```
#ifndef __LINUX_USB_H
#define __LINUX_USB_H

#include <linux/mod_devicetable.h>
#include <linux/usb/ch9.h>

#define USB_MAJOR                       180
#define USB_DEVICE_MAJOR                189


#ifdef __KERNEL__

#include <linux/errno.h>        /* for -ENODEV */
#include <linux/delay.h>  /* for mdelay() */
#include <linux/interrupt.h>            /* for in_interrupt() */
#include <linux/list.h>                 /* for struct list_head */
#include <linux/kref.h>                 /* for struct kref */
#include <linux/device.h> /* for struct device */
#include <linux/fs.h>                   /* for struct file_operations */
#include <linux/completion.h>       /* for struct completion */
#include <linux/sched.h>  /* for current && schedule_timeout */
#include <linux/mutex.h> /* for struct mutex */

struct usb_device;
struct usb_driver;
struct wusb_dev;

/*-------------------------------------------------------------------------*/

/*
 * Host-side wrappers for standard USB descriptors ... these are parsed
 * from the data provided by devices.  Parsing turns them from a flat
 * sequence of descriptors into a hierarchy:
 *
 *  - devices have one (usually) or more configs;
 *  - configs have one (often) or more interfaces;
 *  - interfaces have one (usually) or more settings;
 *  - each interface setting has zero or (usually) more endpoints.
 *
 * And there might be other descriptors mixed in with those.
 *
 * Devices may also have class-specific or vendor-specific descriptors.
 */

struct ep_device;

/**
 * struct usb_host_endpoint - host-side endpoint descriptor and queue
 * @desc: descriptor for this endpoint, wMaxPacketSize in native byteorder
 * @urb_list: urbs queued to this endpoint; maintained by usbcore
 * @hcpriv: for use by HCD; typically holds hardware dma queue head (QH)
 *          with one or more transfer descriptors (TDs) per urb
 * @ep_dev: ep_device for sysfs info
 * @extra: descriptors following this endpoint in the configuration
 * @extralen: how many bytes of "extra" are valid
 * @enabled: URBs may be submitted to this endpoint
 *
 * USB requests are always queued to a given endpoint, identified by a
```

```
 * descriptor within an active interface in a given USB configuration.
 */
struct usb_host_endpoint {
	struct usb_endpoint_descriptor	desc;
	struct list_head	urb_list;
	void	*hcpriv;
	struct ep_device	*ep_dev;	/* For sysfs info */

	unsigned char *extra;   /* Extra descriptors */
	int extralen;
	int enabled;
};

/* host-side wrapper for one interface setting's parsed descriptors */
struct usb_host_interface {
	struct usb_interface_descriptor	desc;

	/* array of desc.bNumEndpoint endpoints associated with this
	 * interface setting.  these will be in no particular order.
	 */
	struct usb_host_endpoint *endpoint;

	char *string;		/* iInterface string, if present */
	unsigned char *extra;   /* Extra descriptors */
	int extralen;
};

enum usb_interface_condition {
	USB_INTERFACE_UNBOUND = 0,
	USB_INTERFACE_BINDING,
	USB_INTERFACE_BOUND,
	USB_INTERFACE_UNBINDING,
};

/**
 * struct usb_interface - what usb device drivers talk to
 * @altsetting: array of interface structures, one for each alternate
 *	setting that may be selected.  Each one includes a set of
 *	endpoint configurations.  They will be in no particular order.
 * @cur_altsetting: the current altsetting.
 * @num_altsetting: number of altsettings defined.
 * @intf_assoc: interface association descriptor
 * @minor: the minor number assigned to this interface, if this
 *	interface is bound to a driver that uses the USB major number.
 *	If this interface does not use the USB major, this field should
 *	be unused.  The driver should set this value in the probe()
 *	function of the driver, after it has been assigned a minor
 *	number from the USB core by calling usb_register_dev().
 * @condition: binding state of the interface: not bound, binding
 *	(in probe()), bound to a driver, or unbinding (in disconnect())
 * @is_active: flag set when the interface is bound and not suspended.
 * @sysfs_files_created: sysfs attributes exist
 * @unregistering: flag set when the interface is being unregistered
 * @needs_remote_wakeup: flag set when the driver requires remote-wakeup
 *	capability during autosuspend.
 * @needs_altsetting0: flag set when a set-interface request for altsetting 0
 *	has been deferred.
 * @needs_binding: flag set when the driver should be re-probed or unbound
```

```
 *          following a reset or suspend operation it doesn't support.
 * @dev: driver model's view of this device
 * @usb_dev: if an interface is bound to the USB major, this will point
 *          to the sysfs representation for that device.
