

# [RFC] Host Virtual Serial Interface driver

**Source:** <http://linux.derkeiler.com/Mailing-Lists/Kernel/2004-08/1800.html>

---

**From:** Hollis Blanchard ([hollisb\\_at\\_us.ibm.com](mailto:hollisb_at_us.ibm.com))

**Date:** 08/06/04

To: [linux-kernel@vger.kernel.org](mailto:linux-kernel@vger.kernel.org)

Date: Fri, 06 Aug 2004 16:23:05 -0500

Hi, I have a new char driver I'd like to get comments on. It is specific to IBM's p5 server line; I've included a description from the comments here:

"Host Virtual Serial Interface (HVSI) is a protocol between the hosted OS and the service processor on IBM pSeries servers. On these servers, there are no serial ports under the OS's control, and sometimes there is no other console available either. However, the service processor has two standard serial ports, so this over-complicated protocol allows the OS to control those ports by proxy."

There's at least one question I have regarding TTY throttling. This proctol means that I can receive up to 252 characters all at once (in a single packet). However, TTY\_THRESHOLD\_THROTTLE is 128 bytes, which means conceivably I could overflow the tty buffer with a few very large packets.

To get around that, I've set `tty->low_latency` (to guarantee immediate feedback from `tty_flip_buffer_push()`). I deliver 128 bytes, check to see if I've been throttled, then deliver the rest if possible.

I've included the whole file below; it's pretty much self-contained. All comments welcome.

--

Hollis Blanchard

IBM Linux Technology Center

/\*

\* Copyright (C) 2004 Hollis Blanchard <[hollisb@us.ibm.com](mailto:hollisb@us.ibm.com)>, IBM

\*

\* This program is free software; you can redistribute it and/or modify  
\* it under the terms of the GNU General Public License as published by  
\* the Free Software Foundation; either version 2 of the License, or  
\* (at your option) any later version.

\*

\* This program is distributed in the hope that it will be useful,  
\* but WITHOUT ANY WARRANTY; without even the implied warranty of  
\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the  
\* GNU General Public License for more details.

\*

\* You should have received a copy of the GNU General Public License

## Linux-Kernel: [RFC] Host Virtual Serial Interface driver

```
* along with this program; if not, write to the Free Software
* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
*/

/* Host Virtual Serial Interface (HVSI) is a protocol between the hosted OS
* and the service processor on IBM pSeries servers. On these servers, there
* are no serial ports under the OS's control, and sometimes there is no other
* console available either. However, the service processor has two standard
* serial ports, so this over-complicated protocol allows the OS to control
* those ports by proxy.
*
* Besides data, the protocol supports the reading/writing of the serial
* port's DTR line, and the reading of the CD line. This is to allow the OS to
* control a modem attached to the service processor's serial port. Note that
* the OS cannot change the speed of the port through this protocol.
*/
/* TODO:
* test FSP reset
* add udbg support for xmon/kdb
*/
#undef DEBUG
#include <linux/init.h>
#include <linux/module.h>
#include <linux/console.h>
#include <linux/major.h>
#include <linux/kernel.h>
#include <linux/sysrq.h>
#include <linux/tty.h>
#include <linux/tty_flip.h>
#include <linux/sched.h>
#include <linux/kbd_kern.h>
#include <linux/spinlock.h>
#include <linux/ctype.h>
#include <linux/interrupt.h>
#include <linux/delay.h>
#include <asm/uaccess.h>
#include <asm/hvconsole.h>
#include <asm/prom.h>
#include <asm/hvcall.h>
#include <asm/vio.h>
#define HVSI_MAJOR 229
#define HVSI_MINOR 128
#define MAX_NR_HVSI_CONSOLES 4
#define HVSI_TIMEOUT (5*HZ)
#define HVSI_VERSION 1
#define HVSI_MAX_PACKET 256
#define HVSI_MAX_READ 16
#define HVSI_MAX_OUTGOING_DATA 12
#define N_OUTBUF 12
#define __ALIGNED__ __attribute__((__aligned__(sizeof(long))))
struct hvsi_struct {
    struct work_struct writer;
    wait_queue_head_t emptyq; /* woken when outbuf is emptied */
    wait_queue_head_t stateq; /* woken when HVSI state changes */
    spinlock_t lock;
    int index;
    struct tty_struct *tty;
    unsigned int count;
    uint8_t throttle_buf[128];
    uint8_t outbuf[N_OUTBUF]; /* to implement write_room and chars_in_buffer */
    /* inbuf is for packet reassembly. leave a little room for leftovers. */
    uint8_t inbuf[HVSI_MAX_PACKET + HVSI_MAX_READ];
};
```

