Linux Plug and Play Documentation
by Adam Belay <ambx1@neo.rr.com>
last updated: Oct. 16, 2002

Overview

Plug and Play provides a means of detecting and setting resources for legacy or otherwise unconfigurable devices. The Linux Plug and Play Layer provides these services to compatible drivers.

The User Interface

The Linux Plug and Play user interface provides a means to activate PnP devices for legacy and user level drivers that do not support Linux Plug and Play. The user interface is integrated into sysfs.

In addition to the standard sysfs file the following are created in each device's directory:
id - displays a list of support EISA IDs
options - displays possible resource configurations
resources - displays currently allocated resources and allows resource changes

-activating a device

#echo "auto" > resources

this will invoke the automatic resource config system to activate the device

-manually activating a device

#echo "manual <depnum> <mode>" > resources
<depnum> - the configuration number
<mode> - static or dynamic
static = for next boot
dynamic = now

-disabling a device

#echo "disable" > resources

EXAMPLE:

Suppose you need to activate the floppy disk controller.
1.) change to the proper directory, in my case it is /driver/bus/pnp/devices/00:0f
cd /driver/bus/pnp/devices/00:0f
cat name

PC standard floppy disk controller

2.) check if the device is already active
cat resources
DISABLED

- Notice the string "DISABLED". This means the device is not active.

3.) check the device's possible configurations (optional)
cat options
Dependent: 01 - Priority acceptable
    port 0x3f0-0x3f0, align 0x7, size 0x6, 16-bit address decoding
    port 0x3f7-0x3f7, align 0x0, size 0x1, 16-bit address decoding
    irq 6
    dma 2 8-bit compatible
Dependent: 02 - Priority acceptable
    port 0x370-0x370, align 0x7, size 0x6, 16-bit address decoding
    port 0x377-0x377, align 0x0, size 0x1, 16-bit address decoding
    irq 6
    dma 2 8-bit compatible

4.) now activate the device
echo "auto" > resources

5.) finally check if the device is active
cat resources
io 0x3f0-0x3f5
io 0x3f7-0x3f7
irq 6
dma 2

also there are a series of kernel parameters:
pnp_reserve_irq=irq1[,irq2] ....
pnp_reserve_dma=dma1[,dma2] ....
pnp_reserve_io=io1,size1[,io2,size2] ....
pnp_reserve_mem=mem1,size1[,mem2,size2] ....

The Unified Plug and Play Layer

All Plug and Play drivers, protocols, and services meet at a central location
called the Plug and Play Layer. This layer is responsible for the exchange of information between PnP drivers and PnP protocols. Thus it automatically forwards commands to the proper protocol. This makes writing PnP drivers significantly easier.

The following functions are available from the Plug and Play Layer:

pnp_get_protocol
- increments the number of uses by one

pnp_put_protocol
- deincrements the number of uses by one

pnp_register_protocol
- use this to register a new PnP protocol

pnp_unregister_protocol
- use this function to remove a PnP protocol from the Plug and Play Layer

pnp_register_driver
- adds a PnP driver to the Plug and Play Layer
- this includes driver model integration
- returns zero for success or a negative error number for failure; count calls to the .add() method if you need to know how many devices bind to the driver

pnp_unregister_driver
- removes a PnP driver from the Plug and Play Layer

Plug and Play Protocols

This section contains information for PnP protocol developers.

The following Protocols are currently available in the computing world:
- PNPBIOS: used for system devices such as serial and parallel ports.
- ISAPNP: provides PnP support for the ISA bus
- ACPI: among its many uses, ACPI provides information about system level devices.
It is meant to replace the PNPBIOS. It is not currently supported by Linux Plug and Play but it is planned to be in the near future.

Requirements for a Linux PnP protocol:
1.) the protocol must use EISA IDs
2.) the protocol must inform the PnP Layer of a device's current configuration
- the ability to set resources is optional but preferred.

The following are PnP protocol related functions:

pnp_add_device
- use this function to add a PnP device to the PnP layer
- only call this function when all wanted values are set in the pnp_dev structure

pnp_init_device
- call this to initialize the PnP structure

pnp_remove_device
- call this to remove a device from the Plug and Play Layer.
- it will fail if the device is still in use.
- automatically will free mem used by the device and related structures

pnp_add_id
- adds an EISA ID to the list of supported IDs for the specified device

For more information consult the source of a protocol such as /drivers/pnp/pnpbios/core.c.

Linux Plug and Play Drivers

This section contains information for Linux PnP driver developers.

The New Way
.....
1.) first make a list of supported EISA IDS
ex:
```
static const struct pnp_id pnp_dev_table[] = {
        /* Standard LPT Printer Port */
        {.id = "PNP0400", .driver_data = 0},
        /* ECP Printer Port */
        {.id = "PNP0401", .driver_data = 0},
        {.id = ""}
};
```

Please note that the character 'X' can be used as a wild card in the function portion (last four characters).
ex:
```
        /* Unknown PnP modems */
        {       "PNPCXXX",              UNKNOWN_DEV     },
```

Supported PnP card IDs can optionally be defined.
ex:
```
static const struct pnp_id pnp_card_table[] = {
        {       "ANYDEVS",              0       },
        {       "",                     0       }
};
```

2.) Optionally define probe and remove functions. It may make sense not to define these functions if the driver already has a reliable method of detecting the resources, such as the parport_pc driver.
ex:
```
static int
serial_pnp_probe(struct pnp_dev * dev, const struct pnp_id *card_id, const
                 struct pnp_id *dev_id)
{
. . .
```

ex:
```
static void serial_pnp_remove(struct pnp_dev * dev)
{
. . .
```

consult /drivers/serial/8250_pnp.c for more information.

3.) create a driver structure
ex:

```
static struct pnp_driver serial_pnp_driver = {
        .name           = "serial",
        .card_id_table  = pnp_card_table,
        .id_table       = pnp_dev_table,
```

```
        .probe          = serial_pnp_probe,
        .remove         = serial_pnp_remove,
};
```

* name and id_table cannot be NULL.

4.) register the driver
ex:

```
static int __init serial8250_pnp_init(void)
{
        return pnp_register_driver(&serial_pnp_driver);
}
```

The Old Way
....

A series of compatibility functions have been created to make it easy to convert ISAPNP drivers. They should serve as a temporary solution only.

They are as follows:

```
struct pnp_card *pnp_find_card(unsigned short vendor,
                               unsigned short device,
                               struct pnp_card *from)

struct pnp_dev *pnp_find_dev(struct pnp_card *card,
                             unsigned short vendor,
                             unsigned short function,
                             struct pnp_dev *from)
```