 * @pm_usage_cnt: PM usage counter for this interface; autosuspend is not
 *          allowed unless the counter is 0.
 *
 * USB device drivers attach to interfaces on a physical device.  Each
 * interface encapsulates a single high level function, such as feeding
 * an audio stream to a speaker or reporting a change in a volume control.
 * Many USB devices only have one interface.  The protocol used to talk to
 * an interface's endpoints can be defined in a usb "class" specification,
 * or by a product's vendor.  The (default) control endpoint is part of
 * every interface, but is never listed among the interface's descriptors.
 *
 * The driver that is bound to the interface can use standard driver model
 * calls such as dev_get_drvdata() on the dev member of this structure.
 *
 * Each interface may have alternate settings.  The initial configuration
 * of a device sets altsetting 0, but the device driver can change
 * that setting using usb_set_interface().  Alternate settings are often
 * used to control the use of periodic endpoints, such as by having
 * different endpoints use different amounts of reserved USB bandwidth.
 * All standards-conformant USB devices that use isochronous endpoints
 * will use them in non-default settings.
 *
 * The USB specification says that alternate setting numbers must run from
 * 0 to one less than the total number of alternate settings.  But some
 * devices manage to mess this up, and the structures aren't necessarily
 * stored in numerical order anyhow.  Use usb_altnum_to_altsetting() to
 * look up an alternate setting in the altsetting array based on its number.
 */
struct usb_interface {
        /* array of alternate settings for this interface,
         * stored in no particular order */
        struct usb_host_interface *altsetting;

        struct usb_host_interface *cur_altsetting;     /* the currently
                                         * active alternate setting */
        unsigned num_altsetting;  /* number of alternate settings */

        /* If there is an interface association descriptor then it will list
         * the associated interfaces */
        struct usb_interface_assoc_descriptor *intf_assoc;

        int minor;                      /* minor number this interface is
                                         * bound to */
        enum usb_interface_condition condition;         /* state of binding */
        unsigned is_active:1;           /* the interface is not suspended */
        unsigned sysfs_files_created:1;     /* the sysfs attributes exist */
        unsigned unregistering:1;  /* unregistration is in progress */
        unsigned needs_remote_wakeup:1;/* driver requires remote wakeup */
        unsigned needs_altsetting0:1;       /* switch to altsetting 0 is pending */
        unsigned needs_binding:1;           /* needs delayed unbind/rebind */

        struct device dev;              /* interface specific device info */
        struct device *usb_dev;
        int pm_usage_cnt;               /* usage counter for autosuspend */
```

```
};
#define  to_usb_interface(d) container_of(d, struct usb_interface, dev)
#define  interface_to_usbdev(intf) \
        container_of(intf->dev.parent, struct usb_device, dev)

static inline void *usb_get_intfdata(struct usb_interface *intf)
{
        return dev_get_drvdata(&intf->dev);
}

static inline void usb_set_intfdata(struct usb_interface *intf, void *data)
{
        dev_set_drvdata(&intf->dev, data);
}

struct usb_interface *usb_get_intf(struct usb_interface *intf);
void usb_put_intf(struct usb_interface *intf);

/* this maximum is arbitrary */
#define USB_MAXINTERFACES 32
#define USB_MAXIADS             USB_MAXINTERFACES/2

/**
 * struct usb_interface_cache - long-term representation of a device interface
 * @num_altsetting: number of altsettings defined.
 * @ref: reference counter.
 * @altsetting: variable-length array of interface structures, one for
 *      each alternate setting that may be selected.  Each one includes a
 *      set of endpoint configurations.  They will be in no particular order.
 *
 * These structures persist for the lifetime of a usb_device, unlike
 * struct usb_interface (which persists only as long as its configuration
 * is installed).  The altsetting arrays can be accessed through these
 * structures at any time, permitting comparison of configurations and
 * providing support for the /proc/bus/usb/devices pseudo-file.
 */
struct usb_interface_cache {
        unsigned num_altsetting;  /* number of alternate settings */
        struct kref ref;                /* reference counter */

        /* variable-length array of alternate settings for this interface,
         * stored in no particular order */
        struct usb_host_interface altsetting[0];
};
#define  ref_to_usb_interface_cache(r) \
                container_of(r, struct usb_interface_cache, ref)
#define  altsetting_to_usb_interface_cache(a) \
                container_of(a, struct usb_interface_cache, altsetting[0])

/**
 * struct usb_host_config - representation of a device's configuration
 * @desc: the device's configuration descriptor.
 * @string: pointer to the cached version of the iConfiguration string, if
 *      present for this configuration.
