```
/*
 *  scsi.h Copyright (C) 1992 Drew Eckhardt
 *         Copyright (C) 1993, 1994, 1995 Eric Youngdale
 *  generic SCSI package header file by
 *      Initial versions: Drew Eckhardt
 *      Subsequent revisions: Eric Youngdale
 *
 *  <drew@colorado.edu>
 *
 *       Modified by Eric Youngdale eric@aib.com to
 *       add scatter-gather, multiple outstanding request, and other
 *       enhancements.
 */

#ifndef _SCSI_H
#define _SCSI_H

/*
 * Some of the public constants are being moved to this file.
 * We include it here so that what came from where is transparent.
 */
#include <scsi/scsi.h>

#include <linux/random.h>


/*
 * Some defs, in case these are not defined elsewhere.
 */
#ifndef TRUE
# define TRUE 1
#endif
#ifndef FALSE
# define FALSE 0
#endif


extern void scsi_make_blocked_list(void);
extern volatile int in_scan_scsis;
extern const unsigned char scsi_command_size[8];
#define COMMAND_SIZE(opcode) scsi_command_size[((opcode) >> 5) & 7]

#define IDENTIFY_BASE       0x80
#define IDENTIFY(can_disconnect, lun)   (IDENTIFY_BASE |\
                     ((can_disconnect) ?  0x40 : 0) |\
                     ((lun) & 0x07))



/*
 *  the return of the status word will be in the following format :
 *  The low byte is the status returned by the SCSI command,
 *  with vendor specific bits masked.
 *
 *  The next byte is the message which followed the SCSI status.
 *  This allows a stos to be used, since the Intel is a little
 *  endian machine.
 *
 *  The final byte is a host return code, which is one of the following.
 *
```

```
 *  IE
 *  lsb     msb
 *  status  msg host code
 *
 *  Our errors returned by OUR driver, NOT SCSI message.  Or'd with
 *  SCSI message passed back to driver <IF any>.
 */


#define DID_OK          0x00 /* NO error                                */
#define DID_NO_CONNECT  0x01 /* Couldn't connect before timeout period  */
#define DID_BUS_BUSY    0x02 /* BUS stayed busy through time out period */
#define DID_TIME_OUT    0x03 /* TIMED OUT for other reason              */
#define DID_BAD_TARGET  0x04 /* BAD target.                             */
#define DID_ABORT       0x05 /* Told to abort for some other reason     */
#define DID_PARITY      0x06 /* Parity error                            */
#define DID_ERROR       0x07 /* Internal error                          */
#define DID_RESET       0x08 /* Reset by somebody.                      */
#define DID_BAD_INTR    0x09 /* Got an interrupt we weren't expecting.  */
#define DRIVER_OK       0x00 /* Driver status                           */

/*
 *  These indicate the error that occurred, and what is available.
 */

#define DRIVER_BUSY         0x01
#define DRIVER_SOFT         0x02
#define DRIVER_MEDIA        0x03
#define DRIVER_ERROR        0x04

#define DRIVER_INVALID      0x05
#define DRIVER_TIMEOUT      0x06
#define DRIVER_HARD         0x07
#define DRIVER_SENSE        0x08

#define SUGGEST_RETRY       0x10
#define SUGGEST_ABORT       0x20
#define SUGGEST_REMAP       0x30
#define SUGGEST_DIE         0x40
#define SUGGEST_SENSE       0x80
#define SUGGEST_IS_OK       0xff

#define DRIVER_MASK         0x0f
#define SUGGEST_MASK        0xf0

#define MAX_COMMAND_SIZE    12

/*
 *  SCSI command sets
 */

#define SCSI_UNKNOWN    0
#define SCSI_1          1
#define SCSI_1_CCS      2
#define SCSI_2          3

/*
 *  Every SCSI command starts with a one byte OP-code.
 *  The next byte's high three bits are the LUN of the
 *  device.  Any multi-byte quantities are stored high byte
```

```
 *  first, and may have a 5 bit MSB in the same byte
 *  as the LUN.
 */

/*
 *      Manufacturers list
 */

#define SCSI_MAN_UNKNOWN     0
#define SCSI_MAN_NEC         1
#define SCSI_MAN_TOSHIBA     2
#define SCSI_MAN_NEC_OLDCDR  3
#define SCSI_MAN_SONY        4
#define SCSI_MAN_PIONEER     5

/*
 *  As the scsi do command functions are intelligent, and may need to
 *  redo a command, we need to keep track of the last command
 *  executed on each one.
