```
/*
 * This file contains the driver for an XT hard disk controller
 * (at least the DTC 5150X) for Linux.
 *
 * Author: Pat Mackinlay, pat@it.com.au
 * Date: 29/09/92
 *
 * Revised: 01/01/93, ...
 *
 * Ref: DTC 5150X Controller Specification (thanks to Kevin Fowler,
 *   kevinf@agora.rain.com)
 * Also thanks to: Salvador Abreu, Dave Thaler, Risto Kankkunen and
 *   Wim Van Dorst.
 *
 * Revised: 04/04/94 by Risto Kankkunen
 *   Moved the detection code from xd_init() to xd_geninit() as it
 *   needed interrupts enabled and Linus didn't want to enable them in
 *   that first phase. xd_geninit() is the place to do these kinds of
 *   things anyway, he says.
 *
 * Modularized: 04/10/96 by Todd Fries, tfries@umr.edu
 *
 * Note: pared down for USENIX example source code
 */

#include <linux/module.h>
#include <linux/errno.h>
#include <linux/sched.h>
#include <linux/mm.h>
#include <linux/fs.h>
#include <linux/kernel.h>
#include <linux/genhd.h>
#include <linux/xd.h>

#include <asm/system.h>
#include <asm/io.h>
#include <asm/segment.h>
#include <asm/dma.h>

#define MAJOR_NR XT_DISK_MAJOR
#include <linux/blk.h>

XD_INFO xd_info[XD_MAXDRIVES];

/*
 * If you try this driver and find that your card is not detected by
 * the driver at bootup, you need to add your BIOS
 * signature and details to the following list of signatures. A BIOS
 * signature is a string embedded into the first few bytes of your
 * controller's on-board ROM BIOS. To find out what yours is, use
 * something like MS-DOS's DEBUG command. Run DEBUG, and then you can
 * examine your BIOS signature with:
 *
 *      d xxxx:0000
 *
 * where xxxx is the segment of your controller (like C800 or D000 or
 * something). On the ASCII dump at the right, you should be able to
 * see a string mentioning the manufacturer's copyright etc. Add this
 * string into the table below. The parameters in the table are, in
 * order:
```

```
 *
 * offset                ; this is the offset (in bytes) from the
 *                       ; start of your ROM where the signature starts
 * signature             ; this is the actual text of the signature
 * xd_?_init_controller ; this is the controller init routine
 *                       ; used by your controller
 * xd_?_init_drive       ; this is the drive init routine used
 *                       ; by your controller
 *
 * The controllers directly supported at the moment are: DTC 5150x, WD
 * 1004A27X, ST11M/R and override. If your controller is made by the
 * same manufacturer as one of these, try using the same init routines
 * as they do. If that doesn't work, your best bet is to use the
 * "override" routines. These routines use a "portable" method of
 * getting the disk's geometry, and may work with your card. If none
 * of these seem to work, try sending me some email and I'll see what
 * I can do <grin>.
 *
 * NOTE: You can now specify your XT controller's parameters from the
 * command line in the form xd=TYPE,IRQ,IO,DMA. The driver should be
 * able to detect your drive's geometry from this info. (eg:
 * xd=0,5,0x320,3 is the "standard").
