct pci_dev *d, *fd = NULL;

isapnp_for_each_dev (d)
if (d->resource[0])
{
fd = d;
break;
}
if (!fd)
return;
printk ("found active IDE!\n");
[...]
}
```

You'll notice that the PnP-ISA system shares the PCI subsystem's pci_dev and pci_bus structures.

Other PnP-ISA Driver Functions

Here are listed some of the functions which can be used by drivers for more control over what the subsystem does to hardware.

```
int isapnp_present(void);
```

returns 0 if there is ISA PnP hardware present in the system, non-zero otherwise.

```
void isapnp_resource_change ( struct resource *resource unsigned long start
unsigned long size );
```

manually set a particular resource (IO, IRQ, etc.) up after the driver has automatically filled in the resources — sets the start and end addresses of the resource, and also flags to the PnP subsystem not to reconfigure this resource dynamically. A driver can use this function to force a PnP device it finds to be configured at a particular address, if for example the device needs to be at this address due to buggy hardware or other software or hardware expects to find it there.

Inside the kernel PnP-ISA driver

This section gives a brief tour of the main low-level and internal functions in the driver. Most of the functions are in `isapnp.c`; if a function is found in another file, it is mentioned in its description.

Low-level hardware functions

```
unsigned char isapnp_read_byte( unsigned char idx );

unsigned short isapnp_read_word( unsigned char idx );

unsigned int isapnp_read_dword( unsigned char idx );
```

read a byte, word, or dword from register `idx` on the currently-selected card/device.

```
void isapnp_write_byte( unsigned char idx unsigned char val );

void isapnp_write_word( unsigned char idx unsigned short val );

void isapnp_write_dword( unsigned char idx unsigned int val );
```

write the byte, word, or dword *`val`* into register *`idx`* on the currently-selected card/device.

`static void` **`isapnp_key`**`(` *`void`* `);`

writes the ISA PnP key LFSR sequence to the ISA PnP write port, which is required to activate the ISA-PnP registers on all ISA PnP cards (which happens simultaneously upon complete reception of this key).

`static void` **`isapnp_wait`**`(` *`void`* `);`

performs the reverse operation to `isapnp_key`; returns all cards to the wait-for-key state.

`void` **`isapnp_wake`**`(` *`unsigned char csn`* `);`

select the given card *`csn`* by writing its CSN into the wake register.

`void` **`isapnp_device`**`(` *`unsigned char device`* `);`

select the given device on the currently-selected card by writing its (vendor-assigned) device number into the logical device select register.

`void` **`isapnp_activate`**`(` *`unsigned char device`* `);`

`void` **`isapnp_deactivate`**`(` *`unsigned char logdev`* `);`

activate/deactivate the logical device *`device`* on the currently-selected card by writing a 1 or 0 respectively into the activate register, after selecting the device.

`static void __init` **`isapnp_peek`**`(` *`unsigned char *data int bytes`* `);`

reads *`bytes`* bytes from the isolation register (effectively isolating the next card) into data, and updates the running checksum kept in `isapnp_checksum_value`.

`static int` **`isapnp_next_rdp`**`(` *`void`* `);`

advances the variable `isapnp_rdp` to the next available (according to `check_region` location in steps of `RDP_STEP`; `RDP_STEP` defaults to 32).

```
static inline void isapnp_set_rdp( void );
```

writes the current value of `isapnp_rdp` into the read-port register.

```
static int __init isapnp_isolate_rdp_select( void );
```

used by the isolation routines described below — selects the next available read port as returned by `isapnp_set_rdp` ready for isolation.

```
static int __init isapnp_isolate( void );
```

tries different read-ports until an isolation is successful, then isolates all ISA PnP cards in the system and returns the number of cards isolated.

```
static int __init isapnp_read_tag( unsigned char *type unsigned short *size
);
```

reads the next single tag from the currently-selected card's resource tag list.

```
static void __init isapnp_skip_bytes( int count );
```

reads and discards *`count`* bytes from the currently-selected card's resource tag list.