 */

#define WAS_RESET       0x01
#define WAS_TIMEDOUT    0x02
#define WAS_SENSE       0x04
#define IS_RESETTING    0x08
#define IS_ABORTING     0x10
#define ASKED_FOR_SENSE 0x20

/*
 *  The scsi_device struct contains what we know about each given scsi
 *  device.
 */

typedef struct scsi_device {
    struct scsi_device * next;      /* Used for linked list */

    unsigned char id, lun, channel;

    unsigned int manufacturer;      /* Manufacturer of device, for using
                                     * vendor-specific cmd's */
    int attached;                   /* # of high level drivers attached to
                                     * this */
    int access_count;               /* Count of open channels/mounts */
    struct wait_queue * device_wait;/* Used to wait if device is busy */
    struct Scsi_Host * host;
    void (*scsi_request_fn)(void);  /* Used to jumpstart things after an
                                     * ioctl */
    struct scsi_cmnd *device_queue; /* queue of SCSI Command structures */
    void *hostdata;                 /* available to low-level driver */
    char type;
    char scsi_level;
    char vendor[8], model[16], rev[4];
    unsigned char current_tag;      /* current tag */
    unsigned char sync_min_period;  /* Not less than this period */
    unsigned char sync_max_offset;  /* Not greater than this offset */
    unsigned char queue_depth;      /* How deep a queue to use */

    unsigned writeable:1;
    unsigned removable:1;
    unsigned random:1;
```

```
    unsigned has_cmdblocks:1;
    unsigned changed:1;          /* Data invalid due to media change */
    unsigned busy:1;             /* Used to prevent races */
    unsigned lockable:1;         /* Able to prevent media removal */
    unsigned borken:1;           /* Tell the Seagate driver to be
                                  * painfully slow on this device */
    unsigned tagged_supported:1; /* Supports SCSI-II tagged queuing */
    unsigned tagged_queue:1;     /* SCSI-II tagged queuing enabled */
    unsigned disconnect:1;       /* can disconnect */
    unsigned soft_reset:1;       /* Uses soft reset option */
    unsigned sync:1;             /* Negotiate for sync transfers */
    unsigned single_lun:1;       /* Indicates we should only allow I/O
                                  * to one of the luns for the device
                                  * at a time. */
    unsigned was_reset:1;        /* There was a bus reset on the bus
                                  * for this device */
    unsigned expecting_cc_ua:1;  /* Expecting a
                                  * CHECK_CONDITION/UNIT_ATTN
                                  * because we did a bus reset. */
} Scsi_Device;

/*
 *  Use these to separate status msg and our bytes
 */

#define status_byte(result) (((result) >> 1) & 0x1f)
#define msg_byte(result)    (((result) >> 8) & 0xff)
#define host_byte(result)   (((result) >> 16) & 0xff)
#define driver_byte(result) (((result) >> 24) & 0xff)
#define suggestion(result)  (driver_byte(result) & SUGGEST_MASK)

#define sense_class(sense)  (((sense) >> 4) & 0x7)
#define sense_error(sense)  ((sense) & 0xf)
#define sense_valid(sense)  ((sense) & 0x80);

/*
 *  These are the SCSI devices available on the system.
 */

extern Scsi_Device * scsi_devices;

/*
 *  Initializes all SCSI devices.  This scans all scsi busses.
 */

extern int scsi_dev_init (void);

struct scatterlist {
    char *  address;    /* Location data is to be transferred to */
    char * alt_address; /* Location of actual if address is a
                         * dma indirect buffer.  NULL otherwise */
    unsigned int length;
};

#ifdef __alpha__
# define ISA_DMA_THRESHOLD (~0UL)
#else
# define ISA_DMA_THRESHOLD (0x00ffffff)
#endif
#define CONTIGUOUS_BUFFERS(X,Y) ((X->b_data+X->b_size) == Y->b_data)
```

```
/*
 * These are the return codes for the abort and reset functions.  The
 * mid-level code uses these to decide what to do next.  Each of the
 * low level abort and reset functions must correctly indicate what it
 * has done.  The descriptions are written from the point of view of
 * the mid-level code, so that the return code is telling the
 * mid-level drivers exactly what the low level driver has already
 * done, and what remains to be done.
