Integrated
Drive
Electronics

Jaisiel Santana Almeida

Javier Hernández Trujillo

IDE

- **¿Qué es IDE?**
- **Breve historia sobre IDE/ATA**
- **Repaso a la arquitectura ATA**
- **Cómo ve el sistema los dispositivos IDE**
- **Estructuras de datos**
- **Código IDE**

¿Qué es IDE?

El sistema IDE (Integrated Drive Electronics, "Dispositivo con electrónica integrada") es el nombre comercial de ATA (Advanced Technology Attachment), que controla los dispositivos de almacenamiento masivo de datos, como son discos duros, CDROM, etc....

En principio ATA se diseño pensando en discos duros y no en unidades extraíbles, lo cual promovió una “ampliación” conocida como ATAPI (ATA Packet Interface).

Aunque muchas veces nos confundimos y lo llamamos bus, ATA es realmente una interfaz de conexión.

Breve historia sobre IDE/ATA

- Antiguamente los discos duros tenían el controlador en la placa base, y esto ocasionaba varios problemas. La idea de integrar el controlador en el propio disco surge como solución a ello.
- IBM y Western Digital diseñaron discos con esta idea y empezaron a vender PC's con este tipo de dispositivos, siendo los primeros ordenadores en llevarlos los 386, en el año 1986.
- Su competidor más cercano ha sido desde siempre la tecnología SCSI, la cual ofrece mejores prestaciones en general, pero el hecho de que IDE fuera más barato y que lo vendieran directamente con los 386 (que cabe decir tuvieron mucho éxito), produjo que los últimos consiguieran ventaja.
- El ATA está en declive en favor de su “sucesor”, SATA (Serial ATA). Aunque el nombre haga parecer que es una sucesión, realmente no lo es, simplemente se llama así por estrategia comercial y eso hay que tenerlo claro: **SATA no es ATA**. De hecho, también es llamado PATA (Parallel ATA), si bien lo desarrollan los mismos que ATA.

Repaso de la arquitectura ATA

- ATA es una interfaz de conexión, y su principal característica es que el controlador no tiene por qué estar integrado en la placa base.
- Cada controlador puede conectarse a un máximo de dos dispositivos, que no tienen por qué ser iguales, y ni siquiera del mismo tipo.
- Los dispositivos conectados al controlador tienen tres tipos de configuración: maestro, esclavo y “cable selected” (selección por cable). Uno de los dispositivos debe ser siempre maestro, tanto si hay más de un dispositivo como si sólo hay uno. En caso de que se configure con “cable selected” será el controlador el que configure el dispositivo, según su situación en el cable.
- No puede haber más de un dispositivo funcionando en el mismo controlador.