## Linux-Kernel: [RFC] Host Virtual Serial Interface driver

```
uint8_t *inbuf_end;
int n_throttle;
int n_outbuf;
uint32_t vtermno;
uint32_t virq;
atomic_t seqno; /* HVSI packet sequence number */
uint16_t mctrl;
uint8_t state; /* HVSI protocol state */
uint8_t flags;
#ifdef CONFIG_MAGIC_SYSRQ
uint8_t sysrq;
#endif /* CONFIG_MAGIC_SYSRQ */
};
static struct hvsi_struct hvsi_ports[MAX_NR_HVSI_CONSOLES];
static struct tty_driver *hvsi_driver;
static int hvsi_count;
static int (*hvsi_wait)(struct hvsi_struct *hp, int state);
enum HVSI_PROTOCOL_STATE {
    HVSI_CLOSED,
    HVSI_WAIT_FOR_VER_RESPONSE,
    HVSI_WAIT_FOR_VER_QUERY,
    HVSI_OPEN,
    HVSI_WAIT_FOR_MCTRL_RESPONSE,
};
#define HVSI_CONSOLE 0x1
#define VS_DATA_PACKET_HEADER 0xff
#define VS_CONTROL_PACKET_HEADER 0xfe
#define VS_QUERY_PACKET_HEADER 0xfd
#define VS_QUERY_RESPONSE_PACKET_HEADER 0xfc
/* control verbs */
#define VSV_SET_MODEM_CTL 1 /* to service processor only */
#define VSV_MODEM_CTL_UPDATE 2 /* from service processor only */
#define VSV_CLOSE_PROTOCOL 3
/* query verbs */
#define VSV_SEND_VERSION_NUMBER 1
#define VSV_SEND_MODEM_CTL_STATUS 2
/* yes, these masks are not consecutive. */
#define HVSI_TSDTR 0x01
#define HVSI_TSCD 0x20
struct hvsi_header {
    uint8_t type;
    uint8_t len;
    uint16_t seqno;
} __attribute__((packed));
struct hvsi_data {
    uint8_t type;
    uint8_t len;
    uint16_t seqno;
    uint8_t data[HVSI_MAX_OUTGOING_DATA];
} __attribute__((packed));
struct hvsi_control {
    uint8_t type;
    uint8_t len;
    uint16_t seqno;
    uint16_t verb;
    /* optional depending on verb: */
    uint32_t word;
    uint32_t mask;
} __attribute__((packed));
struct hvsi_query {
    uint8_t type;
    uint8_t len;
```

## Linux-Kernel: [RFC] Host Virtual Serial Interface driver

```
    uint16_t seqno;
    uint16_t verb;
} __attribute__((packed));
struct hvsi_query_response {
    uint8_t type;
    uint8_t len;
    uint16_t seqno;
    uint16_t verb;
    uint16_t query_seqno;
    union {
        uint8_t version;
        uint32_t mctrl_word;
    } u;
} __attribute__((packed));
static inline int is_open(struct hvsi_struct *hp)
{
    /* if we're waiting for an mctrl then we're already open */
    return (hp->state == HVSI_OPEN)
        || (hp->state == HVSI_WAIT_FOR_MCTRL_RESPONSE);
}
static inline void print_state(struct hvsi_struct *hp)
{
#ifdef DEBUG
    static const char *state_names[] = {
        "HVSI_CLOSED",
        "HVSI_WAIT_FOR_VER_RESPONSE",
        "HVSI_WAIT_FOR_VER_QUERY",
        "HVSI_OPEN",
        "HVSI_WAIT_FOR_MCTRL_RESPONSE",
    };
    const char *name = state_names[hp->state];
    if (hp->state > (sizeof(state_names)/sizeof(char*)))
        name = "UNKNOWN";
    pr_debug("hvsi%i: state = %s\n", hp->index, name);
#endif /* DEBUG */
}
static inline void __set_state(struct hvsi_struct *hp, int state)
{
    hp->state = state;
    print_state(hp);
    wake_up_all(&hp->stateq);
}
static inline void set_state(struct hvsi_struct *hp, int state)
{
    unsigned long flags;
    spin_lock_irqsave(&hp->lock, flags);
    __set_state(hp, state);
    spin_unlock_irqrestore(&hp->lock, flags);
}
static inline int hdrlen(const uint8_t *packet)
{
    const int lengths[] = { 4, 6, 6, 8, };
    struct hvsi_header *header = (struct hvsi_header *)packet;
    return lengths[VS_DATA_PACKET_HEADER - header->type];
}
static inline const uint8_t *payload(const uint8_t *packet)
{
    return packet + hdrlen(packet);
}
static inline int len_packet(const uint8_t *packet)
{
    return (int)((struct hvsi_header *)packet)->len;
}
```

## Linux-Kernel: [RFC] Host Virtual Serial Interface driver

```
}
static inline int len_payload(const uint8_t *packet)
{
    return len_packet(packet) - hdrllen(packet);
}
static inline int is_header(const uint8_t *packet)
{
    struct hvsi_header *header = (struct hvsi_header *)packet;
    return header->type >= VS_QUERY_RESPONSE_PACKET_HEADER;
}
static inline int got_packet(const struct hvsi_struct *hp, uint8_t *packet)
{
    if (hp->inbuf_end < packet + sizeof(struct hvsi_header))
        return 0; /* don't even have the packet header */
    if (hp->inbuf_end < (packet + len_packet(packet)))
        return 0; /* don't have the rest of the packet */
    return 1;
}
/* shift remaining bytes in packetbuf down */
static void compact_inbuf(struct hvsi_struct *hp, uint8_t *read_to)
{
    int remaining = (int)(hp->inbuf_end - read_to);
    pr_debug("%s: %i chars remain\n", __FUNCTION__, remaining);
    if (read_to != hp->inbuf) {
        memmove(hp->inbuf, read_to, remaining);
    }
    hp->inbuf_end = hp->inbuf + remaining;
}
static void dump_hex(const uint8_t *data, int len)
{
    int i;
    printk("    ");
    for (i=0; i < len; i++) {
        printk("%.2x", data[i]);
    }
    printk("\n    ");
    for (i=0; i < len; i++) {
        if (isprint(data[i]))
            printk("%c", data[i]);
        else
            printk(".");
    }
    printk("\n");
}
static void dump_packet(uint8_t *packet)
{
    struct hvsi_header *header = (struct hvsi_header *)packet;
    printk("type 0x%x, len %i, seqno %i:\n", header->type, header->len,
        header->seqno);
    dump_hex(packet, header->len);
}
/* can't use hvc_get_chars because that strips CRs */
static int hvsi_read(struct hvsi_struct *hp, char *buf, int count)
{
    unsigned long got;
    if (plpar_hcall(H_GET_TERM_CHAR, hp->vtermno, 0, 0, 0, &got,
        (unsigned long *)buf, (unsigned long *)buf+1) == H_Success) {
        return got;
    }
    return 0;
}
/*
```