 */

/* We did not do anything.
 * Wait some more for this command to complete, and if this does not
 * work, try something more serious.
 */
#define SCSI_ABORT_SNOOZE 0

/* This means that we were able to abort the command.  We have already
 * called the mid-level done function, and do not expect an interrupt
 * that will lead to another call to the mid-level done function for
 * this command
 */
#define SCSI_ABORT_SUCCESS 1

/*
 * We called for an abort of this command, and we should get an
 * interrupt when this succeeds.  Thus we should not restore the timer
 * for this command in the mid-level abort function.
 */
#define SCSI_ABORT_PENDING 2

/*
 * Unable to abort - command is currently on the bus.  Grin and bear
 * it.
 */
#define SCSI_ABORT_BUSY 3

/* The command is not active in the low level code. Command probably
 * finished. */
#define SCSI_ABORT_NOT_RUNNING 4

/* Something went wrong.  The low level driver will indicate the correct
 * error condition when it calls scsi_done, so the mid-level abort
 * function can simply wait until this comes through
 */
#define SCSI_ABORT_ERROR 5

/* We do not know how to reset the bus, or we do not want to.  Bummer.
 * Anyway, just wait a little more for the command in question, and
 * hope that it eventually finishes.  If it never finishes, the SCSI
 * device could hang, so use this with caution.
 */
#define SCSI_RESET_SNOOZE 0

/* We do not know how to reset the bus, or we do not want to.  Bummer.
 * We have given up on this ever completing.  The mid-level code will
 * request sense information to decide how to proceed from here.
 */
#define SCSI_RESET_PUNT 1
```

```
/*
 * This means that we were able to reset the bus.  We have restarted
 * all of the commands that should be restarted, and we should be able
 * to continue on normally from here.  We do not expect any interrupts
 * that will return DID_RESET to any of the other commands in the
 * host_queue, and the mid-level code does not need to do anything
 * special to keep the commands alive.  If a hard reset was performed
 * then all outstanding commands on the bus have been restarted.
 */
#define SCSI_RESET_SUCCESS 2

/* We called for a reset of this bus, and we should get an interrupt
 * when this succeeds.  Each command should get its own status passed
 * up to scsi_done, but this has not happened yet.  If a hard reset
 * was performed, then we expect an interrupt for *each* of the
 * outstanding commands that will have the effect of restarting the
 * commands.
 */
#define SCSI_RESET_PENDING 3

/* We did a reset, but do not expect an interrupt to signal DID_RESET.
 * This tells the upper level code to request the sense info, and this
 * should keep the command alive. */
#define SCSI_RESET_WAKEUP 4

/* The command is not active in the low level code. Command probably
   finished. */
#define SCSI_RESET_NOT_RUNNING 5

/* Something went wrong, and we do not know how to fix it. */
#define SCSI_RESET_ERROR 6

#define SCSI_RESET_SYNCHRONOUS          0x01
#define SCSI_RESET_ASYNCHRONOUS         0x02
#define SCSI_RESET_SUGGEST_BUS_RESET    0x04
#define SCSI_RESET_SUGGEST_HOST_RESET   0x08
/*
 * This is a bitmask that is ored with one of the above codes.
 * It tells the mid-level code that we did a hard reset.
 */
#define SCSI_RESET_BUS_RESET 0x100
/*
 * This is a bitmask that is ored with one of the above codes.
 * It tells the mid-level code that we did a host adapter reset.
 */
#define SCSI_RESET_HOST_RESET 0x200
/*
 * Used to mask off bits and to obtain the basic action that was
 * performed.
 */
#define SCSI_RESET_ACTION   0xff

void *   scsi_malloc(unsigned int);
int      scsi_free(void *, unsigned int);
extern unsigned int dma_free_sectors;  /* How much room do we have left */
extern unsigned int need_isa_buffer;   /* True if some devices need
                                        * indirection buffers */

/*
```

```
 * The Scsi_Cmnd structure is used by scsi.c internally, and for
 * communication with low level drivers that support multiple
 * outstanding commands.
