Lección IDE

1. **Introducción**

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 diseñó 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.

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 existe el llamado PATA (Parallel ATA), si bien lo desarrollan los mismos que ATA.

El estándar ATA define:

- Descripción de la interfaz y el bus
- Modos y velocidad de transferencia de datos.
- Forma de direccionar los sectores del HD en el direccionamiento lógico LBA
- Define los modos de transferencia de E/S programadas (PIO)
- Se define la comprobación de redundancia cíclica (CRC)
- Descripción y definición de los comandos de interfaz
- Soporte para los modos de transferencia DMA y DMA multipalabra.

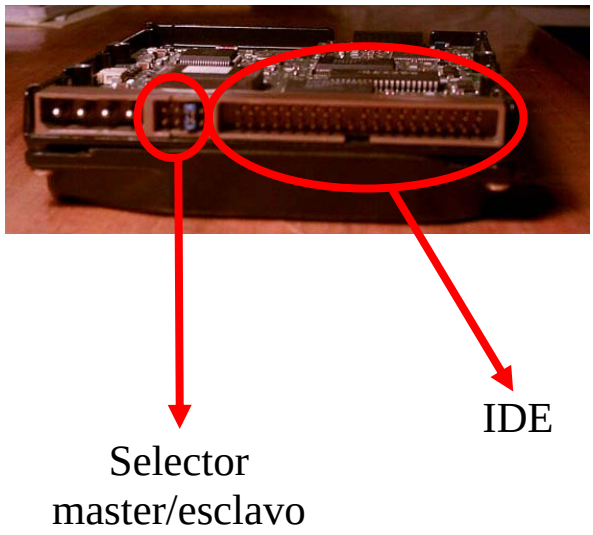
Reiterando lo dicho, ATA es una interfaz de conexión, y su principal característica es que el controlador no tiene por qué estar integrado en la placa base.

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.

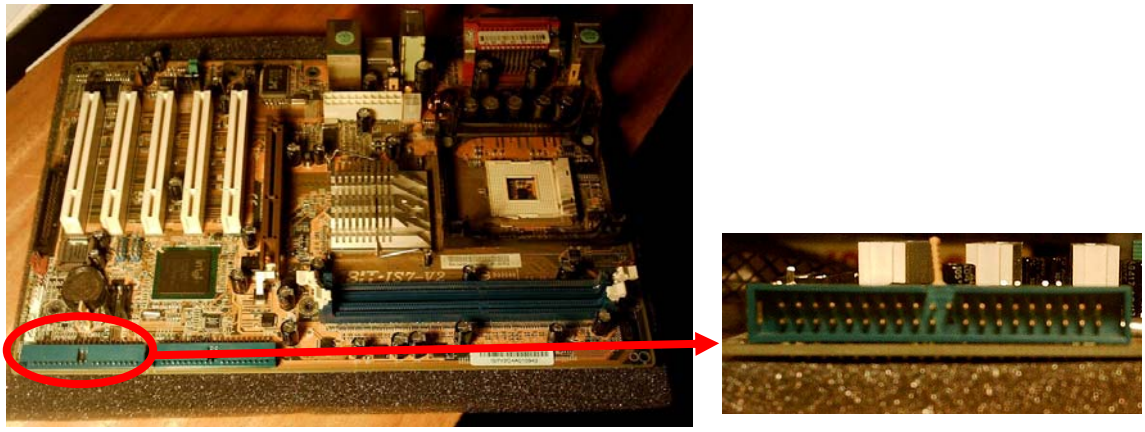
No puede haber más de un dispositivo funcionando en el mismo controlador.