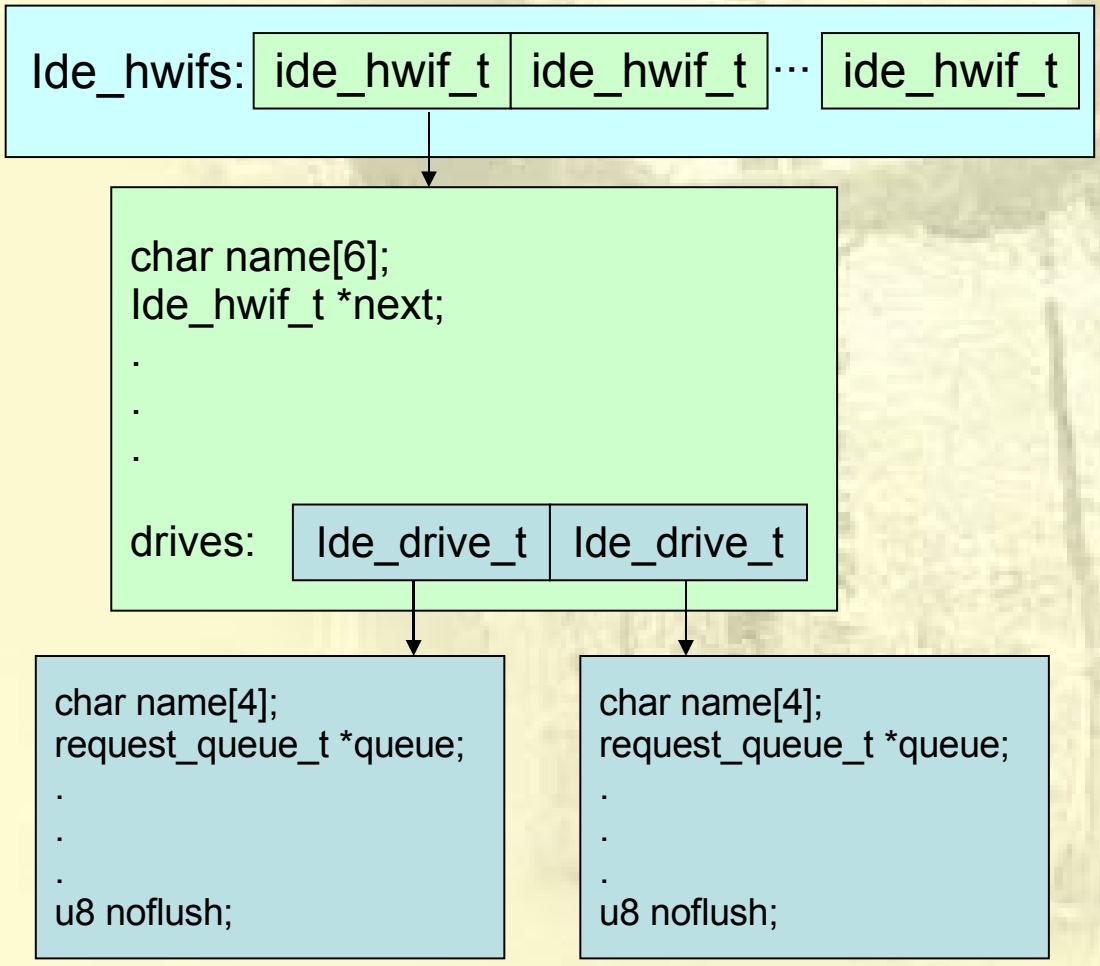
Cómo ve el sistema los dispositivos IDE

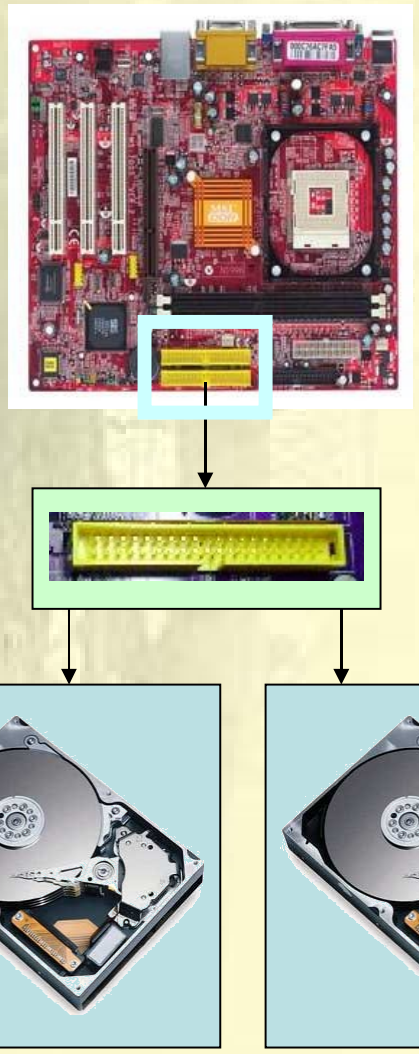
- El sistema organiza los controladores IDE en el vector **ide_hwifs**.
- Cada uno de los controladores es representado por una estructura **ide_hwif_t** del vector **ide_hwifs**.
- El tipo **ide_hwif_t**, entre sus múltiples campos, contiene un vector de tamaño **MAX_DRIVES** de **ide_drive_t** que se corresponde con el dispositivo en si.

Cómo ve el sistema los dispositivos IDE

Sistema Operativo

Hardware





Estructuras de Datos

- **Se encuentran definidas en el fichero /include/linux/ide.h**
- **ide_hwif_t** (struct hwif_s): hace referencia a un controlador IDE.
- **ide_drive_t** (ide_drive_s): hace referencia a un dispositivo.
- **ide_drive_settings_t** (ide_drive_settings_s): representa a una característica del dispositivo.