```
int isapnp_cfg_begin( int csn int device );
```

selects the card which has been assigned CSN *`csn`* and is logical device number *`device`*.

```
int isapnp_cfg_end( void );
```

returns all cards to the wait-for-key state using `isapnp_wait`. These two functions are actually at a higher level than simply using the hardware access functions to do the same, since they also operate a mutex, so that only one (kernel) process may reconfigure any ISA PnP device at once.

Data structures and higher-level functions

The subsystem's real use is in implementing algorithms to isolate and identify devices and their configuration capabilities and generate data structures to represent this to device drivers and to allow the configuration of the hardware for device drivers. We shall therefore start by looking at the data structures the subsystem uses. The structures are mostly defined in `include/linux/isapnp.h`:

The simplest and highest-level structure in the subsystem is the card. A card represents a single card on the ISA PnP bus, as defined in the specification. The structure used to represent a card is struct pci_bus, from the PCI subsystem.

```
struct pci_bus {
struct list_head node; /* node in list of buses */
struct pci_bus *parent; /* parent bus (not used by PnP) */
struct list_head children; /* list of child buses */
struct list_head devices; /* list of devices on this bus */
struct pci_dev *self; /* bridge device seen by parent (not used by PnP) */
struct resource *resource[4];
/* address space routed to this bus (not used by PnP) */

struct pci_ops *ops; /* configuration access functions (not used by PnP) */
void *sysdata; /* hook for sys-specific extension (not used by PnP) */
struct proc_dir_entry *procdir;
/* directory entry in /proc/bus/pci (not used by PnP) */

unsigned char number; /* bus number (CSN for PnP) */

unsigned char primary; /* number of primary bridge (not used by PnP) */
unsigned char secondary; /* number of secondary bridge (not used by PnP) */
unsigned char subordinate;
/* max number of subordinate buses (not used by PnP) */

char name[48];
unsigned short vendor;
unsigned short device;
unsigned int serial; /* serial number */
unsigned char pnpver; /* Plug & Play version */
unsigned char productver; /* product version */
unsigned char checksum;
/* if zero - checksum passed (PnP - checksum of resource map) */
unsigned char pad1;
};
```

Similarly, logical devices are represented by struct pci_dev.

```
struct pci_dev {
```

```
struct list_head global_list; /* node in list of all PCI devices */
struct list_head bus_list; /* node in per-bus list */
struct pci_bus *bus; /* bus this device is on */
struct pci_bus *subordinate; /* bus this device bridges to */

void *sysdata; /* hook for sys-specific extension */
struct proc_dir_entry *procent; /* device entry in /proc/bus/pci */

unsigned int devfn; /* encoded device & function index */
unsigned short vendor;
unsigned short device;
unsigned short subsystem_vendor;
unsigned short subsystem_device;
unsigned int class; /* 3 bytes: (base,sub,prog-if) */
u8 hdr_type; /* PCI header type ('multi' flag masked out) */
u8 rom_base_reg; /* which config register controls the ROM */

struct pci_driver *driver; /* which driver has allocated this device */
void *driver_data; /* data private to the driver */
dma_addr_t dma_mask; /* Mask of the bits of bus address this
   device implements.  Normally this is
   0xffffffff.  You only need to change
   this if your device has broken DMA
   or supports 64-bit transfers.  */

/* device is compatible with these IDs */
unsigned short vendor_compatible[DEVICE_COUNT_COMPATIBLE];
unsigned short device_compatible[DEVICE_COUNT_COMPATIBLE];

/*
 * Instead of touching interrupt line and base address registers
 * directly, use the values stored here. They might be different!
 */
unsigned int irq;
struct resource resource[DEVICE_COUNT_RESOURCE]; /* I/O and memory re-
gions + expansion ROMs */
struct resource dma_resource[DEVICE_COUNT_DMA];
struct resource irq_resource[DEVICE_COUNT_IRQ];

char name[80]; /* device name */
char slot_name[8]; /* slot name */
int active; /* ISAPnP: device is active */
int ro; /* ISAPnP: read only */
unsigned short regs; /* ISAPnP: supported registers */
```

```
int (*prepare)(struct pci_dev *dev); /* ISAPnP hooks */
int (*activate)(struct pci_dev *dev);
int (*deactivate)(struct pci_dev *dev);
};
```

The other important set of data structures are those used for the resource map for each device. These are defined in `include/linux/isapnp.h`. struct isapnp_port, struct isapnp_irq, struct isapnp_dma, struct isapnp_mem, and struct isapnp_mem32 are used to represent IO port ranges, IRQs, DMAs, and 24- and 32-bit memory ranges, each structure defining a linked-list of sets of possible resource configurations.