 */

static XD_SIGNATURE xd_sigs[] = {
        /* Pat Mackinlay, pat@it.com.au */
        { 0x0000, "Override geometry handler", NULL,
                  xd_override_init_drive, "n unknown" },
        /* Ian Justman, citrus!ianj@csusac.ecs.csus.edu */
        { 0x0008, "07/15/86 (C) Copyright 1986 Western Digital Corp",
                  xd_wd_init_controller, xd_wd_init_drive,
                  " Western Digital 1002AWX1" },
        /* Dave Thaler, thalerd@engin.umich.edu */
        { 0x0008,"06/24/88 (C) Copyright 1988 Western Digital Corp",
                  xd_wd_init_controller,xd_wd_init_drive,
                  " Western Digital 1004A27X" },
};

static u_char *xd_bases[] =
{
        (u_char *) 0xC8000,(u_char *) 0xCA000,(u_char *) 0xCC000,
        (u_char *) 0xCE000,(u_char *) 0xD0000,(u_char *) 0xD8000,
        (u_char *) 0xE0000
};

static struct hd_struct xd[XD_MAXDRIVES << 6];
static int xd_sizes[XD_MAXDRIVES << 6];
static int xd_access[XD_MAXDRIVES] = { 0, 0 };
static int xd_blocksizes[XD_MAXDRIVES << 6];
static struct gendisk xd_gendisk = {
        MAJOR_NR,       /* Major number */
        "xd",           /* Major name */
        6,              /* Bits to shift to get real from partition */
        1 << 6,         /* Number of partitions per real */
        XD_MAXDRIVES,   /* maximum number of real */
#ifdef MODULE
        NULL,           /* called from init_module */
#else
        xd_geninit,     /* init function */
#endif
```

```
        xd,             /* hd struct */
        xd_sizes,       /* block sizes */
        0,              /* number */
        (void *) xd_info, /* internal */
        NULL            /* next */
};
static struct file_operations xd_fops = {
        NULL,                   /* lseek - default */
        block_read,             /* read - general block-dev read */
        block_write,            /* write - general block-dev write */
        NULL,                   /* readdir - bad */
        NULL,                   /* select */
        xd_ioctl,               /* ioctl */
        NULL,                   /* mmap */
        xd_open,                /* open */
        xd_release,             /* release */
        block_fsync             /* fsync */
};
static struct wait_queue *xd_wait_int = NULL, *xd_wait_open = NULL;
static u_char xd_valid[XD_MAXDRIVES] = { 0,0 };
static u_char xd_drives = 0, xd_irq = 0, xd_dma = 0, xd_maxsectors;
static u_char xd_override = 0, xd_type = 0;
static u_short xd_iobase = 0;

/*
 * xd_init: register the block device number and set up pointer tables
 */
int xd_init (void)
{
        if (register_blkdev(MAJOR_NR,"xd",&xd_fops)) {
                printk("xd_init: unable to get major number %d\n",MAJOR_NR);
                return -1;
        }
        blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;
        read_ahead[MAJOR_NR] = 8;       /* 8 sector (4kB) read ahead */
        xd_gendisk.next = gendisk_head;
        gendisk_head = &xd_gendisk;

        return 0;
}

/*
 * xd_detect: scan the possible BIOS ROM locations for the signature
 * strings
 */
static u_char xd_detect (u_char *controller,u_char **address)
{
        u_char i,j,found = 0;

        if (xd_override)
        {
                *controller = xd_type;
                *address = NULL;
                return(1);
        }

        for (i = 0;
             i < (sizeof(xd_bases) / sizeof(xd_bases[0])) && !found;
             i++) {
                for (j = 1;
```

```
                    j < (sizeof(xd_sigs) / sizeof(xd_sigs[0])) && !found;
                    j++) {
                        if (!memcmp(xd_bases[i] + xd_sigs[j].offset,
                                    xd_sigs[j].string,
                                    strlen(xd_sigs[j].string))) {
                                *controller = j;
                                *address = xd_bases[i];
                                found++;
                        }
                }
        }
        return (found);
}

/*
 * xd_geninit: grab the IRQ and DMA channel, initialise the drives and
 * set up the "raw" device entries in the table
 */
static void xd_geninit (struct gendisk *ignored)
{
        u_char i,controller,*address;