 * @intf_assoc: list of any interface association descriptors in this config
 * @interface: array of pointers to usb_interface structures, one for each
 *      interface in the configuration.  The number of interfaces is stored
 *      in desc.bNumInterfaces.  These pointers are valid only while the
```

```
 *          the configuration is active.
 * @intf_cache: array of pointers to usb_interface_cache structures, one
 *          for each interface in the configuration.  These structures exist
 *          for the entire life of the device.
 * @extra: pointer to buffer containing all extra descriptors associated
 *          with this configuration (those preceding the first interface
 *          descriptor).
 * @extralen: length of the extra descriptors buffer.
 *
 * USB devices may have multiple configurations, but only one can be active
 * at any time.  Each encapsulates a different operational environment;
 * for example, a dual-speed device would have separate configurations for
 * full-speed and high-speed operation.  The number of configurations
 * available is stored in the device descriptor as bNumConfigurations.
 *
 * A configuration can contain multiple interfaces.  Each corresponds to
 * a different function of the USB device, and all are available whenever
 * the configuration is active.  The USB standard says that interfaces
 * are supposed to be numbered from 0 to desc.bNumInterfaces-1, but a lot
 * of devices get this wrong.  In addition, the interface array is not
 * guaranteed to be sorted in numerical order.  Use usb_ifnum_to_if() to
 * look up an interface entry based on its number.
 *
 * Device drivers should not attempt to activate configurations.  The choice
 * of which configuration to install is a policy decision based on such
 * considerations as available power, functionality provided, and the user's
 * desires (expressed through userspace tools).  However, drivers can call
 * usb_reset_configuration() to reinitialize the current configuration and
 * all its interfaces.
 */
struct usb_host_config {
        struct usb_config_descriptor          desc;

        char *string;                   /* iConfiguration string, if present */

        /* List of any Interface Association Descriptors in this
         * configuration. */
        struct usb_interface_assoc_descriptor *intf_assoc[USB_MAXIADS];

        /* the interfaces associated with this configuration,
         * stored in no particular order */
        struct usb_interface *interface[USB_MAXINTERFACES];

        /* Interface information available even when this is not the
         * active configuration */
        struct usb_interface_cache *intf_cache[USB_MAXINTERFACES];

        unsigned char *extra;   /* Extra descriptors */
        int extralen;
};

int __usb_get_extra_descriptor(char *buffer, unsigned size,
        unsigned char type, void **ptr);
#define usb_get_extra_descriptor(ifpoint, type, ptr) \
                                        __usb_get_extra_descriptor((ifpoint)->extra, \
                                        (ifpoint)->extralen, \
                                        type, (void **)ptr)
```

```
/* ------------------------------------------------------------------------- */

/* USB device number allocation bitmap */
struct usb_devmap {
        unsigned long devicemap[128 / (8*sizeof(unsigned long))];
};

/*
 * Allocated per bus (tree of devices) we have:
 */
struct usb_bus {
        struct device *controller;  /* host/master side hardware */
        int busnum;                 /* Bus number (in order of reg) */
        const char *bus_name;       /* stable id (PCI slot_name etc) */
        u8 uses_dma;                /* Does the host controller use DMA? */
        u8 otg_port;                /* 0, or number of OTG/HNP port */
        unsigned is_b_host:1;       /* true during some HNP roleswitches */
        unsigned b_hnp_enable:1;/* OTG: did A-Host enable HNP? */

        int devnum_next;            /* Next open device number in
                                     * round-robin allocation */

        struct usb_devmap devmap;       /* device address allocation map */
        struct usb_device *root_hub;    /* Root hub */
        struct list_head bus_list;  /* list of busses */

        int bandwidth_allocated;  /* on this bus: how much of the time
                                   * reserved for periodic (intr/iso)
                                   * requests is used, on average?
                                   * Units: microseconds/frame.
                                   * Limits: Full/low speed reserve 90%,
                                   * while high speed reserves 80%.
                                   */
        int bandwidth_int_reqs;         /* number of Interrupt requests */
        int bandwidth_isoc_reqs;  /* number of Isoc. requests */

#ifdef CONFIG_USB_DEVICEFS
        struct dentry *usbfs_dentry;    /* usbfs dentry entry for the bus */
#endif
        struct device *dev;             /* device for this bus */

#if defined(CONFIG_USB_MON)
        struct mon_bus *mon_bus;        /* non-null when associated */
        int monitored;                  /* non-zero when monitored */
#endif
};

/* ------------------------------------------------------------------------- */

/* This is arbitrary.
 * From USB 2.0 spec Table 11-13, offset 7, a hub can
 * have up to 255 ports. The most yet reported is 10.