```
struct isapnp_port {
unsigned short min; /* min base number */
unsigned short max; /* max base number */
unsigned char align; /* align boundary */
unsigned char size; /* size of range */
unsigned char flags; /* port flags */
unsigned char pad; /* pad */
struct isapnp_resources *res; /* parent */
struct isapnp_port *next; /* next port */
};

struct isapnp_irq {
unsigned short map; /* bitmaks for IRQ lines */
unsigned char flags; /* IRQ flags */
unsigned char pad; /* pad */
struct isapnp_resources *res; /* parent */
struct isapnp_irq *next; /* next IRQ */
};
```

and similarly dma and mem like irq and io respectively:

```
struct isapnp_mem32 {
/* TODO */
unsigned char data[17];
struct isapnp_resources *res; /* parent */
struct isapnp_mem32 *next; /* next 32-bit memory resource */
};
```

struct isapnp_resources binds all these together into a set of possible resource configurations for each logical device. In addition, the concept of dependent resources (groups of possible resource settings which must all be configured together) is represented in this structure. The structure actually defines a linked-list of possible configurations, each with priority.

```
struct isapnp_resources {
unsigned short priority; /* priority */
unsigned short dependent; /* dependent resources */
struct isapnp_port *port; /* first port */
struct isapnp_irq *irq; /* first IRQ */
struct isapnp_dma *dma; /* first DMA */
struct isapnp_mem *mem; /* first memory resource */
struct isapnp_mem32 *mem32; /* first 32-bit memory */
struct pci_dev *dev; /* parent */
struct isapnp_resources *alt;
/* alternative resource (aka dependent resources) */
struct isapnp_resources *next; /* next resource */
};
```

`static int __init` **`isapnp_build_device_list`**`(` *`void`* `);`

creates the lists of all the cards in the system, and their logical devices. It loops through the CSNs in the system, and reads the resource maps for each card, processing each tag. It allocates a struct pci_bus for each card, and sets up the *`number`*, device identification, and *`serial`* members. It calls `static void __init isapnp_parse_resource_map(struct pci_bus *card)` to do its dirty work. This function processes each possible resource tag, saving its value into in-memory structures as appropriate. If a "logical device ID" tag is encountered, then this function will call `isapnp_create_device`, described below. Finally, it calls `isapnp_fixup_device` for each newly-created logical device.

`static int __init` **`isapnp_create_device`**`(` *`struct pci_bus *card unsigned short size`* `);`

allocates a struct pci_dev structure for a logical device on a card, adds this structure into the list *`devices`* in the parent card's pci_bus structure, and into the global list of devices, `isapnp_devices`. It also parses the resource tags defining possible configurations for the logical device. It calls functions called `isapnp_add_xxx_resource` to add a resource of type xxx (IO ports, IRQs, DMAs, etc).

```
void isapnp_fixup_device( struct pci_dev *dev );
```

(in quirks.c)

call an appropriate hack function from `quirks.c` if needed to fix a broken resource definition given by the card. e.g. on the Creative Labs Sound Blaster AWE32 PnP, only one IO ranges is given in the resource definition for the wave-table, when it in fact needs 3 IO ranges.

High-level configuration functions

The final main group of functions is concerned with automatically configuring an ISA PnP device based on its resource map. The main function which does this is

```
static int isapnp_config_activate ( struct pci_dev *dev );
```

. This function loops through all defined configurations in the resource map, in order, and finds the first one which does not conflict with any other resources allocated. It enlists the help of struct isapnp_cfgtmp, a structure which temporarily carries the current configuration attempt's values. If the device already contains a list of resources, none of which has the `IORESOURCE_AUTO` flag set, it will simply set up this configuration, and activate the device.

`isapnp_config_activate` is placed in the *`activate`* member of each struct pci_dev.