 */
typedef struct scsi_pointer {
    char * ptr;                     /* data pointer */
    int this_residual;              /* left in this buffer */
    struct scatterlist *buffer;     /* which buffer */
    int buffers_residual;           /* how many buffers left */

    volatile int Status;
    volatile int Message;
    volatile int have_data_in;
    volatile int sent_command;
    volatile int phase;
} Scsi_Pointer;

typedef struct scsi_cmnd {
    struct Scsi_Host * host;
    Scsi_Device * device;
    unsigned char target, lun, channel;
    unsigned char cmd_len;
    unsigned char old_cmd_len;
    struct scsi_cmnd *next, *prev, *device_next, *reset_chain;

    /* These elements define the operation we are about to perform */
    unsigned char cmnd[12];
    unsigned request_bufflen;   /* Actual request size */

    void * request_buffer;      /* Actual requested buffer */

    /* These elements define the operation we ultimately want to perform */
    unsigned char data_cmnd[12];
    unsigned short old_use_sg;  /* We save  use_sg here when requesting
                                 * sense info */
    unsigned short use_sg;      /* Number of pieces of scatter-gather */
    unsigned short sglist_len;  /* size of malloc'd scatter-gather list */
    unsigned short abort_reason;/* If the mid-level code requests an
                                 * abort, this is the reason. */
    unsigned bufflen;           /* Size of data buffer */
    void *buffer;               /* Data buffer */

    unsigned underflow;         /* Return error if less than this amount is
                                 * transfered */

    unsigned transfersize;      /* How much we are guaranteed to transfer
with
                                 * each SCSI transfer (ie, between disconnect
/
                                 * reconnects.   Probably == sector size */


    struct request request;     /* A copy of the command we are working on */

    unsigned char sense_buffer[16];  /* Sense for this command,
                                      * if needed */

    /*
      A SCSI Command is assigned a nonzero serial_number when
      internal_cmnd passes it to the driver's queue command function.
```

```
          The serial_number is cleared when scsi_done is entered
          indicating that the command has been completed.  If a timeout
          occurs, the serial number at the moment of timeout is copied
          into serial_number_at_timeout.  By subsequently comparing the
          serial_number and serial_number_at_timeout fields during abort
          or reset processing, we can detect whether the command has
          already completed.  This also detects cases where the command
          has completed and the SCSI Command structure has already being
          reused for another command, so that we can avoid incorrectly
          aborting or resetting the new command.
        */

        unsigned long serial_number;
        unsigned long serial_number_at_timeout;

        int retries;
        int allowed;
        int timeout_per_command, timeout_total, timeout;

        /*
         *  We handle the timeout differently if it happens when a reset,
         *  abort, etc are in process.
         */
        unsigned volatile char internal_timeout;

        unsigned flags;

        /*
         * These variables are for the cdrom only. Once we have variable
         * size buffers in the buffer cache, they will go away.
         */
        int this_count;
        /* End of special cdrom variables */

        /*
         * Low-level done function - can be used by low-level driver to
         * point to completion function.  Not used by mid/upper level
         * code.
         */
        void (*scsi_done)(struct scsi_cmnd *);
        void (*done)(struct scsi_cmnd *);  /* Mid-level done function */

        /*
         * The following fields can be written to by the host specific code.
         * Everything else should be left alone.
         */

        Scsi_Pointer SCp;   /* Scratchpad used by some host adapters */

        unsigned char * host_scribble; /* The host adapter is allowed to
                                         * call scsi_malloc and get some memory
                                         * and hang it here.  The host adapter
                                         * is also expected to call scsi_free
                                         * to release this memory.  (The memory
                                         * obtained by scsi_malloc is guaranteed
                                         * to be at an address < 16Mb). */

        int result;                    /* Status code from lower level driver */

        unsigned char tag;             /* SCSI-II queued command tag */
```

```
    unsigned long pid;             /* Process ID, starts at 0 */
} Scsi_Cmnd;

/*
 *  scsi_abort aborts the current command that is executing on host
 *  host.  The error code, if non zero is returned in the host byte,
 *  otherwise DID_ABORT is returned in the hostbyte.