## Linux-Kernel: [RFC] Host Virtual Serial Interface driver

```
* we can't call tty_hangup() directly here because we need to call that
* outside of our lock
*/
static struct tty_struct *hvsi_recv_control(struct hvsi_struct *hp,
                                           uint8_t *packet)
{
    struct tty_struct *to_hangup = NULL;
    struct hvsi_control *header = (struct hvsi_control *)packet;
    switch (header->verb) {
        case VSV_MODEM_CTL_UPDATE:
            if ((header->word & HVSI_TSCD) == 0) {
                /* CD went away; no more connection */
                pr_debug("hvsi%i: CD dropped\n", hp->index);
                hp->mctrl &= TIOCM_CD;
                if (!(hp->tty->flags & CLOCAL))
                    to_hangup = hp->tty;
            }
            break;
        case VSV_CLOSE_PROTOCOL:
            #warning here
            printk(KERN_DEBUG
                  "hvsi%i: service processor closed connection!\n", hp->index);
            __set_state(hp, HVSI_CLOSED);
            to_hangup = hp->tty;
            hp->tty = NULL;
            break;
        default:
            printk(KERN_WARNING "hvsi%i: unknown HVSI control packet: ",
                  hp->index);
            dump_hex(packet, header->len);
            break;
    }
    return to_hangup;
}

static void hvsi_recv_response(struct hvsi_struct *hp, uint8_t *packet)
{
    struct hvsi_query_response *resp = (struct hvsi_query_response *)packet;
    switch (hp->state) {
        case HVSI_WAIT_FOR_VER_RESPONSE:
            __set_state(hp, HVSI_WAIT_FOR_VER_QUERY);
            break;
        case HVSI_WAIT_FOR_MCTRL_RESPONSE:
            hp->mctrl = 0;
            if (resp->u.mctrl_word & HVSI_TSDTR)
                hp->mctrl |= TIOCM_DTR;
            if (resp->u.mctrl_word & HVSI_TSCD)
                hp->mctrl |= TIOCM_CD;
            __set_state(hp, HVSI_OPEN);
            break;
        default:
            printk(KERN_ERR "hvsi%i: unexpected query response: ", hp->index);
            dump_packet(packet);
            break;
    }
}

/* respond to service processor's version query */
static int hvsi_version_respond(struct hvsi_struct *hp, uint16_t query_seqno)
{
    struct hvsi_query_response packet __ALIGNED__;
    int wrote;
    packet.type = VS_QUERY_RESPONSE_PACKET_HEADER,
    packet.len = sizeof(struct hvsi_query_response),

```

## Linux-Kernel: [RFC] Host Virtual Serial Interface driver

```
    packet.seqno = atomic_inc_return(&hp->seqno);
    packet.verb = VSV_SEND_VERSION_NUMBER,
    packet.u.version = HVSI_VERSION,
    packet.query_seqno = query_seqno+1,
#ifdef DEBUG
    pr_debug("%s: sending %i bytes\n", __FUNCTION__, packet.len);
    dump_hex((uint8_t*)&packet, packet.len);
#endif
}
#endif

wrote = hvc_put_chars(hp->vtermno, (char *)&packet, packet.len);
if (wrote != packet.len) {
    printk(KERN_ERR "hvsi%i: couldn't send query response!\n",
           hp->index);
    return -EIO;
}
return 0;
}

static void hvsi_recv_query(struct hvsi_struct *hp, uint8_t *packet)
{
    struct hvsi_query *query = (struct hvsi_query *)packet;
    switch (hp->state) {
        case HVSI_WAIT_FOR_VER_QUERY:
            __set_state(hp, HVSI_OPEN);
            hvsi_version_respond(hp, query->seqno);
            break;
        default:
            printk(KERN_ERR "hvsi%i: unexpected query: ", hp->index);
            dump_packet(packet);
            break;
    }
}

static void hvsi_insert_chars(struct hvsi_struct *hp, const char *buf, int len)
{
    int i;
    for (i=0; i < len; i++) {
        char c = buf[i];
#ifdef CONFIG_MAGIC_SYSRQ
        if (c == '\0') {
            hp->sysrq = 1;
            continue;
        } else if (hp->sysrq) {
            handle_sysrq(c, NULL, hp->tty);
            hp->sysrq = 0;
            continue;
        }
#endif /* CONFIG_MAGIC_SYSRQ */
        tty_insert_flip_char(hp->tty, c, 0);
    }
}

/*
 * We could get 252 bytes of data at once here. But the tty layer only
 * throttles us at TTY_THRESHOLD_THROTTLE (128) bytes, so we could overflow
 * it. Accordingly we won't send more than 128 bytes at a time to the flip
 * buffer, which will give the tty buffer a chance to throttle us. Should the
 * value of TTY_THRESHOLD_THROTTLE change in n_tty.c, this code should be
 * revisited.
 */
#define TTY_THRESHOLD_THROTTLE 128
static struct tty_struct *hvsi_recv_data(struct hvsi_struct *hp,
                                         const uint8_t *packet)
{
    const uint8_t *data = payload(packet);
    int datalen = len_payload(packet);
}
```