Estructuras de Datos: ide_hwif_s(1)(ide.h)

```
450  typedef struct hwif_s {
451        struct hwif_s *next;           /* para la lista encadenada en hwgroup*/
452        struct hwif_s *mate;           /* enlace para otro chip del mismo PCI */
453        struct hwgroup_s *hwgroup;     /* puntero al hwgroup */
454        struct proc_dir_entry *proc;   /* /proc/ide/ directorio de especificaciones */
455
456        char name[6];                  /* nombre de la interfaz, ej. "ide0" */
457
458
459        unsigned long   io_ports[IDE_NR_PORTS];     /*puertos de E/S*/
461
462        ide_drive_t     drives[MAX_DRIVES];     /*dispositivos*/
463
464        u8 major;       /* el número del IDE máximo */
465        u8 index;       /* Indice del IDE: 0 para el ide0, 1 para el ide1...*/
```

Estructuras de Datos: ide_hwif_s(2)(Ide.h)

```
/*funciones de lectura y escritura propias del dispositivo,que se inicializan en
    default_hwif_iops */

534 void (*OUTB)(u8 addr, unsigned long port);
535       void (*OUTBSYNC)(ide_drive_t *drive, u8 addr, unsigned long port);
536       void (*OUTW)(u16 addr, unsigned long port);
537       void (*OUTSW)(unsigned long port, void *addr, u32 count);
538       void (*OUTSL)(unsigned long port, void *addr, u32 count);
539
540       u8  (*INB)(unsigned long port);
541       u16 (*INW)(unsigned long port);
542       void (*INSW)(unsigned long port, void *addr, u32 count);
543       void (*INSL)(unsigned long port, void *addr, u32 count);
```

Estructuras de Datos: ide_drive_t(1)

```
345  typedef struct ide_drive_s {
346        char          name[4];       /* nombre del dispositivo, como "hda" */
347        char          driver_req[10]; /* por si tuviera otro controlador*/
348
349        struct request_queue    *queue;  /* cola de peticiones*/
350
351        struct request          *rq;    /* petición actual*/
352        struct ide_drive_s      *next;  /* para la lista circular en hwgroups */
353        void          *driver_data;   /* Información adicional del driver */
354        struct hd_driveid       *id;    /* información para identificar al dispositivo*/

361        unsigned long sleep;            /* dormir hasta “sleep” */
362        unsigned long service_start;    /* Tiempo en que empezó la última solicitud */
363        unsigned long service_time;     /* Tiempo del servicio de la última solicitud*/
364        unsigned long timeout;          /* Tiempo de espera máximo por IRQ */
```

Estructuras de Datos: ide_drive_t(2)

```
366     special_t     special;      /* flags de activación especial */
367     select_t      select;      /* basic drive/head select reg value */
368
369     u8    keep_settings;       /* restaura características tras reseteo */
370     u8    using_dma;           /* disco usando DMA para lectura/escritura */
371     u8    retry_pio;
372     u8    state;
373     u8    waiting_for_dma;     /* usando DMA*/
374     u8    unmask;
375     u8    noflush;
376     u8    dsc_overlap;
377     u8    nice1;
378
```

Estructuras de Datos: ide_drive_t(3)

```
423       unsigned int    bios_cyl;       /* BIOS/fdisk/LILO number of cyls */
424       unsigned int    cyl;            /* "real" number of cyls */
425       unsigned int    drive_data;     /* used by set_pio_mode/selectproc */
426       unsigned int    failures;       /* current failure count */
427       unsigned int    max_failures;   /* maximum allowed failure count */
428       u64             probed_capacity;/* initial reported media capacity (ide-cd only
    currently) */
429
430       u64             capacity64;     /* Número total de sectores */
431
432       int             lun;            /* Unidad lógica */
433       int             crc_count;      /* crc counter to reduce drive speed */
434#ifdef CONFIG_BLK_DEV_IDEACPI
435       struct ide_acpi_drive_link *acpidata;
436#endif
437       struct list_head list;
438       struct device   gendev;
439       struct completion gendev_rel_comp;      /* to deal with device release() */
440} ide_drive_t;
```