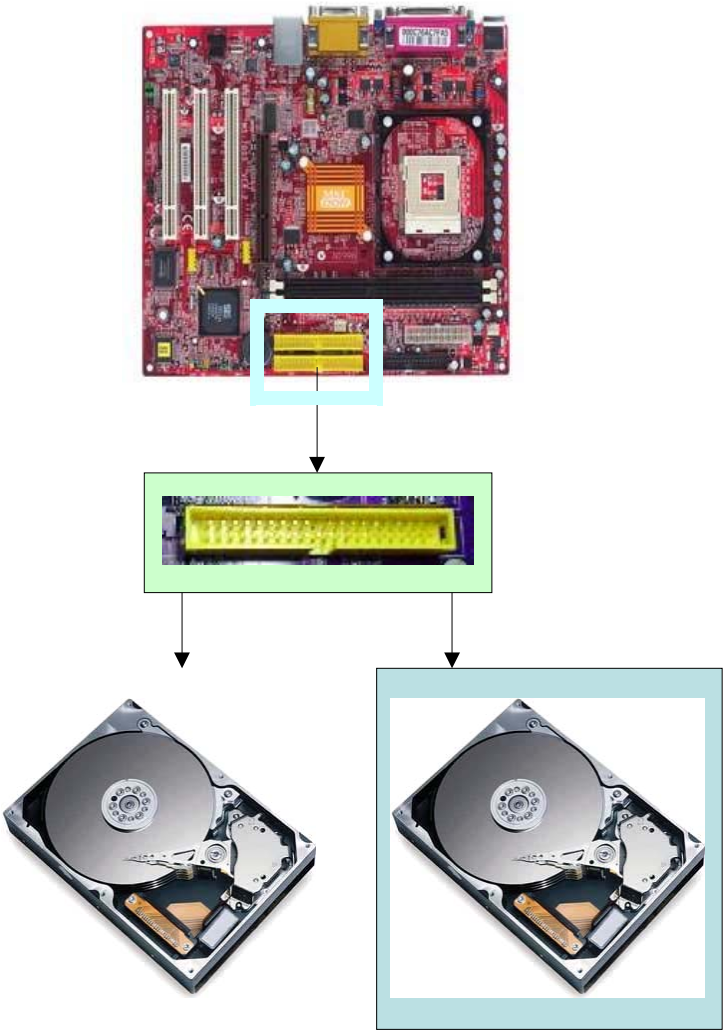
Los discos duros



El bus de datos y el IDE en la placa base

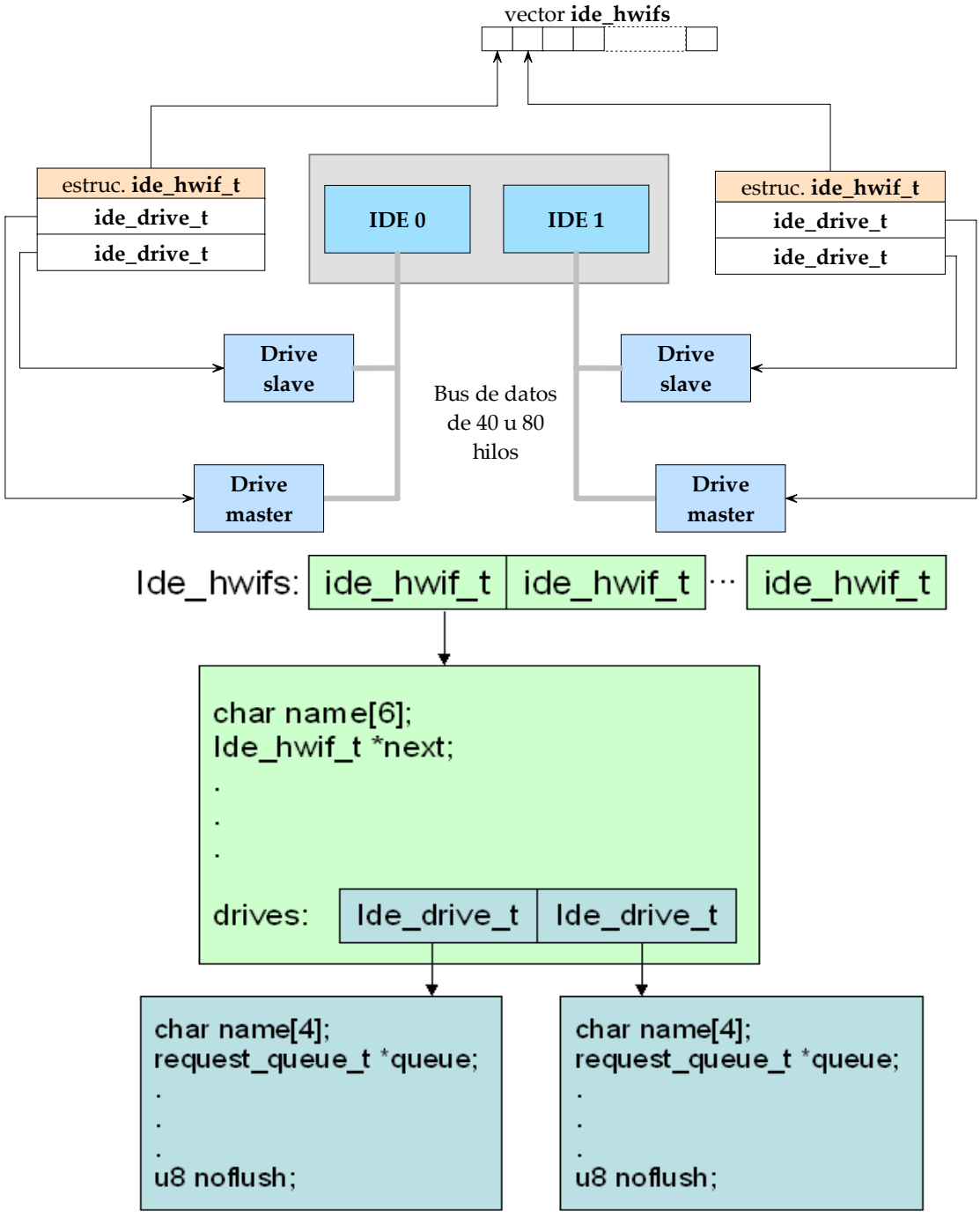
Interfaz IDE en la placa base





2. Estructuras de datos

Las estructuras de datos están definidas en /include/linux/ide.h.



ide_hwif_t

Esta estructura es la que representa al controlador IDE. Dentro de ella se encuentra el vector de dispositivos llamado drives (del tipo ide_drive_t) que como indica el estándar ATA como mucho pueden ser 2 para cada controlador, según la macro MAX_DRIVES. Se encuentra en linux/include/linux/ide.h