## Linux-Kernel: [RFC] Host Virtual Serial Interface driver

```
int overflow = datalen - TTY_THRESHOLD_THROTTLE;
pr_debug("queueing %i chars '%.*s'\n", datalen, datalen, data);
if (datalen == 0)
    return NULL;
if (overflow > 0) {
    pr_debug("%s: got >TTY_THRESHOLD_THROTTLE bytes\n", __FUNCTION__);
    datalen = TTY_THRESHOLD_THROTTLE;
}
hvsi_insert_chars(hp, data, datalen);
if (overflow > 0) {
    /*
     * we still have more data to deliver, so we need to save off the
     * overflow and send it later
     */
    pr_debug("%s: deferring overflow\n", __FUNCTION__);
    memcpy(hp->throttle_buf, data + TTY_THRESHOLD_THROTTLE, overflow);
    hp->n_throttle = overflow;
}
return hp->tty;
}
/*
 * Returns true/false indicating data successfully read from hypervisor.
 * Used both to get packets for tty connections and to advance the state
 * machine during console handshaking (in which case tty = NULL and we ignore
 * incoming data).
 */
static int hvsi_load_chunk(struct hvsi_struct *hp, struct tty_struct **flip,
                          struct tty_struct **hangup)
{
    uint8_t *packet = hp->inbuf;
    int chunklen;
    *flip = NULL;
    *hangup = NULL;
    chunklen = hvsi_read(hp, hp->inbuf_end, HVSI_MAX_READ);
    if (chunklen == 0)
        return 0;
#ifdef DEBUG
    pr_debug("%s: got %i bytes\n", __FUNCTION__, chunklen);
    dump_hex(hp->inbuf_end, chunklen);
#endif
    hp->inbuf_end += chunklen;
    /* handle all completed packets */
    while ((packet < hp->inbuf_end) && got_packet(hp, packet)) {
        struct hvsi_header *header = (struct hvsi_header *)packet;
        if (!is_header(packet)) {
            printk(KERN_ERR "hvsi%i: got malformed packet\n", hp->index);
            /* skip bytes until we find a header or run out of data */
            while ((packet < hp->inbuf_end) && (!is_header(packet))) {
                packet++;
            }
            continue;
        }
#ifdef DEBUG
        pr_debug("%s: handling %i-byte packet\n", __FUNCTION__,
                len_packet(packet));
        dump_packet(packet);
#endif
        switch (header->type) {
            case VS_DATA_PACKET_HEADER:
                if (!is_open(hp)) {
                    break;
                }
        }
    }
}
```

## Linux-Kernel: [RFC] Host Virtual Serial Interface driver

```
        if (hp->tty == NULL) {
            /* no tty buffer to put data in */
            break;
        }
        *flip = hvsi_recv_data(hp, packet);
        break;
    case VS_CONTROL_PACKET_HEADER:
        *hangup = hvsi_recv_control(hp, packet);
        break;
    case VS_QUERY_RESPONSE_PACKET_HEADER:
        hvsi_recv_response(hp, packet);
        break;
    case VS_QUERY_PACKET_HEADER:
        hvsi_recv_query(hp, packet);
        break;
    default:
        printk(KERN_ERR "hvsi%i: unknown HVSI packet type 0x%x\n",
                hp->index, header->type);
        dump_packet(packet);
        break;
    }
    packet += len_packet(packet);
    if (*hangup) {
        pr_debug("%s: hangup\n", __FUNCTION__);
        /*
         * we need to send the hangup now before receiving any more data.
         * If we get "data, hangup, data", we can't deliver the second
         * data before the hangup.
         */
        break;
    }
}
compact_inbuf(hp, packet);
return 1;
}
static void hvsi_send_overflow(struct hvsi_struct *hp)
{
    pr_debug("%s: delivering %i bytes overflow\n", __FUNCTION__,
            hp->n_throttle);
    hvsi_insert_chars(hp, hp->throttle_buf, hp->n_throttle);
    hp->n_throttle = 0;
}
/*
 * must get all pending data because we only get an irq on empty->non-empty
 * transition
 */
static irqreturn_t hvsi_interrupt(int irq, void *arg, struct pt_regs *regs)
{
    struct hvsi_struct *hp = (struct hvsi_struct *)arg;
    struct tty_struct *flip;
    struct tty_struct *hangup;
    unsigned long flags;
    irqreturn_t handled = IRQ_NONE;
    int again = 1;
    pr_debug("%s\n", __FUNCTION__);
    while (again) {
        spin_lock_irqsave(&hp->lock, flags);
        again = hvsi_load_chunk(hp, &flip, &hangup);
        handled = IRQ_HANDLED;
        spin_unlock_irqrestore(&hp->lock, flags);
        /*
         * we have to call tty_flip_buffer_push() and tty_hangup() outside our

```

## Linux-Kernel: [RFC] Host Virtual Serial Interface driver

```

    * spinlock. But we also have to keep going until we've read all the
    * available data.
    */
    if (flip) {
        /* there was data put in the tty flip buffer */
        tty_flip_buffer_push(flip);
        flip = NULL;
    }
    if (hangup) {
        tty_hangup(hangup);
    }
}
spin_lock_irqsave(&hp->lock, flags);
if (hp->tty && hp->n_throttle
    && (!test_bit(TTY_THROTTLED, &hp->tty->flags))) {
    /* we weren't hung up and we weren't throttled, so we can deliver the
    * rest now */
    flip = hp->tty;
    hvsi_send_overflow(hp);
}
spin_unlock_irqrestore(&hp->lock, flags);
if (flip) {
    tty_flip_buffer_push(flip);
}
return handled;
}
/* for boot console, before the irq handler is running */
static int __init poll_for_state(struct hvsi_struct *hp, int state)
{
    unsigned long end_jiffies = jiffies + HVSI_TIMEOUT;
    for (;;) {
        hvsi_interrupt(hp->virq, (void *)hp, NULL); /* get pending data */
        if (hp->state == state) {
            return 0;
        }
        mdelay(5);
        if (time_after(jiffies, end_jiffies)) {
            return -EIO;
        }
    }
}
/* wait for irq handler to change our state */
static int wait_for_state(struct hvsi_struct *hp, int state)
{
    unsigned long end_jiffies = jiffies + HVSI_TIMEOUT;
    unsigned long timeout;
    int ret = 0;
    DECLARE_WAITQUEUE(myself, current);
    set_current_state(TASK_INTERRUPTIBLE);
    add_wait_queue(&hp->stateq, &myself);
    for (;;) {
        set_current_state(TASK_INTERRUPTIBLE);
        if (hp->state == state)
            break;
        timeout = end_jiffies - jiffies;
        if (time_after(jiffies, end_jiffies)) {
            ret = -EIO;
            break;
        }
        schedule_timeout(timeout);
    }
    remove_wait_queue(&hp->stateq, &myself);
}
```