Estructuras de Datos: ide_settings_t

La estructura siguiente contiene los campos para representar la característica en /proc.

```
664         typedef struct ide_settings_s {
665         char                *name;
666         int                 rw;
667         int                 data_type;
668         int                 min;
669         int                 max;
670         int                 mul_factor;
671         int                 div_factor;
672         void                *data;
673         ide_procset_t           *set;
674         int                 auto_remove;
675         struct ide_settings_s   *next;
676} ide_settings_t;
```

Código IDE

- El fichero de las funciones principales ide es /drivers/ide/ide.c

- Inicialización
- Entrada/Salida
- Control
- Finalización

Inicialización: ide_init

IDE_INIT()

El módulo del driver IDE es inicializado por ide_init:

Ide_init solamente es una interfaz para la inicialización, como todas las funciones de inicio de cualquier parte del kernel.

Llama a las funciones.

- ide_system_bus_speed()
- bus_register()
- init_ide_data(); ESTA ES LA PRINCIPAL
- Unas cuantas funciones extra si están activadas ciertas configuraciones
- probe_for_hwifs();
- Luego vienen un montón de funciones de configuración de características especiales que pueda tener el controlador IDE.
- proc_ide_create()

Inicialización: ide_init

```
1607static int __init ide_init(void)
1608{
1609        int ret;
1610
1611        printk(KERN_INFO "Uniform Multi-Platform E-IDE driver\n");
1612        system_bus_speed = ide_system_bus_speed();
1613
1614        printk(KERN_INFO "ide: Assuming %dMHz system bus speed "
1615                         "for PIO modes%s\n", system_bus_speed,
1616                        idebus_parameter ? "" : "; override with idebus=xx");
1617
1618        ret = bus_register(&ide_bus_type);
1619        if (ret < 0) {
1620                printk(KERN_WARNING "IDE: bus_register error: %d\n", ret);
1621                return ret;
1622        }
1623
1624        init_ide_data();
1625
1626        proc_ide_create();
1627
1628        return 0;
1629}
```

Inicialización: ide_init -> ide_system_bus_speed

- **IDE_SYSTEM_BUS_SPEED()**

  Como su nombre indica, calcula la velocidad a la que es capaz de ir el bus al que está conectado el controlador IDE (normalmente el bus PCI).
  La forma de especificar la velocidad de funcionamiento puede ser:

  - definida por el usuario
  - configurada en el bus PCI
  - tomando un valor por defecto.

Inicialización: ide_init -> ide_system_bus_speed

```
230 static int ide_system_bus_speed(void)
231{
232#ifdef CONFIG_PCI  /*si está definida la configuración del PCI la usamos.
233         static struct pci_device_id pci_default[] = {
234                 { PCI_DEVICE(PCI_ANY_ID, PCI_ANY_ID) },
235                 { }
236         };
237#else
238#define pci_default 0
239#endif /* CONFIG_PCI */
240
241         /* si está especificada por el usuario. */
242         if (idebus_parameter)
243                 return idebus_parameter;
244
245         /* safe default value for PCI or VESA and PCI*/
246         return pci_dev_present(pci_default) ? 33 : 50;
247}
```

Inicialización: ide_init -> bus_register

- **BUS_REGISTER()**

  Registra la información del bus en el sistema, esto es, da a conocer qué dispositivos y drivers van asociados a ese bus. Así pues, se encarga de decirle al sistema qué tipo de dispositivos (cdrom, hdd, etc...) están pinchados en ese bus y qué drivers los manejan.

Inicialización: ide_init -> init_ide_data

- **INIT_IDE_DATA()**

  La función principal de la inicialización es la que rellena todas las estructuras que contiene el "vector de controladores" (ide_hwifs). Directamente
  no hace nada, pero llama a las funciones que inicializan cada una de las posiciones de ese vector mediante las llamadas a:

  - ide_init_port_data()
  - init_hwif_default()
  - ide_init_default_irq()

