```
/*
 * TTY abstractions --- taken from include/linux/tty.h,
 * include/linux/tty_driver.h, and include/linux/tty_ldisc.h
 */

/*
 * Where all of the state associated with a tty is kept while the tty
 * is open.  Since the termios state should be kept even if the tty
 * has been closed --- for things like the baud rate, etc --- it is
 * not stored here, but rather a pointer to the real state is stored
 * here.  Possible the winsize structure should have the same
 * treatment, but (1) the default 80x24 is usually right and (2) it's
 * most often used by a windowing system, which will set the correct
 * size each time the window is created or resized anyway.
 * IMPORTANT: since this structure is dynamically allocated, it must
 * be no larger than 4096 bytes.  Changing TTY_BUF_SIZE will change
 * the size of this structure, and it needs to be done with care.
 *                                              - TYT, 9/14/92
 */
struct tty_struct {
        int     magic;
        struct tty_driver driver;
        struct tty_ldisc ldisc;
        struct termios *termios, *termios_locked;
        int pgrp;
        int session;
        kdev_t  device;
        unsigned long flags;
        int count;
        struct winsize winsize;
        unsigned char stopped:1, hw_stopped:1, packet:1;
        unsigned char ctrl_status;

        struct tty_struct *link;
        struct fasync_struct *fasync;
        struct tty_flip_buffer flip;
        int max_flip_cnt;
        struct wait_queue *write_wait;
        struct wait_queue *read_wait;
        void *disc_data;
        void *driver_data;

#define N_TTY_BUF_SIZE 4096

        /*
         * The following is data for the N_TTY line discipline.  For
         * historical reasons, this is included in the tty structure.
         */
        unsigned int column;
        unsigned char lnext:1, erasing:1, raw:1, real_raw:1, icanon:1;
        unsigned char closing:1;
        unsigned short minimum_to_wake;
        unsigned overrun_time;
        int num_overrun;
        unsigned long process_char_map[256/(8*sizeof(unsigned long))];
        char *read_buf;
        int read_head;
        int read_tail;
        int read_cnt;
        unsigned long read_flags[N_TTY_BUF_SIZE/(8*sizeof(unsigned long))];
```

```
        int canon_data;
        unsigned long canon_head;
        unsigned int canon_column;
};

/* tty magic number */
#define TTY_MAGIC               0x5401

/*
 * These bits are used in the flags field of the tty structure.
 *
 * So that interrupts won't be able to mess up the queues,
 * copy_to_cooked must be atomic with respect to itself, as must
 * tty->write.  Thus, you must use the inline functions set_bit() and
 * clear_bit() to make things atomic.
 */
#define TTY_THROTTLED 0
#define TTY_IO_ERROR 1
#define TTY_OTHER_CLOSED 2
#define TTY_EXCLUSIVE 3
#define TTY_DEBUG 4
#define TTY_DO_WRITE_WAKEUP 5
#define TTY_PUSH 6
#define TTY_CLOSING 7

/*
 * This is the flip buffer used for the tty driver.  The buffer is
 * located in the tty structure, and is used as a high speed interface
 * between the tty driver and the tty line discipline.
 */
#define TTY_FLIPBUF_SIZE 512

struct tty_flip_buffer {
        struct tq_struct tqueue;
        unsigned char   char_buf[2*TTY_FLIPBUF_SIZE];
        char            flag_buf[2*TTY_FLIPBUF_SIZE];
        char            *char_buf_ptr;
        unsigned char   *flag_buf_ptr;
        int             count;
        int             buf_num;
};

/*
 * When a break, frame error, or parity error happens, these codes are
 * stuffed into the flags buffer.
 */
#define TTY_NORMAL      0
#define TTY_BREAK       1
#define TTY_FRAME       2
#define TTY_PARITY      3
#define TTY_OVERRUN     4


/*
 * This structure defines the interface between the tty line discpline
 * implementation and the tty routines.  The following routines can be
 * defined; unless noted otherwise, they are optional, and can be
 * filled in with a null pointer.