## Linux-Kernel: [RFC] Host Virtual Serial Interface driver

```
    set_current_state(TASK_RUNNING);
    return ret;
}
static int hvsi_query(struct hvsi_struct *hp, uint16_t verb)
{
    struct hvsi_query packet __ALIGNED__;
    int wrote;
    packet.type = VS_QUERY_PACKET_HEADER;
    packet.len = sizeof(struct hvsi_query);
    packet.seqno = atomic_inc_return(&hp->seqno);
    packet.verb = verb;
#ifdef DEBUG
    pr_debug("%s: sending %i bytes\n", __FUNCTION__, packet.len);
    dump_hex((uint8_t*)&packet, packet.len);
#endif
    wrote = hvc_put_chars(hp->vtermno, (char *)&packet, packet.len);
    if (wrote != packet.len) {
        printk(KERN_ERR "hvsi%i: couldn't send query (%i)!\n", hp->index,
               wrote);
        return -EIO;
    }
    return 0;
}
static int hvsi_get_mctrl(struct hvsi_struct *hp)
{
    int ret;
    set_state(hp, HVSI_WAIT_FOR_MCTRL_RESPONSE);
    hvsi_query(hp, VSV_SEND_MODEM_CTL_STATUS);
    ret = hvsi_wait(hp, HVSI_OPEN);
    if (ret < 0) {
        printk(KERN_ERR "hvsi%i: didn't get modem flags\n", hp->index);
        set_state(hp, HVSI_OPEN);
        return ret;
    }
    pr_debug("%s: mctrl 0x%x\n", __FUNCTION__, hp->mctrl);
    return 0;
}
/* note that we can only set DTR */
static int hvsi_set_mctrl(struct hvsi_struct *hp, uint16_t mctrl)
{
    struct hvsi_control packet __ALIGNED__;
    int wrote;
    packet.type = VS_CONTROL_PACKET_HEADER;
    packet.seqno = atomic_inc_return(&hp->seqno);
    packet.len = sizeof(struct hvsi_control);
    packet.verb = VSV_SET_MODEM_CTL;
    packet.mask = HVSI_TSDTR;
    if (mctrl & TIOCM_DTR)
        packet.word = HVSI_TSDTR;
#ifdef DEBUG
    pr_debug("%s: sending %i bytes\n", __FUNCTION__, packet.len);
    dump_hex((uint8_t*)&packet, packet.len);
#endif
    wrote = hvc_put_chars(hp->vtermno, (char *)&packet, packet.len);
    if (wrote != packet.len) {
        printk(KERN_ERR "hvsi%i: couldn't set DTR!\n", hp->index);
        return -EIO;
    }
    return 0;
}
static void hvsi_drain_input(struct hvsi_struct *hp)
{

```

## Linux-Kernel: [RFC] Host Virtual Serial Interface driver

```
uint8_t buf[HVSI_MAX_READ] __ALIGNED__;
unsigned long end_jiffies = jiffies + HVSI_TIMEOUT;
while (time_before(end_jiffies, jiffies)) {
    if (0 == hvsi_read(hp, buf, HVSI_MAX_READ))
        break;
}
}
static int hvsi_handshake(struct hvsi_struct *hp)
{
    int ret;
    /*
     * We could have a CLOSE or other data waiting for us before we even try
     * to open; try to throw it all away so we don't get confused. (CLOSE
     * is the first message sent up the pipe when the FSP comes online. We
     * need to distinguish between "it came up a while ago and we're the first
     * user" and "it was just reset before it saw our handshake packet".)
     */
    hvsi_drain_input(hp);
    set_state(hp, HVSI_WAIT_FOR_VER_RESPONSE);
    ret = hvsi_query(hp, VSV_SEND_VERSION_NUMBER);
    if (ret < 0) {
        printk(KERN_ERR "hvsi%i: couldn't send version query\n", hp->index);
        return ret;
    }
    ret = hvsi_wait(hp, HVSI_OPEN);
    if (ret < 0) {
        return ret;
    }
    return 0;
}
static int hvsi_put_chars(struct hvsi_struct *hp, const char *buf, int count)
{
    struct hvsi_data packet __ALIGNED__;
    int ret;
    BUG_ON(count > HVSI_MAX_OUTGOING_DATA);
    packet.type = VS_DATA_PACKET_HEADER;
    packet.seqno = atomic_inc_return(&hp->seqno);
    packet.len = count + sizeof(struct hvsi_header);
    memcpy(&packet.data, buf, count);
    ret = hvc_put_chars(hp->vtermno, (char *)&packet, packet.len);
    if (ret == packet.len) {
        /* return the number of chars written, not the packet length */
        return count;
    }
    return ret; /* return any errors */
}
static void hvsi_close_protocol(struct hvsi_struct *hp)
{
    struct hvsi_control packet __ALIGNED__;
    packet.type = VS_CONTROL_PACKET_HEADER;
    packet.seqno = atomic_inc_return(&hp->seqno);
    packet.len = 6;
    packet.verb = VSV_CLOSE_PROTOCOL;
#ifdef DEBUG
    pr_debug("%s: sending %i bytes\n", __FUNCTION__, packet.len);
    dump_hex((uint8_t*)&packet, packet.len);
#endif
    hvc_put_chars(hp->vtermno, (char *)&packet, packet.len);
}
static int hvsi_open(struct tty_struct *tty, struct file *filp)
{
    struct hvsi_struct *hp;
```