        if (xd_detect(&controller,&address)) {

                printk("xd_geninit: detected a%s controller (type %d) "
                       "at address %p\n", xd_sigs[controller].name,
                       controller,address);
                if (controller)
                        xd_sigs[controller].init_controller(address);
                xd_drives = xd_initdrives(xd_sigs[controller].init_drive);

                printk("xd_geninit: detected %d hard drive%s "
                       "(using IRQ%d & DMA%d)\n", xd_drives,
                       xd_drives == 1 ? "" : "s",xd_irq,xd_dma);
                for (i = 0; i < xd_drives; i++)
                        printk("xd_geninit: drive %d geometry - "
                               "heads = %d, cylinders = %d, sectors = %d\n",
                               i, xd_info[i].heads,
                               xd_info[i].cylinders, xd_info[i].sectors);

                if (!request_irq(xd_irq, xd_interrupt_handler,
                                 0, "XT harddisk", NULL)) {
                        if (request_dma(xd_dma,"xd")) {
                                printk("xd_geninit: unable to get DMA%d\n",
                                       xd_dma);
                                free_irq(xd_irq, NULL);
                        }
                }
                else
                        printk("xd_geninit: unable to get IRQ%d\n",xd_irq);
        }

        for (i = 0; i < xd_drives; i++) {
                xd[i << 6].nr_sects = xd_info[i].heads *
                        xd_info[i].cylinders * xd_info[i].sectors;
                xd_valid[i] = 1;
        }

        xd_gendisk.nr_real = xd_drives;

```

```
        for(i=0;i<(XD_MAXDRIVES << 6);i++) xd_blocksizes[i] = 1024;
        blksize_size[MAJOR_NR] = xd_blocksizes;
}

/* xd_open: open a device */
static int xd_open (struct inode *inode,struct file *file)
{
        int dev = DEVICE_NR(inode->i_rdev);

        if (dev < xd_drives) {
                while (!xd_valid[dev])
                        sleep_on(&xd_wait_open);

                xd_access[dev]++;

                return (0);
        }
        else
                return -ENXIO;
}

/* do_xd_request: handle an incoming request */
static void do_xd_request (void)
{
        u_int block,count,retry;
        int code;

        sti();
        while (code = 0, CURRENT) {
                /* do some checking on the request structure */
                INIT_REQUEST;

                if (CURRENT_DEV < xd_drives
                    && CURRENT->sector + CURRENT->nr_sectors
                         <= xd[MINOR(CURRENT->rq_dev)].nr_sects) {
                        block = CURRENT->sector +
                                xd[MINOR(CURRENT->rq_dev)].start_sect;
                        count = CURRENT->nr_sectors;

                        switch (CURRENT->cmd) {
                        case READ:
                        case WRITE:
                                for (retry = 0;
                                     (retry < XD_RETRIES) && !code;
                                     retry++)
                                        code = xd_readwrite(CURRENT->cmd,
                                                            CURRENT_DEV,
                                                            CURRENT->buffer,
                                                            block,count);
                                break;
                        default:
                                printk("do_xd_request: unknown request\n");
                                break;
                        }
                }
                end_request(code);        /* wrap up, 0 = fail, 1 = success
*/
        }
}
```

```
/* xd_ioctl: handle device ioctl's */
static int xd_ioctl (struct inode *inode, struct file *file,
                     u_int cmd, u_long arg)
{
        XD_GEOMETRY *geometry = (XD_GEOMETRY *) arg;
        int dev = DEVICE_NR(inode->i_rdev),err;

        if (inode && (dev < xd_drives))
                switch (cmd) {
                case HDIO_GETGEO:
                        if (arg) {
                                if ((err = verify_area(VERIFY_WRITE,