Inicialización: ide_init -> init_ide_data

```
195 static void __init init_ide_data (void)
196{
197        ide_hwif_t *hwif;
198        unsigned int index;
199        static unsigned long magic_cookie = MAGIC_COOKIE;
200
201        if (magic_cookie != MAGIC_COOKIE)  /*para evitar más inicializaciones*/
202                return;         /* already initialized */
203        magic_cookie = 0;
204
205        /*Inicializa todas las estructuras de las HWIF */
206        for (index = 0; index < MAX_HWIFS; ++index) {
207                hwif = &ide_hwifs[index];
208                ide_init_port_data(hwif, index);
209                init_hwif_default(hwif, index);
210#if !defined(CONFIG_PPC32) || !defined(CONFIG_PCI)
211                hwif->irq =
212                        ide_init_default_irq(hwif->io_ports[IDE_DATA_OFFSET]);
213#endif
```

Inicialización: ide_init -> init_ide_data -> ide_init_port_data (anteriormente init_hwif_data)

IDE_INIT_PORT_DATA()

Inicializa todo el espacio de memoria que corresponde a una estructura ide_hwif_t (la que representa el controlador IDE en si mismo), básicamente lo que se hace es poner todo es espacio a 0 e ir rellenando después los campos. Las principales funciones aquí son:

- default_hwif_iops()
- default_hwif_transport()

Inicialización: ide_init -> init_ide_data -> ide_init_port_data

```
117 void ide_init_port_data(ide_hwif_t *hwif, unsigned int index)
116{
117        unsigned int unit;
118
119        /* Rellena con cero toda la estructura */
120        memset(hwif, 0, sizeof(ide_hwif_t));
121
122        /*rellenamos los campos oportunos*/
123        hwif->index     = index;
124        hwif->major     = ide_hwif_to_major[index];
125
126        hwif->name[0]   = 'i';
127        hwif->name[1]   = 'd';
128        hwif->name[2]   = 'e';
129        hwif->name[3]   = '0' + index;
130
131        hwif->bus_state = BUSSTATE_ON; /*especificamos que el bus está activo*/
132
133        init_completion(&hwif->gendev_rel_comp);
134
135        default_hwif_iops(hwif);
136        default_hwif_transport(hwif);
```

Inicialización: ide_init -> init_ide_data -> ide_init_port_data

```
137     for (unit = 0; unit < MAX_DRIVES; ++unit) {/*rellena información de los dispositivos*/
138             ide_drive_t *drive = &hwif->drives[unit];
139
140             drive->media                    = ide_disk;
141             drive->select.all               = (unit<<4)|0xa0;
142             drive->hwif                     = hwif;
143             drive->ctl                      = 0x08;
144             drive->ready_stat               = READY_STAT;
145             drive->bad_wstat                = BAD_W_STAT;
146             drive->special.b.recalibrate    = 1;
147             drive->special.b.set_geometry   = 1;
148             drive->name[0]                  = 'h';
149             drive->name[1]                  = 'd';
150             drive->name[2]                  = 'a' + (index * MAX_DRIVES) + unit;
151             drive->max_failures             = IDE_DEFAULT_MAX_FAILURES;
152             drive->using_dma                = 0;
153             drive->vdma                     = 0;
154             INIT_LIST_HEAD(&drive->list);
155             init_completion(&drive->gendev_rel_comp);
156     }
157}
```

Inicialización: init_hwif_data -> default_hwif_iops

Se inicializa las funciones que leerán y escribirán datos desde y hacia el controlador.

```
80 void default_hwif_iops (ide_hwif_t *hwif)
81{
82        hwif->OUTB      = ide_outb;
83        hwif->OUTBSYNC  = ide_outbsync;
84        hwif->OUTW      = ide_outw;
85        hwif->OUTSW     = ide_outsw;
86        hwif->OUTSL     = ide_outsl;
87        hwif->INB       = ide_inb;
88        hwif->INW       = ide_inw;
89        hwif->INSW      = ide_insw;
90        hwif->INSL      = ide_insl;
91}
```

Inicialización: init_hwif_data-> default_hwif_transport

A diferencia de las funciones anteriores, éstas no trabajan sobre los datos que trasiegan entre la CPU y un dispositivo (o su controlador), sino que tratan el control de la transferencia de los datos.