 *
 * Current Wireless USB host hardware (Intel i1480 for example) allows
 * up to 22 devices to connect. Upcoming hardware might raise that
 * limit. Because the arrays need to add a bit for hub status data, we
 * do 31, so plus one evens out to four bytes.
 */
```

```
#define USB_MAXCHILDREN		(31)

struct usb_tt;

/**
 * struct usb_device - kernel's representation of a USB device
 * @devnum: device number; address on a USB bus
 * @devpath: device ID string for use in messages (e.g., /port/...)
 * @state: device state: configured, not attached, etc.
 * @speed: device speed: high/full/low (or error)
 * @tt: Transaction Translator info; used with low/full speed dev, highspeed hub
 * @ttport: device port on that tt hub
 * @toggle: one bit for each endpoint, with ([0] = IN, [1] = OUT) endpoints
 * @parent: our hub, unless we're the root
 * @bus: bus we're part of
 * @ep0: endpoint 0 data (default control pipe)
 * @dev: generic device interface
 * @descriptor: USB device descriptor
 * @config: all of the device's configs
 * @actconfig: the active configuration
 * @ep_in: array of IN endpoints
 * @ep_out: array of OUT endpoints
 * @rawdescriptors: raw descriptors for each config
 * @bus_mA: Current available from the bus
 * @portnum: parent port number (origin 1)
 * @level: number of USB hub ancestors
 * @can_submit: URBs may be submitted
 * @discon_suspended: disconnected while suspended
 * @persist_enabled:  USB_PERSIST enabled for this device
 * @have_langid: whether string_langid is valid
 * @authorized: policy has said we can use it;
 *	(user space) policy determines if we authorize this device to be
 *	used or not. By default, wired USB devices are authorized.
 *	WUSB devices are not, until we authorize them from user space.
 *	FIXME -- complete doc
 * @authenticated: Crypto authentication passed
 * @wusb: device is Wireless USB
 * @string_langid: language ID for strings
 * @product: iProduct string, if present (static)
 * @manufacturer: iManufacturer string, if present (static)
 * @serial: iSerialNumber string, if present (static)
 * @filelist: usbfs files that are open to this device
 * @usb_classdev: USB class device that was created for usbfs device
 *	access from userspace
 * @usbfs_dentry: usbfs dentry entry for the device
 * @maxchild: number of ports if hub
 * @children: child devices - USB devices that are attached to this hub
 * @pm_usage_cnt: usage counter for autosuspend
 * @quirks: quirks of the whole device
 * @urbnum: number of URBs submitted for the whole device
 * @active_duration: total time device is not suspended
 * @autosuspend: for delayed autosuspends
 * @pm_mutex: protects PM operations
 * @last_busy: time of last use
 * @autosuspend_delay: in jiffies
 * @connect_time: time device was first connected
 * @auto_pm: autosuspend/resume in progress
 * @do_remote_wakeup:  remote wakeup should be enabled
```

```
 * @reset_resume: needs reset instead of resume
 * @autosuspend_disabled: autosuspend disabled by the user
 * @autoresume_disabled: autoresume disabled by the user
 * @skip_sys_resume: skip the next system resume
 *
 * Notes:
 * Usbcore drivers should not set usbdev->state directly.  Instead use
 * usb_set_device_state().
 */
struct usb_device {
	int		devnum;
	char		devpath [16];
	enum usb_device_state	state;
	enum usb_device_speed	speed;

	struct usb_tt	*tt;
	int		ttport;

	unsigned int toggle[2];

	struct usb_device *parent;
	struct usb_bus *bus;
	struct usb_host_endpoint ep0;

	struct device dev;

	struct usb_device_descriptor descriptor;
	struct usb_host_config *config;

	struct usb_host_config *actconfig;
	struct usb_host_endpoint *ep_in[16];
	struct usb_host_endpoint *ep_out[16];

	char **rawdescriptors;

	unsigned short bus_mA;
	u8 portnum;
	u8 level;

	unsigned can_submit:1;
	unsigned discon_suspended:1;
	unsigned persist_enabled:1;
	unsigned have_langid:1;
	unsigned authorized:1;
	unsigned authenticated:1;
	unsigned wusb:1;
	int string_langid;

	/* static strings from the device */
	char *product;
	char *manufacturer;
	char *serial;

	struct list_head filelist;
#ifdef CONFIG_USB_DEVICE_CLASS
	struct device *usb_classdev;
#endif
#ifdef CONFIG_USB_DEVICEFS
```

```
	struct dentry *usbfs_dentry;
#endif

	int maxchild;
	struct usb_device *children[USB_MAXCHILDREN];

	int pm_usage_cnt;
	u32 quirks;
	atomic_t urbnum;

	unsigned long active_duration;

#ifdef CONFIG_PM
	struct delayed_work autosuspend;
	struct mutex pm_mutex;