                                                       geometry,
                                                       sizeof(*geometry))))
                                        return (err);
                                put_user(xd_info[dev].heads,
                                         &geometry->heads);
                                put_user(xd_info[dev].sectors,
                                         &geometry->sectors);
                                put_user(xd_info[dev].cylinders,
                                         &geometry->cylinders);
                                put_user(xd[MINOR(inode->i_rdev)].start_sect,
                                         &geometry->start);

                                return (0);
                        }
                        break;
                case BLKRASET:
                        if(!suser())
                                return -EACCES;
                        if(!(inode->i_rdev))
                                return -EINVAL;
                        if(arg > 0xff)
                                return -EINVAL;
                        read_ahead[MAJOR(inode->i_rdev)] = arg;
                        return 0;
                case BLKGETSIZE:
                        if (arg) {
                                if ((err = verify_area(VERIFY_WRITE,
                                                       (long *) arg,
                                                       sizeof(long))))
                                        return (err);
                                put_user(xd[MINOR(inode->i_rdev)].nr_sects,
                                         (long *) arg);

                                return (0);
                        }
                        break;
                case BLKFLSBUF:
                        if(!suser())  return -EACCES;
                        if(!(inode->i_rdev))
                                return -EINVAL;
                        fsync_dev(inode->i_rdev);
                        invalidate_buffers(inode->i_rdev);
                        return 0;

                case BLKRRPART:
                        return (xd_reread_partitions(inode->i_rdev));
                        RO_IOCTLS(inode->i_rdev,arg);
                }
```

```
		return (-EINVAL);
}

/* xd_release: release the device */
static void xd_release (struct inode *inode, struct file *file)
{
	int dev = DEVICE_NR(inode->i_rdev);

	if (dev < xd_drives) {
		sync_dev(inode->i_rdev);
		xd_access[dev]--;
	}
}

/* xd_reread_partitions: rereads the partition table from a drive */
static int xd_reread_partitions(kdev_t dev)
{
	int target = DEVICE_NR(dev);
	int start = target << xd_gendisk.minor_shift;
	int partition;

	cli(); xd_valid[target] = (xd_access[target] != 1); sti();
	if (xd_valid[target])
		return (-EBUSY);

	for (partition = xd_gendisk.max_p - 1; partition >= 0; partition--) {
		int minor = (start | partition);
		kdev_t devp = MKDEV(MAJOR_NR, minor);
		sync_dev(devp);
		invalidate_inodes(devp);
		invalidate_buffers(devp);
		xd_gendisk.part[minor].start_sect = 0;
		xd_gendisk.part[minor].nr_sects = 0;
	};

	xd_gendisk.part[start].nr_sects = xd_info[target].heads *
		xd_info[target].cylinders * xd_info[target].sectors;
	resetup_one_dev(&xd_gendisk,target);

	xd_valid[target] = 1;
	wake_up(&xd_wait_open);

	return (0);
}

/* xd_readwrite: handle a read/write request */
static int xd_readwrite (u_char operation, u_char drive,
                         char *buffer, u_int block, u_int count)
{
	u_char cmdblk[6],sense[4];
	u_short track,cylinder;
	u_char head,sector,control,mode,temp;

#ifdef DEBUG_READWRITE
	printk("xd_readwrite: operation = %s, drive = %d, "
	       "buffer = 0x%X, block = %d, count = %d\n",
	       operation == READ ? "read" : "write", drive, buffer,
	       block, count);
#endif /* DEBUG_READWRITE */
```

```
	control = xd_info[drive].control;
	while (count) {
		temp = count < xd_maxsectors ? count : xd_maxsectors;

		track = block / xd_info[drive].sectors;
		head = track % xd_info[drive].heads;
		cylinder = track / xd_info[drive].heads;
		sector = block % xd_info[drive].sectors;

#ifdef DEBUG_READWRITE