struct hwif_s (que se renombrará como **ide_hwif_t**, según la última línea)

```
450typedef struct hwif_s {
 451        struct hwif_s *next;           /* para mantener lista encadenada*/
 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 (por ejemplo, “ide0”)*/
 457
 458
 459        unsigned long   io_ports[IDE_NR_PORTS]; /*puertos de E/S */
 460        unsigned long   sata_scr[SATA_NR_PORTS];
 461
 462        ide_drive_t     drives[MAX_DRIVES];    /* dispositivos */
 463
 464        u8 major;
 465        u8 index;
 466        u8 channel;
 467        u8 bus_state;
 468
 469        u32 host_flags;
 470
 471        u8 pio_mask;
 472
 473        u8 ultra_mask;
 474        u8 mwdma_mask;
 475        u8 swdma_mask;
 476
 477        u8 cbl;         /* tipo de cable */
 478
 479        hwif_chipset_t chipset; /* submódulo para personalización*/
 480
 481        struct device *dev;
 482
 483        const struct ide_port_info *cds;        /* estructura del chipset del dispositivo */
 484
 485        ide_ack_intr_t *ack_intr;
 486
 487        void (*rw_disk)(ide_drive_t *, struct request *);
 488
 489#if 0
 490        ide_hwif_ops_t  *hwifops;
 491#else
 492
 493        void    (*port_init_devs)(struct hwif_s *);
 494
 495        void    (*set_pio_mode)(ide_drive_t *, const u8);
 496
 497        void    (*set_dma_mode)(ide_drive_t *, const u8);
```

```
498
499         void    (*selectproc)(ide_drive_t *);
500
501         int     (*reset_poll)(ide_drive_t *);
502
503         void    (*pre_reset)(ide_drive_t *);
504
505         void    (*resetproc)(ide_drive_t *);
506
507         void    (*maskproc)(ide_drive_t *, int);
508
509         void    (*quirkproc)(ide_drive_t *);
510
511         int     (*busproc)(ide_drive_t *, int);
512#endif
513         u8 (*mdma_filter)(ide_drive_t *);
514         u8 (*udma_filter)(ide_drive_t *);
515
516         u8 (*cable_detect)(struct hwif_s *);
517
518         void (*ata_input_data)(ide_drive_t *, void *, u32);
519         void (*ata_output_data)(ide_drive_t *, void *, u32);
520
521         void (*atapi_input_bytes)(ide_drive_t *, void *, u32);
522         void (*atapi_output_bytes)(ide_drive_t *, void *, u32);
523
524         void (*dma_host_set)(ide_drive_t *, int);
525         int (*dma_setup)(ide_drive_t *);
526         void (*dma_exec_cmd)(ide_drive_t *, u8);
527         void (*dma_start)(ide_drive_t *);
528         int (*ide_dma_end)(ide_drive_t *drive);
529         int (*ide_dma_test_irq)(ide_drive_t *drive);
530         void (*ide_dma_clear_irq)(ide_drive_t *drive);
531         void (*dma_lost_irq)(ide_drive_t *drive);
532         void (*dma_timeout)(ide_drive_t *drive);
533
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);
544
545
546         unsigned int    *dmatable_cpu;
547
548         dma_addr_t      dmatable_dma;
549
550         struct scatterlist *sg_table;
551         int sg_max_nents;               /* Número máximo de entradas*/
552         int sg_nents;                   /* Número actual de entradas*/
553         int sg_dma_direction;           /* Dirección de transferencia del DMA*/
554
555         /* fase de datos del comando activo */
556         int             data_phase;
557
```

```
558       unsigned int nsect;
559       unsigned int nleft;
560       struct scatterlist *cursg;
561       unsigned int cursg_ofs;
562
563       int           rqsize;        /* máximos sectores por solicitud*/
564       int           irq;           /* nuestro número IRQ*/
565
566       unsigned long   dma_base;      /* dirección base para puertos de DMA*/
567       unsigned long   dma_command;
568       unsigned long   dma_vendor1;
569       unsigned long   dma_status;    /* registro de estado del DMA*/
570       unsigned long   dma_vendor3;
571       unsigned long   dma_prdtable;
572
573       unsigned long   config_data;
574       unsigned long   select_data;
575
576       unsigned long   extra_base;    /* dirección de puertos DMA extra*/
577       unsigned        extra_ports;   /* número de puertos DMA extra*/
578
579       unsigned        noprobe    : 1;
580       unsigned        present    : 1;
581       unsigned        hold       : 1;
582       unsigned        serialized : 1;
583       unsigned        sharing_irq: 1;
584       unsigned        reset      : 1;
585       unsigned        sg_mapped  : 1;
586       unsigned        mmio       : 1;
587       unsigned        straight8  : 1;
588
589       struct device   gendev;
590       struct completion gendev_rel_comp;
591
592       void           *hwif_data;
593
594       unsigned dma;
595
596#ifdef CONFIG_BLK_DEV_IDEACPI
597       struct ide_acpi_hwif_link *acpidata;
598#endif
599} ____cacheline_internodealigned_in_smp ide_hwif_t;
```

ide_drive_t

Esta estructura representa un dispositivo de conexión IDE, ya sea CD-ROM, cinta, disco duro, etc. Se encuentra en linux/include/linux/ide.h

struct ide_drive_s (que se renombrará como **ide_drive_t**, según la última línea)

Además contiene los siguientes atributos correspondientes a las unidades:
- Nombre
- Identificación del modelo de unidad ide