 *
 * int  (*open)(struct tty_struct *);
```

```
 *
 *      This function is called when the line discpline is associated
 *      with the tty.  The line discpline can use this as an
 *      opportunity to initialize any state needed by the ldisc routines.
 *
 * void (*close)(struct tty_struct *);
 *
 *      This function is called when the line discpline is being
 *      shutdown, either because the tty is being closed or because
 *      the tty is being changed to use a new line discpline
 *
 * void (*flush_buffer)(struct tty_struct *tty);
 *
 *      This function instructs the line discipline to clear its
 *      buffers of any input characters it may have queued to be
 *      delivered to the user mode process.
 *
 * int  (*chars_in_buffer)(struct tty_struct *tty);
 *
 *      This function returns the number of input characters the line
 *      iscpline may have queued up to be delivered to the user mode
 *      process.
 *
 * int  (*read)(struct tty_struct * tty, struct file * file,
 *              unsigned char * buf, unsigned int nr);
 *
 *      This function is called when the user requests to read from
 *      the tty.  The line discpline will return whatever characters
 *      it has buffered up for the user.  If this function is not
 *      defined, the user will receive an EIO error.
 *
 * int  (*write)(struct tty_struct * tty, struct file * file,
 *               const unsigned char * buf, unsigned int nr);
 *
 *      This function is called when the user requests to write to the
 *      tty.  The line discpline will deliver the characters to the
 *      low-level tty device for transmission, optionally performing
 *      some processing on the characters first.  If this function is
 *      not defined, the user will receive an EIO error.
 *
 * int  (*ioctl)(struct tty_struct * tty, struct file * file,
 *               unsigned int cmd, unsigned long arg);
 *
 *      This function is called when the user requests an ioctl which
 *      is not handled by the tty layer or the low-level tty driver.
 *      It is intended for ioctls which affect line discpline
 *      operation.  Not that the search order for ioctls is (1) tty
 *      layer, (2) tty low-level driver, (3) line discpline.  So a
 *      low-level driver can "grab" an ioctl request before the line
 *      discpline has a chance to see it.
 *
 * void (*set_termios)(struct tty_struct *tty, struct termios * old);
 *
 *      This function notifies the line discpline that a change has
 *      been made to the termios stucture.
 *
 * int  (*select)(struct tty_struct * tty, struct inode * inode,
 *                struct file * file, int sel_type,
 *                struct select_table_struct *wait);
 *
```

```
 *      This function is called when a user attempts to select on a
 *      tty device.  It is solely the responsibility of the line
 *      discipline to handle select requests.
 *
 * void (*receive_buf)(struct tty_struct *, const unsigned char *cp,
 *                     char *fp, int count);
 *
 *      This function is called by the low-level tty driver to send
 *      characters received by the hardware to the line discpline for
 *      processing.  <cp> is a pointer to the buffer of input
 *      character received by the device.  <fp> is a pointer to a
 *      pointer of flag bytes which indicate whether a character was
 *      received with a parity error, etc.
 *
 * int  (*receive_room)(struct tty_struct *);
 *
 *      This function is called by the low-level tty driver to
 *      determine how many characters the line discpline can accept.
 *      The low-level driver must not send more characters than was
 *      indicated by receive_room, or the line discpline may drop
 *      those characters.
 *
 * void (*write_wakeup)(struct tty_struct *);
 *
 *      This function is called by the low-level tty driver to signal
 *      that line discpline should try to send more characters to the
 *      low-level driver for transmission.  If the line discpline does
 *      not have any more data to send, it can just return.
 */
struct tty_ldisc {
        int     magic;
        int     num;
        int     flags;
        /*
         * The following routines are called from above.