		printk("xd_readwrite: drive = %d, head = %d, "
		       "cylinder = %d, sector = %d, count = %d\n",
		       drive, head, cylinder, sector, temp);
#endif /* DEBUG_READWRITE */

		mode = xd_setup_dma(operation == READ ? DMA_MODE_READ :
				    DMA_MODE_WRITE,
				    (u_char *)buffer,temp * 0x200);
		xd_build(cmdblk,operation == READ ? CMD_READ :
			 CMD_WRITE, drive, head, cylinder, sector,
			 temp & 0xFF,control);

		switch (xd_command(cmdblk, mode, (u_char *) buffer,
				   (u_char *) buffer, sense, XD_TIMEOUT)) {
		case 1:
			printk("xd_readwrite: timeout, "
			       "recalibrating drive\n");
			xd_recalibrate(drive);
			return (0);
		case 2:
			switch ((sense[0] & 0x30) >> 4) {
			case 0: printk("xd_readwrite: drive error, "
				       "code = 0x%X", sense[0] & 0x0F);
				break;
			case 1: printk("xd_readwrite: controller error, "
				       "code = 0x%X", sense[0] & 0x0F);
				break;
			case 2: printk("xd_readwrite: command error, "
				       "code = 0x%X", sense[0] & 0x0F);
				break;
			case 3: printk("xd_readwrite: miscellaneous "
				       "error, code = 0x%X",
				       sense[0] & 0x0F);
				break;
			}
			if (sense[0] & 0x80)
				printk(" - drive = %d, head = %d, "
				       "cylinder = %d, sector = %d\n",
				       sense[1] & 0xE0, sense[1] & 0x1F,
				       ((sense[2] & 0xC0) << 2) |
				       sense[3],sense[2] & 0x3F);
			else
				printk(" - no valid disk address\n");
			return (0);
		}
		count -= temp, buffer += temp * 0x200, block += temp;
	}
	return (1);
}
```

```
/*
 * xd_recalibrate: recalibrate a given drive and reset controller if
 * necessary
 */
static void xd_recalibrate (u_char drive)
{
        u_char cmdblk[6];

        xd_build(cmdblk,CMD_RECALIBRATE,drive,0,0,0,0,0);
        if (xd_command(cmdblk,PIO_MODE,0,0,0,XD_TIMEOUT * 8))
                printk("xd_recalibrate: warning! error "
                       "recalibrating, controller may be unstable\n");
}

/* xd_interrupt_handler: interrupt service routine */
static void xd_interrupt_handler(int irq, void *dev_id,
                                 struct pt_regs * regs)
{
        /* check if it was our device */
        if (inb(XD_STATUS) & STAT_INTERRUPT) {
#ifdef DEBUG_OTHER
                printk("xd_interrupt_handler: interrupt detected\n");
#endif /* DEBUG_OTHER */
                /* acknowledge interrupt */
                outb(0,XD_CONTROL);
                /* and wake up sleeping processes */
                wake_up(&xd_wait_int);
        } else
                printk("xd_interrupt_handler: unexpected interrupt\n");
}

/* xd_dma: set up the DMA controller for a data transfer */
static u_char xd_setup_dma (u_char mode,u_char *buffer,u_int count)
{
        /* transfer to address < 16M? */
        if (buffer < ((u_char *) 0x1000000 - count)) {
                if (((u_int) buffer & 0xFFFF0000) !=
                    (((u_int) buffer + count) & 0xFFFF0000)) {
#ifdef DEBUG_OTHER
                        printk("xd_setup_dma: using PIO, "
                               "transfer overlaps 64k boundary\n");
#endif /* DEBUG_OTHER */
                        return (PIO_MODE);
                }
                disable_dma(xd_dma);
                clear_dma_ff(xd_dma);
                set_dma_mode(xd_dma,mode);
                set_dma_addr(xd_dma,(u_int) buffer);
                set_dma_count(xd_dma,count);

                return (DMA_MODE);                        /* use DMA and INT
*/
        }
#ifdef DEBUG_OTHER
        printk("xd_setup_dma: using PIO, cannot DMA above 16 meg\n");
#endif /* DEBUG_OTHER */
        return (PIO_MODE);
}

/*
```

```
 * xd_build: put stuff into an array in a format suitable for the
 * controller
 */
static u_char *xd_build (u_char *cmdblk, u_char command, u_char drive,
                         u_char head, u_short cylinder, u_char sector,
                         u_char count, u_char control)
{
        cmdblk[0] = command;
        cmdblk[1] = ((drive & 0x07) << 5) | (head & 0x1F);
        cmdblk[2] = ((cylinder & 0x300) >> 2) | (sector & 0x3F);
        cmdblk[3] = cylinder & 0xFF;
        cmdblk[4] = count;
        cmdblk[5] = control;

        return (cmdblk);
}

/*
 * xd_waitport: waits until port & mask == flags or a timeout