- Entrada al directorio /proc/ide/ proc y su configuración settings
- Tiempo después del cual pasa a estado de reposo, “sleep”, si el estado de la unidad es de reposo, “sleeping”
- Tiempo en el que se realizó la última petición service_start
- Tiempo máximo de espera por la IRQ timeout
- Valores para restaurar la unidad después de un reset keep_settings
- Uso de DMA autodma, using_dma, waiting_for_dma
- Índice de velocidad de transferencia al iniciar el sistema init_speed y velocidad de transferencia actual current_speed
- Cabezas, sectores y cilindros detectados por la bios (bios_head, bios_sect, bios_cyl) y reales head, sect y cyl

```
345typedef 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 dispositivo*/
 355#ifdef CONFIG_IDE_PROC_FS
 356       struct proc_dir_entry *proc;    /* /proc/ide/ → directorio de especificaciones */
 357       struct ide_settings_s *settings;/* /proc/ide/ →características de dispositivos */
 358#endif
 359       struct hwif_s          *hwif;
 360
 361       unsigned long sleep;           /* dormir hasta darse el valor “sleep” */
 362       unsigned long service_start;    /* tiempo en que empezó última solicitud*/
 363       unsigned long service_time;     /* tiempo del servicio de la última solicitud*/
 364       unsigned long timeout;          /* tiempo de espera máximo por IRQ*/
 365
 366       special_t      special;        /* flags de activación especial*/
 367       select_t       select;
 368
 369       u8      keep_settings;         /* restaura las características tras un 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
 379       unsigned present       : 1;    /* el dispositivo está físicamente presente*/
 380       unsigned dead          : 1;
 381       unsigned id_read       : 1;
 382       unsigned noprobe       : 1;
 383       unsigned removable     : 1;
 384       unsigned attach        : 1;    /* requerido para dispositivos extraíbles */
 385       unsigned forced_geom    : 1;
 386       unsigned no_unmask      : 1;
 387       unsigned no_io_32bit    : 1;
 388       unsigned atapi_overlap  : 1;
 389       unsigned doorlocking    : 1;
```

```
390       unsigned nodma         : 1;    /* no permite DMA*/
391       unsigned autotune      : 2;
392       unsigned remap_0_to_1   : 1;
393       unsigned blocked       : 1;
394       unsigned vdma          : 1;
395       unsigned scsi          : 1;
396       unsigned sleeping      : 1;
397       unsigned post_reset    : 1;
398       unsigned udma33_warned  : 1;
399
400       u8      addressing;
401       u8      quirk_list;
402       u8      init_speed;
403       u8      current_speed;
404       u8      desired_speed;
405       u8      dn;
406       u8      wcache;         /* estado de cache de escritura*/
407       u8      acoustic;
408       u8      media;          /* disco, cdrom…*/
409       u8      ctl;
410       u8      ready_stat;
411       u8      mult_count;
412       u8      mult_req;
413       u8      tune_req;
414       u8      io_32bit;
415       u8      bad_wstat;
416       u8      nowerr;
417       u8      sect0;
418       u8      head;
419       u8      sect;
420       u8      bios_head;
421       u8      bios_sect;
422
423       unsigned int    bios_cyl;
424       unsigned int    cyl;
425       unsigned int    drive_data;
426       unsigned int    failures;       /* contador de fallos actual */
427       unsigned int    max_failures;   /* número máximo de fallos permitido */
428       u64             probed_capacity;
429
430       u64             capacity64;     /* número total de sectores*/
431
432       int             lun;            /* unidad lógica*/
433       int             crc_count;      /* contador “crc” para reducir la velocidad del dispositivo */
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;      /* para el tratamiento de la extracción del
dispositivo*/
440} ide_drive_t;
```

ide_settings_t

Esta estructura representa una característica que pueda tener un dispositivo IDE.

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

```
664typedef 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;
```

ide_hwgroup_t

Añade funciones y mantiene organizados los dispositivos. Podría no considerarse tan relevante como las ya vistas, en tanto que se trata de una estructura auxiliar; sin embargo, su cometido es indispensable.

```
610 typedef struct hwgroup_s {
611
612       ide_startstop_t (*handler)(ide_drive_t *);
613
614
615       volatile int busy;
616
617       unsigned int sleeping   : 1;
618
619       unsigned int polling    : 1;
620
621       unsigned int resetting  : 1;
622
623             /* dispositivo actual*/
624       ide_drive_t *drive;
625             /* puntero para el hwif actual en la lista encadenada*/
626       ide_hwif_t *hwif;
627
628             /* petición actual*/
629       struct request *rq;
630
631
632       struct timer_list timer;
633
634       unsigned long poll_timeout;
```

```
635
636         int (*expiry)(ide_drive_t *);
637
638         int req_gen;
639         int req_gen_timer;
640} ide_hwgroup_t;
```

Funciones

El fichero con las funciones principales es /drivers/ide/ide.c

A continuación dividiremos el recorrido por el código en tres secciones:

- Inicialización
- Entrada Salida
- Control

3. Funciones de Inicialización

En esta etapa se inicializan las estructuras de datos y funciones asociadas con algunas que proporciona el sistema por defecto. Es ya una vez inicializado cuando se le puede cambiar al dispositivo las funciones por defecto por otras específicas del dispositivo o del controlador.