## Linux-Kernel: [RFC] Host Virtual Serial Interface driver

```
unsigned long flags;
int line = tty->index;
int ret;
pr_debug("%s\n", __FUNCTION__);
if (line < 0 || line >= hvsi_count)
    return -ENODEV;
hp = &hvsi_ports[line];
tty->driver_data = hp;
tty->low_latency = 1; /* avoid throttle/tty_flip_buffer_push race */
spin_lock_irqsave(&hp->lock, flags);
hp->tty = tty;
hp->count++;
atomic_set(&hp->seqno, 0);
h_vio_signal(hp->vtermno, VIO_IRQ_ENABLE);
spin_unlock_irqrestore(&hp->lock, flags);
if (hp->flags & HVSI_CONSOLE) {
    /* this has already been handshaked as the console */
    return 0;
}
ret = hvsi_handshake(hp);
if (ret < 0) {
    printk(KERN_ERR "%s: HVSI handshaking failed\n", tty->name);
    return ret;
}
ret = hvsi_get_mctrl(hp);
if (ret < 0) {
    printk(KERN_ERR "%s: couldn't get initial modem flags\n", tty->name);
    return ret;
}
ret = hvsi_set_mctrl(hp, hp->mctrl | TIOCM_DTR);
if (ret < 0) {
    printk(KERN_ERR "%s: couldn't set DTR\n", tty->name);
    return ret;
}
return 0;
}
/* wait for hvsi_write_worker to empty hp->outbuf */
static void hvsi_flush_output(struct hvsi_struct *hp)
{
    unsigned long end_jiffies = jiffies + HVSI_TIMEOUT;
    unsigned long timeout;
    DECLARE_WAITQUEUE(myself, current);
    set_current_state(TASK_UNINTERRUPTIBLE);
    add_wait_queue(&hp->emptyq, &myself);
    for (;;) {
        set_current_state(TASK_UNINTERRUPTIBLE);
        if (hp->n_outbuf <= 0)
            break;
        timeout = end_jiffies - jiffies;
        if (time_after(jiffies, end_jiffies))
            break;
        schedule_timeout(timeout);
    }
    remove_wait_queue(&hp->emptyq, &myself);
    set_current_state(TASK_RUNNING);
    /* 'writer' could still be pending if it didn't see n_outbuf = 0 yet */
    cancel_delayed_work(&hp->writer);
    flush_scheduled_work();
    /*
     * it's also possible that our timeout expired and hvsi_write_worker
     * didn't manage to push outbuf. poof.
     */
}
```

## Linux-Kernel: [RFC] Host Virtual Serial Interface driver

```
    hp->n_outbuf = 0;
}
static void hvsi_close(struct tty_struct *tty, struct file *filp)
{
    struct hvsi_struct *hp = tty->driver_data;
    unsigned long flags;
    pr_debug("%s\n", __FUNCTION__);
    if (tty_hung_up_p(filp))
        return;
    spin_lock_irqsave(&hp->lock, flags);
    if ((--hp->count == 0) && !(hp->flags & HVSI_CONSOLE)) {
        h_vio_signal(hp->vtermno, VIO_IRQ_DISABLE); /* no more irqs */
        __set_state(hp, HVSI_CLOSED);
        /*
         * any data delivered to the tty layer after this will be discarded
         * (except for XON/XOFF)
         */
        tty->closing = 1;
        spin_unlock_irqrestore(&hp->lock, flags);
        /* let any existing irq handlers finish. no more will start. */
        synchronize_irq(hp->virq);
        /* hvsi_write_worker will re-schedule itself until outbuf is empty. */
        hvsi_flush_output(hp);
        /* tell FSP to stop sending data */
        hvsi_close_protocol(hp);
        /*
         * drain anything FSP is still in the middle of sending, and let
         * hvsi_handshake drain the rest on the next open.
         */
        hvsi_drain_input(hp);
        spin_lock_irqsave(&hp->lock, flags);
        hp->tty = NULL;
        hp->inbuf_end = hp->inbuf; /* discard remaining partial packets */
    } else if (hp->count < 0)
        printk(KERN_ERR "hvsi_close %lu: oops, count is %d\n",
               hp - hvsi_ports, hp->count);
    spin_unlock_irqrestore(&hp->lock, flags);
}
static void hvsi_hangup(struct tty_struct *tty)
{
    struct hvsi_struct *hp = tty->driver_data;
    pr_debug("%s\n", __FUNCTION__);
    hp->count = 0;
    hp->tty = NULL;
}
/* called with hp->lock held */
static void hvsi_push(struct hvsi_struct *hp)
{
    int n;
    if (hp->n_outbuf <= 0)
        return;
    n = hvsi_put_chars(hp, hp->outbuf, hp->n_outbuf);
    if (n != 0) {
        /*
         * either all data was sent or there was an error, and we throw away
         * data on error.
         */
        hp->n_outbuf = 0;
    }
}
/* hvsi_write_worker will keep rescheduling itself until outbuf is empty */
static void hvsi_write_worker(void *arg)
```