 * occurs. return 1 for a timeout
 */
static inline u_char xd_waitport (u_short port, u_char flags,
                                  u_char mask, u_long timeout)
{
        u_long expiry = jiffies + timeout;

        while (((inb(port) & mask) != flags) && (jiffies < expiry))
                ;

        return (jiffies >= expiry);
}

/*
 * xd_command: handle all data transfers necessary for a single
 * command
 */
static u_int xd_command (u_char *command, u_char mode, u_char *indata,
                         u_char *outdata, u_char *sense, u_long timeout)
{
        u_char cmdblk[6],csb,complete = 0;

#ifdef DEBUG_COMMAND
        printk("xd_command: command = 0x%X, mode = 0x%X, "
               "indata = 0x%X, outdata = 0x%X, sense = 0x%X\n",
               command, mode, indata, outdata, sense);
#endif /* DEBUG_COMMAND */

        outb(0,XD_SELECT);
        outb(mode,XD_CONTROL);

        if (xd_waitport(XD_STATUS,STAT_SELECT,STAT_SELECT,timeout))
                return (1);

        while (!complete) {
                if (xd_waitport(XD_STATUS,STAT_READY,STAT_READY,timeout))
                        return (1);
                switch (inb(XD_STATUS) & (STAT_COMMAND | STAT_INPUT)) {
                case 0:
                        if (mode == DMA_MODE) {
                                enable_dma(xd_dma);
```

```
                                sleep_on(&xd_wait_int);
                                disable_dma(xd_dma);
                        } else
                                outb(outdata ? *outdata++ : 0,XD_DATA);
                        break;
                case STAT_INPUT:
                        if (mode == DMA_MODE) {
                                enable_dma(xd_dma);
                                sleep_on(&xd_wait_int);
                                disable_dma(xd_dma);
                        } else
                                if (indata)
                                        *indata++ = inb(XD_DATA);
                                else
                                        inb(XD_DATA);
                        break;
                case STAT_COMMAND:
                        outb(command ? *command++ : 0,XD_DATA);
                        break;
                        case STAT_COMMAND | STAT_INPUT:
                        complete = 1;
                        break;
                }
        }
        csb = inb(XD_DATA);

        /* wait until deselected */
        if (xd_waitport(XD_STATUS,0,STAT_SELECT,timeout))
                return (1);

        /* read sense data if error */
        if (csb & CSB_ERROR) {
                xd_build(cmdblk,CMD_SENSE,(csb & CSB_LUN) >> 5,0,0,0,0,0);
                if (xd_command(cmdblk,0,sense,0,0,XD_TIMEOUT))
                        printk("xd_command: warning! "
                               "sense command failed!\n");
        }

#ifdef DEBUG_COMMAND
        printk("xd_command: completed with csb = 0x%X\n",csb);
#endif /* DEBUG_COMMAND */

        return (csb & CSB_ERROR);
}

static u_char xd_initdrives (void (*init_drive)(u_char drive))
{
        u_char cmdblk[6],i,count = 0;

        for (i = 0; i < XD_MAXDRIVES; i++) {
                xd_build(cmdblk,CMD_TESTREADY,i,0,0,0,0,0);
                if (!xd_command(cmdblk,PIO_MODE,0,0,0,XD_TIMEOUT * 8)) {
                        init_drive(count);
                        count++;
                }
        }
        return (count);
}

static void xd_wd_init_controller (u_char *address)
```

```
{
        switch ((u_long) address) {
        case 0xC8000:   xd_iobase = 0x320; break;
        case 0xCA000:   xd_iobase = 0x324; break;
        case 0xCC000:   xd_iobase = 0x328; break;
        case 0xCE000:   xd_iobase = 0x32C; break;
        case 0xD0000:   xd_iobase = 0x328; break;
        case 0xD8000:   xd_iobase = 0x32C; break;
        default:        printk("xd_wd_init_controller: "
                               "unsupported BIOS address %p\n",address);
                xd_iobase = 0x320; break;
        }
        xd_irq = 5;             /* don't know how to auto-detect this yet */
        xd_dma = 3;
        xd_maxsectors = 0x01;   /* this one doesn't wrap properly either...