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.

Así pues, se encarga de llamar a las funciones:

- ide_system_bus_speed().
- bus_register().
- init_ide_data(); esta función es la que realiza la inicialización.
- Funciones propias de ciertas configuraciones.

- probe_for_hwifs().
- Funciones características especiales que pueda tener el controlador IDE.
- proc_ide_create()

```
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(); /*Función principal en la inicialización*/
1625
1626        proc_ide_create();
1627
1628        return 0;
1629}
```

ide_system_bus_speed

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.

```
230static int ide_system_bus_speed(void)
 231{
 232#ifdef CONFIG_PCI
 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
242       if (idebus_parameter)
243             return idebus_parameter;
244
245
246       return pci_dev_present(pci_default) ? 33 : 50;
247}
```

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.

La función se encuentra en /drivers/base/bus.c, y es la que sigue:

```
871int bus_register(struct bus_type *bus)
 872{
 873       int retval;
 874       struct bus_type_private *priv;
 875
 876       priv = kzalloc(sizeof(struct bus_type_private), GFP_KERNEL);
 877       if (!priv)
 878             return -ENOMEM;
 879
 880       priv->bus = bus;
 881       bus->p = priv;
 882
 883       BLOCKING_INIT_NOTIFIER_HEAD(&priv->bus_notifier);
 884
 885       retval = kobject_set_name(&priv->subsys.kobj, "%s", bus->name);
 886       if (retval)
 887             goto out;
 888
 889       priv->subsys.kobj.kset = bus_kset;
 890       priv->subsys.kobj.ktype = &bus_ktype;
 891       priv->drivers_autoprobe = 1;
 892
 893       retval = kset_register(&priv->subsys);
 894       if (retval)
 895             goto out;
 896
 897       retval = bus_create_file(bus, &bus_attr_uevent);
 898       if (retval)
 899             goto bus_uevent_fail;
 900
 901       priv->devices_kset = kset_create_and_add("devices", NULL,
 902                                                &priv->subsys.kobj);
 903       if (!priv->devices_kset) {
 904             retval = -ENOMEM;
 905             goto bus_devices_fail;
 906       }
 907
 908       priv->drivers_kset = kset_create_and_add("drivers", NULL,
```

```
909                                     &priv->subsys.kobj);
910     if (!priv->drivers_kset) {
911             retval = -ENOMEM;
912             goto bus_drivers_fail;
913     }
914
915     klist_init(&priv->klist_devices, klist_devices_get, klist_devices_put);
916     klist_init(&priv->klist_drivers, NULL, NULL);
917
918     retval = add_probe_files(bus);
919     if (retval)
920             goto bus_probe_files_fail;
921
922     retval = bus_add_attrs(bus);
923     if (retval)
924             goto bus_attrs_fail;
925
926     pr_debug("bus: '%s': registered\n", bus->name);
927     return 0;
928
929bus_attrs_fail:
930     remove_probe_files(bus);
931bus_probe_files_fail:
932     kset_unregister(bus->p->drivers_kset);
933bus_drivers_fail:
934     kset_unregister(bus->p->devices_kset);
935bus_devices_fail:
936     bus_remove_file(bus, &bus_attr_uevent);
937bus_uevent_fail:
938     kset_unregister(&bus->p->subsys);
939     kfree(bus->p);
940out:
941     return retval;
942}
```

init_ide_data

Siendo la función principal de la inicialización, es la que rellena todas las estructuras que contiene el "vector de controladores" (ide_hwifs). Lo hace llamando a las funciones que inicializan cada una de las posiciones de ese vector mediante las llamadas a:

- init_hwif_data()
- init_hwif_default()
- ide_init_default_irq()

```
194#define MAGIC_COOKIE 0x12345678
195static 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)
202             return;         /* ya inicializado*/
203       magic_cookie = 0;
204
205       /* Inicializa todas las estructuras de la interfaz*/
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
214       }
215}
```