Para un correcto funcionamiento del sistema, siempre se debería llamar a una de estas funciones antes de llamar a las otras.

```
272 void default_hwif_transport(ide_hwif_t *hwif)
273{
274        hwif->ata_input_data           = ata_input_data;
275        hwif->ata_output_data          = ata_output_data;
276        hwif->atapi_input_bytes        = atapi_input_bytes;
277        hwif->atapi_output_bytes       = atapi_output_bytes;
278}
```

Inicialización: ide_init -> init_ide_data -> init_hwif_default

INIT_HWIF_DEFAULT()

Recordemos que los dispositivos tiene unos registros de información sobre el mismo (vendor_id, product_id, etc...). Esta función los lee y los inicializa en una estructura del sistema que representa esos registros (hw_regs_t). La función que hace esto es:

- ide_init_hwif_ports()

Primero se inicializa todo el espacio de memoria de hw_regs_t a 0 y luego se llama a esta función para que rellene la información de la que el dispositivo disponga (no siempre se llena todo).

Inicialización: ide_init -> init_ide_data -> init_hwif_default

```
160static void init_hwif_default(ide_hwif_t *hwif, unsigned int index)
161{
162        hw_regs_t hw;
163
164        memset(&hw, 0, sizeof(hw_regs_t));
165
166        ide_init_hwif_ports(&hw, ide_default_io_base(index), 0, &hwif->irq);
167
168        memcpy(hwif->io_ports, hw.io_ports, sizeof(hw.io_ports));
169
170        hwif->noprobe = !hwif->io_ports[IDE_DATA_OFFSET];
171#ifdef CONFIG_BLK_DEV_HD
172        if (hwif->io_ports[IDE_DATA_OFFSET] == HD_DATA)
173                hwif->noprobe = 1;      /* may be overridden by ide_setup() */
174#endif
175}
```

Inicialización: ide_init -> init_ide_data -> init_hwif_ports

```
223#ifdef CONFIG_IDE_ARCH_OBSOLETE_INIT
224static inline void ide_init_hwif_ports(hw_regs_t *hw,
225                                       unsigned long io_addr,
226                                       unsigned long ctl_addr,
227                                       int *irq)
228{
229        if (!ctl_addr)
230                ide_std_init_ports(hw, io_addr, ide_default_io_ctl(io_addr));
231        else
232                ide_std_init_ports(hw, io_addr, ctl_addr);
233
234        if (irq)
235                *irq = 0;
236
237        hw->io_ports[IDE_IRQ_OFFSET] = 0;
238
239#ifdef CONFIG_PPC32
240        if (ppc_ide_md.ide_init_hwif)
241                ppc_ide_md.ide_init_hwif(hw, io_addr, ctl_addr, irq);
242#endif
```

Inicialización: ide_init -> init_ide_data -> init_hwif_ports

```
243}
 244#else
 245static inline void ide_init_hwif_ports(hw_regs_t *hw,
 246                                       unsigned long io_addr,
 247                                       unsigned long ctl_addr,
 248                                       int *irq)
 249{
 250        if (io_addr || ctl_addr)
 251                printk(KERN_WARNING "%s: must not be called\n", __FUNCTION__);
 252}
```

Inicialización: ide_init -> init_ide_data -> ide_init_default_irq

IDE_INIT_DEFAULT_IRQ()

Asigna una interrupción que aún no esté tomada al controlador que se configura. Obsérvese que esta interrupción es compartida por ambos dispositivos que pueden estar conectados.

Inicialización: ide_init -> init_ide_data -> proc_ide_create

- **PROC_IDE_CREATE()**

  Crea una entrada para este controlador en /proc/ide. Tienen la forma /proc/ide/ide0, /proc/ide/ide1, etc...

Inicialización: ide_init -> proc_ide_create

```
861void proc_ide_create(void)
862{
863        struct proc_dir_entry *entry;
864
865        proc_ide_root = proc_mkdir("ide", NULL);
866
867        if (!proc_ide_root)
868                return;
869
870        entry = create_proc_entry("drivers", 0, proc_ide_root);
871        if (entry)
872                entry->proc_fops = &ide_drivers_operations;
873}
```

Entrada/Salida

- Existen estructuras y colas para las peticiones de entrada/salida hacia/desde un dispositivo.

  Para una petición sobre un dispositivo IDE se tienen las funciones.

  - ide_hwif_request_regions()
  - hwif_request_region()

- Estas funciones mueven información desde disco/memoria hacia memoria/disco.