	unsigned long last_busy;
	int autosuspend_delay;
	unsigned long connect_time;

	unsigned auto_pm:1;
	unsigned do_remote_wakeup:1;
	unsigned reset_resume:1;
	unsigned autosuspend_disabled:1;
	unsigned autoresume_disabled:1;
	unsigned skip_sys_resume:1;
#endif
	struct wusb_dev *wusb_dev;
};
#define  to_usb_device(d) container_of(d, struct usb_device, dev)

extern struct usb_device *usb_get_dev(struct usb_device *dev);
extern void usb_put_dev(struct usb_device *dev);

/* USB device locking */
#define usb_lock_device(udev)		down(&(udev)->dev.sem)
#define usb_unlock_device(udev)		up(&(udev)->dev.sem)
#define usb_trylock_device(udev)	down_trylock(&(udev)->dev.sem)
extern int usb_lock_device_for_reset(struct usb_device *udev,
				     const struct usb_interface *iface);

/* USB port reset for device reinitialization */
extern int usb_reset_device(struct usb_device *dev);

extern struct usb_device *usb_find_device(u16 vendor_id, u16 product_id);

/* USB autosuspend and autoresume */
#ifdef CONFIG_USB_SUSPEND
extern int usb_autopm_set_interface(struct usb_interface *intf);
extern int usb_autopm_get_interface(struct usb_interface *intf);
extern void usb_autopm_put_interface(struct usb_interface *intf);

static inline void usb_autopm_enable(struct usb_interface *intf)
{
	intf->pm_usage_cnt = 0;
	usb_autopm_set_interface(intf);
}
```

```
static inline void usb_autopm_disable(struct usb_interface *intf)
{
	intf->pm_usage_cnt = 1;
	usb_autopm_set_interface(intf);
}

static inline void usb_mark_last_busy(struct usb_device *udev)
{
	udev->last_busy = jiffies;
}

#else

static inline int usb_autopm_set_interface(struct usb_interface *intf)
{ return 0; }

static inline int usb_autopm_get_interface(struct usb_interface *intf)
{ return 0; }

static inline void usb_autopm_put_interface(struct usb_interface *intf)
{ }
static inline void usb_autopm_enable(struct usb_interface *intf)
{ }
static inline void usb_autopm_disable(struct usb_interface *intf)
{ }
static inline void usb_mark_last_busy(struct usb_device *udev)
{ }
#endif

/*-------------------------------------------------------------------------*/

/* for drivers using iso endpoints */
extern int usb_get_current_frame_number(struct usb_device *usb_dev);

/* used these for multi-interface device registration */
extern int usb_driver_claim_interface(struct usb_driver *driver,
			struct usb_interface *iface, void *priv);

/**
 * usb_interface_claimed - returns true iff an interface is claimed
 * @iface: the interface being checked
 *
 * Returns true (nonzero) iff the interface is claimed, else false (zero).
 * Callers must own the driver model's usb bus readlock.  So driver
 * probe() entries don't need extra locking, but other call contexts
 * may need to explicitly claim that lock.
 *
 */
static inline int usb_interface_claimed(struct usb_interface *iface)
{
	return (iface->dev.driver != NULL);
}

extern void usb_driver_release_interface(struct usb_driver *driver,
			struct usb_interface *iface);
const struct usb_device_id *usb_match_id(struct usb_interface *interface,
					 const struct usb_device_id *id);
extern int usb_match_one_id(struct usb_interface *interface,
```

```
                                const struct usb_device_id *id);

extern struct usb_interface *usb_find_interface(struct usb_driver *drv,
                int minor);
extern struct usb_interface *usb_ifnum_to_if(const struct usb_device *dev,
                unsigned ifnum);
extern struct usb_host_interface *usb_altnum_to_altsetting(
                const struct usb_interface *intf, unsigned int altnum);


/**
 * usb_make_path - returns stable device path in the usb tree
 * @dev: the device whose path is being constructed
 * @buf: where to put the string
 * @size: how big is "buf"?
 *
 * Returns length of the string (> 0) or negative if size was too small.
 *
 * This identifier is intended to be "stable", reflecting physical paths in
 * hardware such as physical bus addresses for host controllers or ports on
 * USB hubs.  That makes it stay the same until systems are physically
 * reconfigured, by re-cabling a tree of USB devices or by moving USB host
 * controllers.  Adding and removing devices, including virtual root hubs
 * in host controller driver modules, does not change these path identifers;
 * neither does rebooting or re-enumerating.  These are more useful identifiers
 * than changeable ("unstable") ones like bus numbers or device addresses.