## Linux-Kernel: [RFC] Host Virtual Serial Interface driver

```

{
    struct hvsi_struct *hp = (struct hvsi_struct *)arg;
    unsigned long flags;
#ifdef DEBUG
    static long start_j = 0;
    if (start_j == 0)
        start_j = jiffies;
#endif /* DEBUG */
    spin_lock_irqsave(&hp->lock, flags);
    hvsi_push(hp);
    if (hp->n_outbuf > 0) {
        schedule_delayed_work(&hp->writer, 10);
    } else {
#ifdef DEBUG
        pr_debug("%s: outbuf emptied after %li jiffies\n", __FUNCTION__,
                jiffies - start_j);
        start_j = 0;
#endif /* DEBUG */
        wake_up_all(&hp->emptyq);
        if (test_bit(TTY_DO_WRITE_WAKEUP, &hp->tty->flags)
            && hp->tty->ldisc.write_wakeup) {
            hp->tty->ldisc.write_wakeup(hp->tty);
        }
        wake_up_interruptible(&hp->tty->write_wait);
    }
    spin_unlock_irqrestore(&hp->lock, flags);
}
static int hvsi_write_room(struct tty_struct *tty)
{
    struct hvsi_struct *hp = (struct hvsi_struct *)tty->driver_data;
    return N_OUTBUF - hp->n_outbuf;
}
static int hvsi_chars_in_buffer(struct tty_struct *tty)
{
    struct hvsi_struct *hp = (struct hvsi_struct *)tty->driver_data;
    return hp->n_outbuf;
}
static int hvsi_write(struct tty_struct *tty, int from_user,
                    const unsigned char *buf, int count)
{
    struct hvsi_struct *hp = tty->driver_data;
    const char *source = buf;
    char *kbuf;
    unsigned long flags;
    int total = 0;
    int origcount = count;
    if (from_user) {
        kbuf = kmalloc(count, GFP_KERNEL);
        if (kbuf == NULL)
            return -ENOMEM;
        if (copy_from_user(kbuf, buf, count)) {
            kfree(kbuf);
            return -EFAULT;
        }
        source = kbuf;
    }
    spin_lock_irqsave(&hp->lock, flags);
    if (!is_open(hp)) {
        /* we're either closing or not yet open; don't accept data */
        pr_debug("%s: not open\n", __FUNCTION__);
        goto out;
    }
}

```

## Linux-Kernel: [RFC] Host Virtual Serial Interface driver

```
/*
 * when the hypervisor buffer (16K) fills, data will stay in hp->outbuf
 * and hvsi_write_worker will be scheduled. subsequent hvsi_write() calls
 * will see there is no room in outbuf and return.
 */
while ((count > 0) && (hvsi_write_room(hp->tty) > 0)) {
    int chunksize = min(count, hvsi_write_room(hp->tty));
    BUG_ON(hp->n_outbuf < 0);
    memcpy(hp->outbuf + hp->n_outbuf, source, chunksize);
    hp->n_outbuf += chunksize;
    total += chunksize;
    source += chunksize;
    count -= chunksize;
    hvsi_push(hp);
}
if (hp->n_outbuf > 0) {
    /*
     * we weren't able to write it all to the hypervisor.
     * schedule another push attempt.
     */
    schedule_delayed_work(&hp->writer, 10);
}
out:
spin_unlock_irqrestore(&hp->lock, flags);
if (from_user)
    kfree(kbuf);
if (total != origcount)
    pr_debug("%s: wanted %i, only wrote %i\n", __FUNCTION__, origcount,
            total);
return total;
}
/*
 * I have never seen throttle or unthrottle called, so this little throttle
 * buffering scheme may or may not work.
 */
static void hvsi_throttle(struct tty_struct *tty)
{
    struct hvsi_struct *hp = (struct hvsi_struct *)tty->driver_data;
    pr_debug("%s\n", __FUNCTION__);
    h_vio_signal(hp->vtermno, VIO_IRQ_DISABLE);
}
static void hvsi_unthrottle(struct tty_struct *tty)
{
    struct hvsi_struct *hp = (struct hvsi_struct *)tty->driver_data;
    unsigned long flags;
    int shouldflip = 0;
    pr_debug("%s\n", __FUNCTION__);
    spin_lock_irqsave(&hp->lock, flags);
    if (hp->n_throttle) {
        hvsi_send_overflow(hp);
        shouldflip = 1;
    }
    spin_unlock_irqrestore(&hp->lock, flags);
    if (shouldflip)
        tty_flip_buffer_push(hp->tty);
    h_vio_signal(hp->vtermno, VIO_IRQ_ENABLE);
}
static int hvsi_tiocmget(struct tty_struct *tty, struct file *file)
{
    struct hvsi_struct *hp = (struct hvsi_struct *)tty->driver_data;
    hvsi_get_mctrl(hp);
    return hp->mctrl;
}
```