         */
        int     (*open)(struct tty_struct *);
        void    (*close)(struct tty_struct *);
        void    (*flush_buffer)(struct tty_struct *tty);
        int     (*chars_in_buffer)(struct tty_struct *tty);
        int     (*read)(struct tty_struct * tty, struct file * file,
                        unsigned char * buf, unsigned int nr);
        int     (*write)(struct tty_struct * tty, struct file * file,
                         const unsigned char * buf, unsigned int nr);
        int     (*ioctl)(struct tty_struct * tty, struct file * file,
                         unsigned int cmd, unsigned long arg);
        void    (*set_termios)(struct tty_struct *tty, struct termios * old);
        int     (*select)(struct tty_struct * tty, struct inode * inode,
                          struct file * file, int sel_type,
                          struct select_table_struct *wait);

        /*
         * The following routines are called from below.
         */
        void    (*receive_buf)(struct tty_struct *, const unsigned char *cp,
                               char *fp, int count);
        int     (*receive_room)(struct tty_struct *);
        void    (*write_wakeup)(struct tty_struct *);
};
```

```
#define TTY_LDISC_MAGIC 0x5403

#define LDISC_FLAG_DEFINED      0x00000001


/*
 * This structure defines the interface between the low-level tty
 * driver and the tty routines.  The following routines can be
 * defined; unless noted otherwise, they are optional, and can be
 * filled in with a null pointer.
 *
 * int  (*open)(struct tty_struct * tty, struct file * filp);
 *
 *      This routine is called when a particular tty device is opened.
 *      This routine is mandatory; if this routine is not filled in,
 *      the attempted open will fail with ENODEV.
 *
 * void (*close)(struct tty_struct * tty, struct file * filp);
 *
 *      This routine is called when a particular tty device is closed.
 *
 * int (*write)(struct tty_struct * tty, int from_user,
 *               const unsigned char *buf, int count);
 *
 *      This routine is called by the kernel to write a series of
 *      characters to the tty device.  The characters may come from
 *      user space or kernel space.  This routine will return the
 *      number of characters actually accepted for writing.  This
 *      routine is mandatory.
 *
 * void (*put_char)(struct tty_struct *tty, unsigned char ch);
 *
 *      This routine is called by the kernel to write a single
 *      character to the tty device.  If the kernel uses this routine,
 *      it must call the flush_chars() routine (if defined) when it is
 *      done stuffing characters into the driver.  If there is no room
 *      in the queue, the character is ignored.
 *
 * void (*flush_chars)(struct tty_struct *tty);
 *
 *      This routine is called by the kernel after it has written a
 *      series of characters to the tty device using put_char().
 *
 * int  (*write_room)(struct tty_struct *tty);
 *
 *      This routine returns the numbers of characters the tty driver
 *      will accept for queuing to be written.  This number is subject
 *      to change as output buffers get emptied.
 *
 * int  (*ioctl)(struct tty_struct *tty, struct file * file,
 *          unsigned int cmd, unsigned long arg);
 *
 *      This routine allows the tty driver to implement
 *      device-specific ioctl's.  If the ioctl number passed in cmd
 *      is not recognized by the driver, it should return ENOIOCTLCMD.
 *
 * void (*set_termios)(struct tty_struct *tty, struct termios * old);
 *
 *      This routine allows the tty driver to be notified when
 *      device's termios settings have changed.  Note that a
```

```
 *      well-designed tty driver should be prepared to accept the case
 *      where old == NULL, and try to do something rational.
 *
 * void (*set_ldisc)(struct tty_struct *tty);
 *
 *      This routine allows the tty driver to be notified when the
 *      device’s termios settings have changed.
 *
 * void (*throttle)(struct tty_struct * tty);
 *
 *      This routine notifies the tty driver that input buffers for
 *      the line discipline are close to full, and it should somehow
 *      signal that no more characters should be sent to the tty.
 *
 * void (*unthrottle)(struct tty_struct * tty);
 *
 *      This routine notifies the tty drivers that it should signals
 *      that characters can now be sent to the tty without fear of
 *      overrunning the input buffers of the line disciplines.
 *
 * void (*stop)(struct tty_struct *tty);
 *
 *      This routine notifies the tty driver that it should stop
 *      outputting characters to the tty device.
 *
 * void (*start)(struct tty_struct *tty);
 *
 *      This routine notifies the tty driver that it resume sending
 *      characters to the tty device.