E/S: ide_hwif_request_regions

```
294 int ide_hwif_request_regions(ide_hwif_t *hwif)
295{
296       unsigned long addr;
297       unsigned int i;
298
299       if (hwif->mmio)/* si usa mapeo de memoria salimos*/
300             return 0;
301       addr = hwif->io_ports[IDE_CONTROL_OFFSET];
302       if (addr && !hwif_request_region(hwif, addr, 1))
303             goto control_region_busy;
304       hwif->straight8 = 0;
305       addr = hwif->io_ports[IDE_DATA_OFFSET];
306       if ((addr | 7) == hwif->io_ports[IDE_STATUS_OFFSET]) {
307             if (!hwif_request_region(hwif, addr, 8))
308                   goto data_region_busy;
309             hwif->straight8 = 1;
310             return 0;
311       }
```

E/S: ide_hwif_request_regions

```
312     for (i = IDE_DATA_OFFSET; i <= IDE_STATUS_OFFSET; i++) {
313             addr = hwif->io_ports[i];
314             if (!hwif_request_region(hwif, addr, 1)) {
315                     while (--i)
316                             release_region(addr, 1);
317                     goto data_region_busy;
318             }
319     }
320     return 0;
321
322data_region_busy:
323     addr = hwif->io_ports[IDE_CONTROL_OFFSET];
324     if (addr)
325             release_region(addr, 1);
326control_region_busy:
327     /* If any errors are return, we drop the hwif interface. */
328     return -EBUSY;
329}
```

E/S: ide_hwif_request_regions-> hwif_request_region

```
273 static struct resource* hwif_request_region(ide_hwif_t *hwif,
274                                               unsigned long addr, int num)
275{
276       struct resource *res = request_region(addr, num, hwif->name);
277
278       if (!res)
279             printk(KERN_ERR "%s: I/O resource 0x%lX-0x%lX not free.\n",
280                               hwif->name, addr, addr+num-1);
281       return res;
282}
```

Control

- Las funciones de control nos permiten añadir, modificar, eliminar y buscar características de los dispositivos IDE.

  ide_add_setting

  __ide_remove_setting

  ide_find_setting_by_ioctl

  ide_find_setting_by_name

  auto_remove_settings

  ide_read_setting

  ide_write_setting

  ide_add_generic_settings (pone características por defecto)

  generic_ide_ioctl

Control: ide_add_setting

Esta función a través de __ide_add_setting añade la característica que se representará en /proc

```
187int ide_add_setting(ide_drive_t *drive, const char *name, int rw, int data_type, int min, int max, int mul_factor, int div_factor, void *data, ide_procset_t *set)
188{
189    return __ide_add_setting(drive, name, rw, data_type, min, max, mul_factor, div_factor, data, set, 1);
190}
```

Control: ide_add_setting-> __ide_add_setting(1)

```
154 static int __ide_add_setting(ide_drive_t *drive, const char *name, int rw, int data_type, int
min, int max, int mul_factor, int div_factor, void *data, ide_procset_t *set, int auto_remove)
 155{
 156        ide_settings_t **p = (ide_settings_t **) &drive->settings, *setting = NULL;
 157
 158        mutex_lock(&ide_setting_mtx);
 159        while ((*p) && strcmp((*p)->name, name) < 0)
 160                p = &((*p)->next);
 161        if ((setting = kzalloc(sizeof(*setting), GFP_KERNEL)) == NULL)
 162                goto abort;
 163        if ((setting->name = kmalloc(strlen(name) + 1, GFP_KERNEL)) == NULL)
 164                goto abort;
```

Control: ide_add_setting-> __ide_add_setting(2)

```
165        strcpy(setting->name, name);
166        setting->rw = rw;
167        setting->data_type = data_type;
168        setting->min = min;
169        setting->max = max;
170        setting->mul_factor = mul_factor;
171        setting->div_factor = div_factor;
172        setting->data = data;
173        setting->set = set;
174
175        setting->next = *p;
176        if (auto_remove)
177                setting->auto_remove = 1;
178        *p = setting;
179        mutex_unlock(&ide_setting_mtx);
180        return 0;
181abort:
182        mutex_unlock(&ide_setting_mtx);
183        kfree(setting);
184        return -1;
185}
```