 *
 * With a partial exception for devices connected to USB 2.0 root hubs, these
 * identifiers are also predictable.  So long as the device tree isn't changed,
 * plugging any USB device into a given hub port always gives it the same path.
 * Because of the use of "companion" controllers, devices connected to ports on
 * USB 2.0 root hubs (EHCI host controllers) will get one path ID if they are
 * high speed, and a different one if they are full or low speed.
 */
static inline int usb_make_path(struct usb_device *dev, char *buf, size_t size)
{
        int actual;
        actual = snprintf(buf, size, "usb-%s-%s", dev->bus->bus_name,
                        dev->devpath);
        return (actual >= (int)size) ? -1 : actual;
}

/*-------------------------------------------------------------------------*/

/**
 * usb_endpoint_num - get the endpoint's number
 * @epd: endpoint to be checked
 *
 * Returns @epd's number: 0 to 15.
 */
static inline int usb_endpoint_num(const struct usb_endpoint_descriptor *epd)
{
        return epd->bEndpointAddress & USB_ENDPOINT_NUMBER_MASK;
}

/**
 * usb_endpoint_type - get the endpoint's transfer type
 * @epd: endpoint to be checked
```

```
 *
 * Returns one of USB_ENDPOINT_XFER_{CONTROL, ISOC, BULK, INT} according
 * to @epd's transfer type.
 */
static inline int usb_endpoint_type(const struct usb_endpoint_descriptor *epd)
{
	return epd->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK;
}

/**
 * usb_endpoint_dir_in - check if the endpoint has IN direction
 * @epd: endpoint to be checked
 *
 * Returns true if the endpoint is of type IN, otherwise it returns false.
 */
static inline int usb_endpoint_dir_in(const struct usb_endpoint_descriptor *epd)
{
	return ((epd->bEndpointAddress & USB_ENDPOINT_DIR_MASK) == USB_DIR_IN);
}

/**
 * usb_endpoint_dir_out - check if the endpoint has OUT direction
 * @epd: endpoint to be checked
 *
 * Returns true if the endpoint is of type OUT, otherwise it returns false.
 */
static inline int usb_endpoint_dir_out(
				const struct usb_endpoint_descriptor *epd)
{
	return ((epd->bEndpointAddress & USB_ENDPOINT_DIR_MASK) == USB_DIR_OUT);
}

/**
 * usb_endpoint_xfer_bulk - check if the endpoint has bulk transfer type
 * @epd: endpoint to be checked
 *
 * Returns true if the endpoint is of type bulk, otherwise it returns false.
 */
static inline int usb_endpoint_xfer_bulk(
				const struct usb_endpoint_descriptor *epd)
{
	return ((epd->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK) ==
		USB_ENDPOINT_XFER_BULK);
}

/**
 * usb_endpoint_xfer_control - check if the endpoint has control transfer type
 * @epd: endpoint to be checked
 *
 * Returns true if the endpoint is of type control, otherwise it returns false.
 */
static inline int usb_endpoint_xfer_control(
				const struct usb_endpoint_descriptor *epd)
{
	return ((epd->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK) ==
		USB_ENDPOINT_XFER_CONTROL);
}
```

```
/**
 * usb_endpoint_xfer_int - check if the endpoint has interrupt transfer type
 * @epd: endpoint to be checked
 *
 * Returns true if the endpoint is of type interrupt, otherwise it returns
 * false.
 */
static inline int usb_endpoint_xfer_int(
                                        const struct usb_endpoint_descriptor *epd)
{
        return ((epd->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK) ==
                USB_ENDPOINT_XFER_INT);
}

/**
 * usb_endpoint_xfer_isoc - check if the endpoint has isochronous transfer type
 * @epd: endpoint to be checked
 *
 * Returns true if the endpoint is of type isochronous, otherwise it returns
 * false.
 */
static inline int usb_endpoint_xfer_isoc(
                                        const struct usb_endpoint_descriptor *epd)
{
        return ((epd->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK) ==
                USB_ENDPOINT_XFER_ISOC);
}

/**
 * usb_endpoint_is_bulk_in - check if the endpoint is bulk IN
 * @epd: endpoint to be checked
 *
 * Returns true if the endpoint has bulk transfer type and IN direction,
 * otherwise it returns false.
 */
static inline int usb_endpoint_is_bulk_in(
                                        const struct usb_endpoint_descriptor *epd)
{
        return (usb_endpoint_xfer_bulk(epd) && usb_endpoint_dir_in(epd));
}

/**
 * usb_endpoint_is_bulk_out - check if the endpoint is bulk OUT
 * @epd: endpoint to be checked
 *
 * Returns true if the endpoint has bulk transfer type and OUT direction,
 * otherwise it returns false.