 *
 * void (*hangup)(struct tty_struct *tty);
 *
 *      This routine notifies the tty driver that it should hangup the
 *      tty device.
 *
 */
struct tty_driver {
        int     magic;          /* magic number for this structure */
        const char      *name;
        int     name_base;      /* offset of printed name */
        short   major;          /* major device number */
        short   minor_start;    /* start of minor device number*/
        short   num;            /* number of devices */
        short   type;           /* type of tty driver */
        short   subtype;        /* subtype of tty driver */
        struct termios init_termios; /* Initial termios */
        int     flags;          /* tty driver flags */
        int     *refcount;      /* for loadable tty drivers */
        struct tty_driver *other; /* only used for the PTY driver */

        /*
         * Pointer to the tty data structures
         */
        struct tty_struct **table;
        struct termios **termios;
        struct termios **termios_locked;

        /*
         * Interface routines from the upper tty layer to the tty
         * driver.
```

```
         */
        int  (*open)(struct tty_struct * tty, struct file * filp);
        void (*close)(struct tty_struct * tty, struct file * filp);
        int  (*write)(struct tty_struct * tty, int from_user,
                      const unsigned char *buf, int count);
        void (*put_char)(struct tty_struct *tty, unsigned char ch);
        void (*flush_chars)(struct tty_struct *tty);
        int  (*write_room)(struct tty_struct *tty);
        int  (*chars_in_buffer)(struct tty_struct *tty);
        int  (*ioctl)(struct tty_struct *tty, struct file * file,
                    unsigned int cmd, unsigned long arg);
        void (*set_termios)(struct tty_struct *tty, struct termios * old);
        void (*throttle)(struct tty_struct * tty);
        void (*unthrottle)(struct tty_struct * tty);
        void (*stop)(struct tty_struct *tty);
        void (*start)(struct tty_struct *tty);
        void (*hangup)(struct tty_struct *tty);
        void (*flush_buffer)(struct tty_struct *tty);
        void (*set_ldisc)(struct tty_struct *tty);

        /*
         * linked list pointers
         */
        struct tty_driver *next;
        struct tty_driver *prev;
};

/* tty driver magic number */
#define TTY_DRIVER_MAGIC                0x5402

/*
 * tty driver flags
 *
 * TTY_DRIVER_RESET_TERMIOS --- requests the tty layer to reset the
 *      termios setting when the last process has closed the device.
 *      Used for PTY's, in particular.
 *
 * TTY_DRIVER_REAL_RAW --- if set, indicates that the driver will
 *      guarantee never not to set any special character handling
 *      flags if ((IGNBRK || (!BRKINT && !PARMRK)) && (IGNPAR ||
 *      !INPCK)).  That is, if there is no reason for the driver to
 *      send notifications of parity and break characters up to the
 *      line driver, it won't do so.  This allows the line driver to
 *      optimize for this case if this flag is set.  (Note that there
 *      is also a promise, if the above case is true, not to signal
 *      overruns, either.)
 */
#define TTY_DRIVER_INSTALLED            0x0001
#define TTY_DRIVER_RESET_TERMIOS        0x0002
#define TTY_DRIVER_REAL_RAW             0x0004

/* tty driver types */
#define TTY_DRIVER_TYPE_SYSTEM          0x0001
#define TTY_DRIVER_TYPE_CONSOLE         0x0002
#define TTY_DRIVER_TYPE_SERIAL          0x0003
#define TTY_DRIVER_TYPE_PTY             0x0004
#define TTY_DRIVER_TYPE_SCC             0x0005  /* scc driver */

/* system subtypes (magic, used by tty_io.c) */
#define SYSTEM_TYPE_TTY                 0x0001
```

```
#define SYSTEM_TYPE_CONSOLE            0x0002

/* pty subtypes (magic, used by tty_io.c) */
#define PTY_TYPE_MASTER                0x0001
#define PTY_TYPE_SLAVE                 0x0002

#endif /* _LINUX_TTY_LDISC_H */
```