```
static int isapnp_config_deactivate( struct pci_dev *dev );
```

deactivates a device. `isapnp_config_deactivate` is placed in the *`deactivate`* member of each struct pci_dev.

```
static int isapnp_alternative_switch( struct isapnp_cfgtmp *cfg struct
isapnp_resources *from struct isapnp_resources *to );
```

updates the pointers in *`cfg`* for each resource to the next set of dependent resources, or to the first using the `isapnp_find_xxx` group of functions.

The `isapnp_check_xxx` group of functions check if the configuration for a resource being considered will conflict with anything currently configured or in use, and return 0 if all is OK, non-zero otherwise.

The `isapnp_valid_xxx` group of functions loop through the possible locations for the resource in question, and find the next possible valid location, checking each one using the appropriate `isapnp_check_xxx` function. They will return 0, having set up the appropriate

`cfg->result.resource[n]` structure if a suitable location is found, and non-zero if no possible location could be found.

```
static int isapnp_check_valid( struct isapnp_cfgtmp *cfg );
```

checks that all the resources in *`cfg`* have been configured properly, by checking that none of them have the `IORESOURCE_AUTO` flag set.

```
static int isapnp_config_prepare( struct pci_dev *dev );
```

sets up the `resource` array and other resource structures in *`dev`* to default initial values which will cause auto-configuration on the next call to `isapnp_config_activate`. This function is placed in the prepare field of each struct pci_dev.

`/proc` Interface

The `/proc` interface to the driver is implemented in `isapnp_proc.c`. Essentially, the `/proc` interface implements two functions: reading and writing.

Reading is done by the

```
static void isapnp_info_read( isapnp_info_buffer_t *buffer );
```

routine. Each time the file is opened, a new isapnp_info_buffer_t structure is allocated; the read routine itself will buffer some more output if necessary, and copy buffered output into the user memory as requested. The buffer contains the complete contents of the file as generated at open time; it is not generated dynamically as reading progresses.

The code which generates the output is fairly self-explanatory. The only other main function is

```
static void isapnp_print_device( isapnp_info_buffer_t *buffer struct pci_dev
*dev );
```

, which prints out the contents of the *`dev`* argument in human-readable form.

Writing to the file in `/proc` is more involved. When data is written to the file, it is fed into a parser line-by-line. The parsing function is

```
static int isapnp_decode_line( char *line );
```

. This in turn calls `isapnp_get_str` to separate the command and arguments out, and then looks for various commands. Each command is logical device-based; the first commands in each session must select the device which is to be operated on — the `card` command selects cards by name, and `csn` by csn; once a card is selected, devices are selected by means of the `dev` command which selects them by logical device number. After this has been done, devices can be autoconfigured, activated, deactivated, manually configured, or their registers can be manually poked.

List of standard PnP device IDs

Table 27-1 shows ID ranges taken from the Microsoft Device ID Reference List (ftp://ftp.microsoft.com/pub/developr/drg/plug-and-play/devids.txt) dated 25th August 1998. The file also contains a large list of specific identifiers.

Table 27-1. Device ID ranges

Device ID range	Device types
PNP0xxx	System devices
PNP8xxx	Network adapters
PNPAxxx	SCSI, proprietary CD adapters
PNPBxxx	Sound, video capture, multimedia
PNPCxxx - Dxxx	Modems
PNP0802	Microsoft Sound System-compatible device

Notes

1. In reality, the EISA ID translates into a 32-bit number. The macro `ISAPNP_CARD_ID(a,b,c,d)`, defined in `include/linux/isapnp.h` can be used to generate the vendor-portion of the number.
2. Of course, implicit in this scheme is that every compatible device uses the same method for allocating its resources, i.e. the first IO port listed in the configuration performs the same function on each implementation of the compatible device. This is not strictly enforced, but is true in general of the vast majority of hardware "out there".