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()

```
115void ide_init_port_data(ide_hwif_t *hwif, unsigned int index)
116{
117       unsigned int unit;
118
119  /* rellenamos todo el espacio de memoria de la estructura con ceros*/
120       memset(hwif, 0, sizeof(ide_hwif_t));
121
122       /* rellenamos los campos iniciales distintos de cero */
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; //ponemos que el bus está activo
132
133       init_completion(&hwif->gendev_rel_comp); //inicializa los locks del controlador
134
135       default_hwif_iops(hwif);
136       default_hwif_transport(hwif);
/*inicializa cada dispositivo del controlador con valor  por defecto*/
137       for (unit = 0; unit < MAX_DRIVES; ++unit) {
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}
```

default_hwif_iops

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

```
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}
```

default_hwif_transport

A diferencia de las funciones anteriores, estas 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.

```
272void 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}
```

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).

```
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}
```

ide_init_hwif_ports

```
#ifdef CONFIG_IDE_ARCH_OBSOLETE_INIT //si la arquitectura es obsoleta se llama a esta parte
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
243}
244#else
```

```
245static inline void ide_init_hwif_ports(hw_regs_t *hw, //parte que se llama si no es obsoleta
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}
253#endif /* CONFIG_IDE_ARCH_OBSOLETE_INIT */
```

IDE_INIT_DEFAULT_IRQ

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

proc_ide_create

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

```
859void proc_ide_create(void)
860{
861        struct proc_dir_entry *entry;
862
863        proc_ide_root = proc_mkdir("ide", NULL);
864
865        if (!proc_ide_root)
866                return;
867
868        entry = create_proc_entry("drivers", 0, proc_ide_root);
869        if (entry)
870                entry->proc_fops = &ide_drivers_operations;
871}
```

4. Funciones de Entrada/Salida

A continuación veremos las funciones encargadas de la entrada/salida. El funcionamiento consiste en pedir regiones a memoria, una vez en memoria hacemos las lecturas y escrituras oportunas y el sistema se encargará de llevar los cambios a su destino.

Existen estructuras y colas para las peticiones de entrada/salida hacia/desde un dispositivo. Para una petición sobre un dispositivo IDE se tiene la función:

- ide_hwif_request_regions()

ide_hwif_request_regions

Esta función realiza comprobaciones para evitar errores y llama a la función hwif_request_region.

```
294int ide_hwif_request_regions(ide_hwif_t *hwif)
295{
296	unsigned long addr;
297	unsigned int i;
298
299	if (hwif->mmio) //si usamos 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	}
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	/*si ocurre algún error devolvemos –EBUSY*/
328	return -EBUSY;
329}
```

hwif_request_region

Esta función solicita la región mostrando un error si no lo consigue.

```
273static 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}
```

5. Funciones de control

Las funciones de control nos permiten añadir, modificar, eliminar y buscar características de los dispositivos IDE. Algunas de las funciones de control son:

- 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

Nosotros sólo veremos algunas de ellas.

ide_add_setting

Esta función a través de __ide_add_setting añade la característica que se representará en /proc. Esta función lo único que hace es devolver el resultado de la llamada a __ide_add_settings

```
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}
```

__ide_add_setting

Inicializa los campos de la estructura ide_settings_t y la enlaza en la lista de características del dispositivo.

```
154static 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;
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}
```

__ide_remove_setting

Busca la característica a eliminar por el nombre, enlaza la cola de características sin la que va a ser eliminada y libera los recursos de la característica.

```
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}
```

auto_remove_setting

Elimina todas las características que tienen habilitado el flag de auto_remove.

```
229static 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}
```

ide_unregister

Libera los recursos utilizados por el controlador y los dispositivos conectados a este.

```
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;
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];
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);
552
553	hwgroup = hwif->hwgroup;
554	/*
555	 * libera la irq si el controlador fuera el único hwif que la usa.
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);
570
571	/*
572	 * * borra de los archivos del Núcleo
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);
578
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
595       /* reinicia las características iniciales */
596       tmp_hwif = *hwif;
597
598       /* recupera los datos hwif al estado primitivo */
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}
```

6. Bibliografía

→ Linux Cross Reference: lxr.linux.no

→ Google