 */
static inline int usb_endpoint_is_bulk_out(
                                        const struct usb_endpoint_descriptor *epd)
{
        return (usb_endpoint_xfer_bulk(epd) && usb_endpoint_dir_out(epd));
}

/**
 * usb_endpoint_is_int_in - check if the endpoint is interrupt IN
 * @epd: endpoint to be checked
 *
```

```
 * Returns true if the endpoint has interrupt transfer type and IN direction,
 * otherwise it returns false.
 */
static inline int usb_endpoint_is_int_in(
                                        const struct usb_endpoint_descriptor *epd)
{
        return (usb_endpoint_xfer_int(epd) && usb_endpoint_dir_in(epd));
}

/**
 * usb_endpoint_is_int_out - check if the endpoint is interrupt OUT
 * @epd: endpoint to be checked
 *
 * Returns true if the endpoint has interrupt transfer type and OUT direction,
 * otherwise it returns false.
 */
static inline int usb_endpoint_is_int_out(
                                        const struct usb_endpoint_descriptor *epd)
{
        return (usb_endpoint_xfer_int(epd) && usb_endpoint_dir_out(epd));
}

/**
 * usb_endpoint_is_isoc_in - check if the endpoint is isochronous IN
 * @epd: endpoint to be checked
 *
 * Returns true if the endpoint has isochronous transfer type and IN direction,
 * otherwise it returns false.
 */
static inline int usb_endpoint_is_isoc_in(
                                        const struct usb_endpoint_descriptor *epd)
{
        return (usb_endpoint_xfer_isoc(epd) && usb_endpoint_dir_in(epd));
}

/**
 * usb_endpoint_is_isoc_out - check if the endpoint is isochronous OUT
 * @epd: endpoint to be checked
 *
 * Returns true if the endpoint has isochronous transfer type and OUT direction,
 * otherwise it returns false.
 */
static inline int usb_endpoint_is_isoc_out(
                                        const struct usb_endpoint_descriptor *epd)
{
        return (usb_endpoint_xfer_isoc(epd) && usb_endpoint_dir_out(epd));
}

/*-------------------------------------------------------------------------*/

#define USB_DEVICE_ID_MATCH_DEVICE \
                (USB_DEVICE_ID_MATCH_VENDOR | USB_DEVICE_ID_MATCH_PRODUCT)
#define USB_DEVICE_ID_MATCH_DEV_RANGE \
                (USB_DEVICE_ID_MATCH_DEV_LO | USB_DEVICE_ID_MATCH_DEV_HI)
#define USB_DEVICE_ID_MATCH_DEVICE_AND_VERSION \
                (USB_DEVICE_ID_MATCH_DEVICE | USB_DEVICE_ID_MATCH_DEV_RANGE)
#define USB_DEVICE_ID_MATCH_DEV_INFO \
                (USB_DEVICE_ID_MATCH_DEV_CLASS | \
```

```
                USB_DEVICE_ID_MATCH_DEV_SUBCLASS | \
                USB_DEVICE_ID_MATCH_DEV_PROTOCOL)
#define USB_DEVICE_ID_MATCH_INT_INFO \
                (USB_DEVICE_ID_MATCH_INT_CLASS | \
                USB_DEVICE_ID_MATCH_INT_SUBCLASS | \
                USB_DEVICE_ID_MATCH_INT_PROTOCOL)

/**
 * USB_DEVICE - macro used to describe a specific usb device
 * @vend: the 16 bit USB Vendor ID
 * @prod: the 16 bit USB Product ID
 *
 * This macro is used to create a struct usb_device_id that matches a
 * specific device.
 */
#define USB_DEVICE(vend,prod) \
        .match_flags = USB_DEVICE_ID_MATCH_DEVICE, \
        .idVendor = (vend), \
        .idProduct = (prod)
/**
 * USB_DEVICE_VER - describe a specific usb device with a version range
 * @vend: the 16 bit USB Vendor ID
 * @prod: the 16 bit USB Product ID
 * @lo: the bcdDevice_lo value
 * @hi: the bcdDevice_hi value
 *
 * This macro is used to create a struct usb_device_id that matches a
 * specific device, with a version range.