 */

extern int scsi_abort (Scsi_Cmnd *, int code);

extern void scsi_do_cmd (Scsi_Cmnd *, const void *cmnd ,
                         void *buffer, unsigned bufflen,
                         void (*done)(struct scsi_cmnd *),
                         int timeout, int retries);


extern Scsi_Cmnd * allocate_device(struct request **, Scsi_Device *, int);

extern Scsi_Cmnd * request_queueable(struct request *, Scsi_Device *);
extern int scsi_reset (Scsi_Cmnd *, unsigned int);

extern int max_scsi_hosts;

extern void proc_print_scsidevice(Scsi_Device *, char *, int *, int);

extern void print_command(unsigned char *);
extern void print_sense(const char *, Scsi_Cmnd *);
extern void print_driverbyte(int scsiresult);
extern void print_hostbyte(int scsiresult);

extern void scsi_mark_host_reset(struct Scsi_Host *Host);
extern void scsi_mark_bus_reset(struct Scsi_Host *Host, int channel);

#if defined(MAJOR_NR) && (MAJOR_NR != SCSI_TAPE_MAJOR)
#include "hosts.h"

static Scsi_Cmnd * end_scsi_request(Scsi_Cmnd * SCpnt, int uptodate,
                                    int sectors)
{
    struct request * req;
    struct buffer_head * bh;

    req = &SCpnt->request;
    req->errors = 0;
    if (!uptodate) {
        printk(DEVICE_NAME " I/O error: dev %s, sector %lu\n",
               kdevname(req->rq_dev), req->sector);
    }

    do {
        if ((bh = req->bh) != NULL) {
            req->bh = bh->b_reqnext;
            req->nr_sectors -= bh->b_size >> 9;
            req->sector += bh->b_size >> 9;
            bh->b_reqnext = NULL;
            mark_buffer_uptodate(bh, uptodate);
            unlock_buffer(bh);
            sectors -= bh->b_size >> 9;
            if ((bh = req->bh) != NULL) {
```

```
                req->current_nr_sectors = bh->b_size >> 9;
                if (req->nr_sectors < req->current_nr_sectors) {
                    req->nr_sectors = req->current_nr_sectors;
                    printk("end_scsi_request: buffer-list destroyed\n");
                }
            }
        }
    } while(sectors && bh);
    if (req->bh){
        req->buffer = bh->b_data;
        return SCpnt;
    }
    DEVICE_OFF(req->rq_dev);
    if (req->sem != NULL) {
        up(req->sem);
    }
    add_blkdev_randomness(MAJOR(req->rq_dev));

    if (SCpnt->host->block) {
        struct Scsi_Host * next;

        for (next = SCpnt->host->block; next != SCpnt->host;
             next = next->block)
            wake_up(&next->host_wait);
    }

    req->rq_status = RQ_INACTIVE;
    wake_up(&wait_for_request);
    wake_up(&SCpnt->device->device_wait);
    return NULL;
}


/* This is just like INIT_REQUEST, but we need to be aware of the fact
 * that an interrupt may start another request, so we run this with
 * interrupts turned off
 */
#define INIT_SCSI_REQUEST       \
    if (!CURRENT) {             \
        CLEAR_INTR;             \
        restore_flags(flags);     \
        return;                 \
    }                           \
    if (MAJOR(CURRENT->rq_dev) != MAJOR_NR)           \
        panic(DEVICE_NAME ": request list destroyed");\
    if (CURRENT->bh) {                                \
        if (!buffer_locked(CURRENT->bh))              \
            panic(DEVICE_NAME ": block not locked");  \
    }
#endif

#define SCSI_SLEEP(QUEUE, CONDITION) {                \
    if (CONDITION) {                                  \
        struct wait_queue wait = { current, NULL};  \
        add_wait_queue(QUEUE, &wait);                 \
        for(;;) {                           \
        current->state = TASK_UNINTERRUPTIBLE;      \
        if (CONDITION) {                    \
            if (intr_count)                           \
                panic("scsi: trying to call schedule() in interrupt" \
```

```
                        ", file %s, line %d.\n", __FILE__, __LINE__);  \
            schedule();                   \
        }                                 \
        else                              \
            break;                        \
        }                                 \
        remove_wait_queue(QUEUE, &wait);\
        current->state = TASK_RUNNING;  \
    }; }

#endif

/*
 * Overrides for Emacs so that we follow Linus's tabbing style.  Emacs
 * will notice this stuff at the end of the file and automatically
 * adjust the settings for this buffer only.  This must remain at the
 * end of the file.
 * ---------------------------------------------------------------------------
 * Local variables:
 * c-indent-level: 4
 * c-brace-imaginary-offset: 0
 * c-brace-offset: -4
 * c-argdecl-indent: 4
 * c-label-offset: -4
 * c-continued-statement-offset: 4
 * c-continued-brace-offset: 0
 * indent-tabs-mode: nil
 * tab-width: 8
 * End:
 */
```