## Linux-Kernel: [RFC] Host Virtual Serial Interface driver

```
}
static int hvsi_tiocmset(struct tty_struct *tty, struct file *file,
                        unsigned int set, unsigned int clear)
{
    struct hvsi_struct *hp = (struct hvsi_struct *)tty->driver_data;
    unsigned long flags;
    uint16_t new_mctrl;
    /* we can only alter DTR */
    clear &= TIOCM_DTR;
    set &= TIOCM_DTR;
    spin_lock_irqsave(&hp->lock, flags);
    new_mctrl = (hp->mctrl & ~clear) | set;
    if (hp->mctrl != new_mctrl) {
        hvsi_set_mctrl(hp, new_mctrl);
        hp->mctrl = new_mctrl;
    }
    spin_unlock_irqrestore(&hp->lock, flags);
    return 0;
}

static struct tty_operations hvsi_ops = {
    .open = hvsi_open,
    .close = hvsi_close,
    .write = hvsi_write,
    .hangup = hvsi_hangup,
    .write_room = hvsi_write_room,
    .chars_in_buffer = hvsi_chars_in_buffer,
    .throttle = hvsi_throttle,
    .unthrottle = hvsi_unthrottle,
    .tiocmget = hvsi_tiocmget,
    .tiocmset = hvsi_tiocmset,
};

static int __init hvsi_init(void)
{
    int i;
    hvsi_driver = alloc_tty_driver(hvsi_count);
    if (!hvsi_driver)
        return -ENOMEM;
    hvsi_driver->owner = THIS_MODULE;
    hvsi_driver->devfs_name = "hvsi/";
    hvsi_driver->driver_name = "hvsi";
    hvsi_driver->name = "hvsi";
    hvsi_driver->major = HVSI_MAJOR;
    hvsi_driver->minor_start = HVSI_MINOR;
    hvsi_driver->type = TTY_DRIVER_TYPE_SYSTEM;
    hvsi_driver->init_termios = tty_std_termios;
    hvsi_driver->init_termios.c_cflag = B9600 | CS8 | CREAD | HUPCL;
    hvsi_driver->flags = TTY_DRIVER_REAL_RAW;
    tty_set_operations(hvsi_driver, &hvsi_ops);
    for (i=0; i < hvsi_count; i++) {
        struct hvsi_struct *hp = &hvsi_ports[i];
        int ret = 1;
        ret = request_irq(hp->virq, hvsi_interrupt, SA_INTERRUPT, "hvsi", hp);
        if (ret) {
            printk(KERN_ERR "HVSI: couldn't reserve irq 0x%x (error %i)\n",
                   hp->virq, ret);
        }
    }
    hvsi_wait = wait_for_state; /* irqs active now */
    if (tty_register_driver(hvsi_driver))
        panic("Couldn't register hvsi console driver\n");
    printk(KERN_INFO "HVSI: registered %i devices\n", hvsi_count);
    return 0;
}
```

## Linux-Kernel: [RFC] Host Virtual Serial Interface driver

```

}
device_initcall(hvsi_init);
/***** console (not tty) code: *****/
static void hvsi_console_print(struct console *console, const char *buf,
                              unsigned int count)
{
    struct hvsi_struct *hp = &hvsi_ports[console->index];
    char c[HVSI_MAX_OUTGOING_DATA] __ALIGNED__;
    unsigned int i = 0, n = 0;
    int ret, donecr = 0;
    mb();
    if (!is_open(hp)) {
        return;
    }
    /*
     * ugh, we have to translate LF -> CRLF ourselves, in place.
     * copied from hvc_console.c:
     */
    while (count > 0 || i > 0) {
        if (count > 0 && i < sizeof(c)) {
            if (buf[n] == '\n' && !donecr) {
                c[i++] = '\r';
                donecr = 1;
            } else {
                c[i++] = buf[n++];
                donecr = 0;
                --count;
            }
        } else {
            ret = hvsi_put_chars(hp, c, i);
            if (ret < 0)
                i = 0;
            i -= ret;
        }
    }
}
static struct tty_driver *hvsi_console_device(struct console *console,
                                              int *index)
{
    *index = console->index;
    return hvsi_driver;
}
static int __init hvsi_console_setup(struct console *console, char *options)
{
    struct hvsi_struct *hp = &hvsi_ports[console->index];
    int ret;
    if (console->index < 0 || console->index >= hvsi_count)
        return -1;
    /* give the FSP a chance to change the baud rate when we re-open */
    hvsi_close_protocol(hp);
    ret = hvsi_handshake(hp);
    if (ret < 0)
        return ret;
    ret = hvsi_get_mctrl(hp);
    if (ret < 0)
        return ret;
    ret = hvsi_set_mctrl(hp, hp->mctrl | TIOCM_DTR);
    if (ret < 0)
        return ret;
    hp->flags |= HVSI_CONSOLE;
    return 0;
}

```

## Linux-Kernel: [RFC] Host Virtual Serial Interface driver

```
static struct console hvsi_con_driver = {
    .name           = "hvsi",
    .write          = hvsi_console_print,
    .device         = hvsi_console_device,
    .setup          = hvsi_console_setup,
    .flags          = CON_PRINTBUFFER,
    .index          = -1,
};
static int __init hvsi_console_init(void)
{
    struct device_node *vty;
    hvsi_wait = poll_for_state; /* no irqs yet; must poll */
    /* search device tree for vty nodes */
    for (vty = of_find_compatible_node(NULL, "serial", "hvterm-protocol");
         vty != NULL;
         vty = of_find_compatible_node(vty, "serial", "hvterm-protocol")) {
        struct hvsi_struct *hp;
        uint32_t *vtermno;
        uint32_t *irq;
        vtermno = (uint32_t *)get_property(vty, "reg", NULL);
        irq = (uint32_t *)get_property(vty, "interrupts", NULL);
        if (!vtermno || !irq)
            continue;
        if (hvsi_count >= MAX_NR_HVSI_CONSOLES) {
            of_node_put(vty);
            break;
        }
        hp = &hvsi_ports[hvsi_count];
        INIT_WORK(&hp->writer, hvsi_write_worker, hp);
        init_waitqueue_head(&hp->emptyq);
        init_waitqueue_head(&hp->stateq);
        hp->lock = SPIN_LOCK_UNLOCKED;
        hp->index = hvsi_count;
        hp->inbuf_end = hp->inbuf;
        hp->state = HVSI_CLOSED;
        hp->vtermno = *vtermno;
        hp->virq = virt_irq_create_mapping(irq[0]);
        if (hp->virq == NO_IRQ) {
            printk(KERN_ERR "%s: couldn't create irq mapping for 0x%x\n",
                   __FUNCTION__, hp->virq);
            continue;
        } else {
            hp->virq = irq_offset_up(hp->virq);
        }
        hvsi_count++;
    }
    if (hvsi_count)
        register_console(&hvsi_con_driver);
    return 0;
}
console_initcall(hvsi_console_init);
-
```

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in the body of a message to majordomo@vger.kernel.org  
More majordomo info at <http://vger.kernel.org/majordomo-info.html>  
Please read the FAQ at <http://www.tux.org/lkml/>