*/

        /* outb(0,XD_RESET); */   /* reset the controller */
}

static void xd_wd_init_drive (u_char drive)
{
        u_char cmdblk[6],buf[0x200];

        xd_build(cmdblk,CMD_READ,drive,0,0,0,1,0);
        if (!xd_command(cmdblk,PIO_MODE,buf,0,0,XD_TIMEOUT * 2)) {
                /* heads */
                xd_info[drive].heads = buf[0x1AF];
                /* cylinders */
                xd_info[drive].cylinders = ((u_short *) (buf + 1))[0xD6];
                /* sectors */
                xd_info[drive].sectors = 17;
#if 0
                /* reduced write */
                xd_info[drive].rwrite = ((u_short *) (buf))[0xD8];
                /* write precomp */
                xd_info[drive].wprecomp = ((u_short *) (buf))[0xDA];
                /* ecc length */
                xd_info[drive].ecc = buf[0x1B4];
#endif /* 0 */
                /* control byte */
                xd_info[drive].control = buf[0x1B5];

                xd_setparam(CMD_WDSETPARAM,drive,
                            xd_info[drive].heads,
                            xd_info[drive].cylinders,
                            ((u_short *) (buf))[0xD8],
                            ((u_short *) (buf))[0xDA], buf[0x1B4]);
        }
        else
                printk("xd_wd_init_drive: error reading "
                       "geometry for drive %d\n",drive);
}


/*
 * xd_override_init_drive: this finds disk geometry in a "binary
 * search" style, narrowing in on the "correct" number of heads
 * etc. by trying values until it gets the highest successful
 * value. Idea courtesy Salvador Abreu (spa@fct.unl.pt).
```

```
 */
static void xd_override_init_drive (u_char drive)
{
        u_short min[] = { 0,0,0 },max[] = { 16,1024,64 },test[] = { 0,0,0 };
        u_char cmdblk[6],i;

        for (i = 0; i < 3; i++) {
                while (min[i] != max[i] - 1) {
                        test[i] = (min[i] + max[i]) / 2;
                        xd_build(cmdblk, CMD_SEEK, drive,
                                 (u_char) test[0], (u_short) test[1],
                                 (u_char) test[2], 0, 0);
                        if (!xd_command(cmdblk, PIO_MODE, 0, 0, 0,
                                        XD_TIMEOUT * 2))
                                min[i] = test[i];
                        else
                                max[i] = test[i];
                }
                test[i] = min[i];
        }
        xd_info[drive].heads = (u_char) min[0] + 1;
        xd_info[drive].cylinders = (u_short) min[1] + 1;
        xd_info[drive].sectors = (u_char) min[2] + 1;
        xd_info[drive].control = 0;
}

/* xd_setup: initialise from command line parameters */
void xd_setup (char *command,int *integers)
{
        xd_override = 1;

        xd_type = integers[1];
        xd_irq = integers[2];
        xd_iobase = integers[3];
        xd_dma = integers[4];

        xd_maxsectors = 0x01;
}

/* xd_setparam: set the drive characteristics */
static void xd_setparam (u_char command, u_char drive, u_char heads,
                         u_short cylinders, u_short rwrite,
                         u_short wprecomp, u_char ecc)
{
        u_char cmdblk[14];

        xd_build(cmdblk,command,drive,0,0,0,0,0);
        cmdblk[6] = (u_char) (cylinders >> 8) & 0x03;
        cmdblk[7] = (u_char) (cylinders & 0xFF);
        cmdblk[8] = heads & 0x1F;
        cmdblk[9] = (u_char) (rwrite >> 8) & 0x03;
        cmdblk[10] = (u_char) (rwrite & 0xFF);
        cmdblk[11] = (u_char) (wprecomp >> 8) & 0x03;
        cmdblk[12] = (u_char) (wprecomp & 0xFF);
        cmdblk[13] = ecc;

        if (xd_command(cmdblk,PIO_MODE,0,0,0,XD_TIMEOUT * 2))
                printk("xd_setparam: error setting "
                       "characteristics for drive %d\n",drive);
}
```

```
#ifdef MODULE
int init_module(void)
{
        int error = xd_init();
        if (!error)
        {
                printk(KERN_INFO "XD: Loaded as a module.\n");
                xd_geninit(&(struct gendisk) { 0,0,0,0,0,0,0,0,0,0,0 });
        }

        return error;
}

void cleanup_module(void)
{
        unregister_blkdev(MAJOR_NR, "xd");
        free_irq(xd_irq, NULL);
        free_dma(xd_dma);
}
#endif /* MODULE */
```