Control: __ide_remove_setting

```
203static void __ide_remove_setting (ide_drive_t *drive, char *name)
204{
205        ide_settings_t **p, *setting;
206
207        p = (ide_settings_t **) &drive->settings;
208
209        while ((*p) && strcmp((*p)->name, name))
210                p = &((*p)->next);
211        if ((setting = (*p)) == NULL)
212                return;
213
214        (*p) = setting->next;
215
216        kfree(setting->name);
217        kfree(setting);
218}
```

Control: ide_add_setting-> auto_remove_setting

```
229 static void auto_remove_settings (ide_drive_t *drive)
230{
231        ide_settings_t *setting;
232repeat:
233        setting = drive->settings;
234        while (setting) {
235                if (setting->auto_remove) {
236                        __ide_remove_setting(drive, setting->name);
237                        goto repeat;
238                }
239                setting = setting->next;
240        }
241}
```

Finalización: ide_unregister(1)

```
521void ide_unregister(unsigned int index, int init_default, int restore)
522{
523        ide_drive_t *drive;
524        ide_hwif_t *hwif, *g;
525        static ide_hwif_t tmp_hwif; /* protected by ide_cfg_mtx */
526        ide_hwgroup_t *hwgroup;
527        int irq_count = 0, unit;
528
529        BUG_ON(index >= MAX_HWIFS);
530
531        BUG_ON(in_interrupt());
532        BUG_ON(irqs_disabled());
533        mutex_lock(&ide_cfg_mtx);
534        spin_lock_irq(&ide_lock);
535        hwif = &ide_hwifs[index];
```

Finalización: ide_unregister(2)

```
536     if (!hwif->present)
537             goto abort;
538     for (unit = 0; unit < MAX_DRIVES; ++unit) {
539             drive = &hwif->drives[unit];
540             if (!drive->present)
541                     continue;
542             spin_unlock_irq(&ide_lock);
543             device_unregister(&drive->gendev);
544             wait_for_completion(&drive->gendev_rel_comp);
545             spin_lock_irq(&ide_lock);
546     }
547     hwif->present = 0;
548
549     spin_unlock_irq(&ide_lock);
550
551     ide_proc_unregister_port(hwif);
```

Finalización: ide_unregister(3)

```
553 hwgroup = hwif->hwgroup;
554         /*
555          * free the irq if we were the only hwif using it
556          */
557         g = hwgroup->hwif;
558         do {
559                 if (g->irq == hwif->irq)
560                         ++irq_count;
561                 g = g->next;
562         } while (g != hwgroup->hwif);
563         if (irq_count == 1)
564                 free_irq(hwif->irq, hwgroup);
565
566         ide_remove_port_from_hwgroup(hwif);
567
568         device_unregister(&hwif->gendev);
569         wait_for_completion(&hwif->gendev_rel_comp);
571         /*
572          * Remove us from the kernel's knowledge
573          */
574         blk_unregister_region(MKDEV(hwif->major, 0), MAX_DRIVES<<PARTN_BITS);
575         kfree(hwif->sg_table);
576         unregister_blkdev(hwif->major, hwif->name);
577         spin_lock_irq(&ide_lock);
```

Finalización: ide_unregister(4)

```
579         if (hwif->dma_base) {
580                 (void) ide_release_dma(hwif);
581
582                 hwif->dma_base = 0;
583                 hwif->dma_command = 0;
584                 hwif->dma_vendor1 = 0;
585                 hwif->dma_status = 0;
586                 hwif->dma_vendor3 = 0;
587                 hwif->dma_prdtable = 0;
588
589                 hwif->extra_base  = 0;
590                 hwif->extra_ports = 0;
591         }
592
593         ide_hwif_release_regions(hwif);
594
```

Finalización: ide_unregister(5)

```
595         /* copy original settings */
596         tmp_hwif = *hwif;
597
598         /* restore hwif data to pristine status */
599         ide_init_port_data(hwif, index);
600
601         if (init_default)
602                 init_hwif_default(hwif, index);
603
604         if (restore)
605                 ide_hwif_restore(hwif, &tmp_hwif);
606
607abort:
608         spin_unlock_irq(&ide_lock);
609         mutex_unlock(&ide_cfg_mtx);
610}
```

FIN

Bibliografía:

- Linux Cross Reference: lxr.linux.no
- Google
- Documentación DSO disco ide 06/07