 */
#define USB_DEVICE_VER(vend, prod, lo, hi) \
        .match_flags = USB_DEVICE_ID_MATCH_DEVICE_AND_VERSION, \
        .idVendor = (vend), \
        .idProduct = (prod), \
        .bcdDevice_lo = (lo), \
        .bcdDevice_hi = (hi)

/**
 * USB_DEVICE_INTERFACE_PROTOCOL - describe a usb device with a specific interface protocol
 * @vend: the 16 bit USB Vendor ID
 * @prod: the 16 bit USB Product ID
 * @pr: bInterfaceProtocol value
 *
 * This macro is used to create a struct usb_device_id that matches a
 * specific interface protocol of devices.
 */
#define USB_DEVICE_INTERFACE_PROTOCOL(vend, prod, pr) \
        .match_flags = USB_DEVICE_ID_MATCH_DEVICE | \
                        USB_DEVICE_ID_MATCH_INT_PROTOCOL, \
        .idVendor = (vend), \
        .idProduct = (prod), \
        .bInterfaceProtocol = (pr)

/**
 * USB_DEVICE_INFO - macro used to describe a class of usb devices
 * @cl: bDeviceClass value
 * @sc: bDeviceSubClass value
 * @pr: bDeviceProtocol value
 *
```

```
 * This macro is used to create a struct usb_device_id that matches a
 * specific class of devices.
 */
#define USB_DEVICE_INFO(cl, sc, pr) \
		.match_flags = USB_DEVICE_ID_MATCH_DEV_INFO, \
		.bDeviceClass = (cl), \
		.bDeviceSubClass = (sc), \
		.bDeviceProtocol = (pr)

/**
 * USB_INTERFACE_INFO - macro used to describe a class of usb interfaces
 * @cl: bInterfaceClass value
 * @sc: bInterfaceSubClass value
 * @pr: bInterfaceProtocol value
 *
 * This macro is used to create a struct usb_device_id that matches a
 * specific class of interfaces.
 */
#define USB_INTERFACE_INFO(cl, sc, pr) \
		.match_flags = USB_DEVICE_ID_MATCH_INT_INFO, \
		.bInterfaceClass = (cl), \
		.bInterfaceSubClass = (sc), \
		.bInterfaceProtocol = (pr)

/**
 * USB_DEVICE_AND_INTERFACE_INFO - describe a specific usb device with a class of usb interfaces
 * @vend: the 16 bit USB Vendor ID
 * @prod: the 16 bit USB Product ID
 * @cl: bInterfaceClass value
 * @sc: bInterfaceSubClass value
 * @pr: bInterfaceProtocol value
 *
 * This macro is used to create a struct usb_device_id that matches a
 * specific device with a specific class of interfaces.
 *
 * This is especially useful when explicitly matching devices that have
 * vendor specific bDeviceClass values, but standards-compliant interfaces.
 */
#define USB_DEVICE_AND_INTERFACE_INFO(vend, prod, cl, sc, pr) \
		.match_flags = USB_DEVICE_ID_MATCH_INT_INFO \
			| USB_DEVICE_ID_MATCH_DEVICE, \
		.idVendor = (vend), \
		.idProduct = (prod), \
		.bInterfaceClass = (cl), \
		.bInterfaceSubClass = (sc), \
		.bInterfaceProtocol = (pr)

/* ----------------------------------------------------------------------- */

/* Stuff for dynamic usb ids */
struct usb_dynids {
		spinlock_t lock;
		struct list_head list;
};

struct usb_dynid {
		struct list_head node;
		struct usb_device_id id;
```

```
};

extern ssize_t usb_store_new_id(struct usb_dynids *dynids,
                                struct device_driver *driver,
                                const char *buf, size_t count);

/**
 * struct usbdrv_wrap - wrapper for driver-model structure
 * @driver: The driver-model core driver structure.
 * @for_devices: Non-zero for device drivers, 0 for interface drivers.
 */
struct usbdrv_wrap {
        struct device_driver driver;
        int for_devices;
};

/**
 * struct usb_driver - identifies USB interface driver to usbcore
 * @name: The driver name should be unique among USB drivers,
 *      and should normally be the same as the module name.
 * @probe: Called to see if the driver is willing to manage a particular
 *      interface on a device.  If it is, probe returns zero and uses
 *      usb_set_intfdata() to associate driver-specific data with the
 *      interface.  It may also use usb_set_interface() to specify the
 *      appropriate altsetting.  If unwilling to manage the interface,
 *      return -ENODEV, if genuine IO errors occured, an appropriate
 *      negative errno value.
 * @disconnect: Called when the interface is no longer accessible, usually
 *      because its device has been (or is being) disconnected or the
 *      driver module is being unloaded.
 * @ioctl: Used for drivers that want to talk to userspace through
 *      the "usbfs" filesystem.  This lets devices provide ways to
 *      expose information to user space regardless of where they
 *      do (or don't) show up otherwise in the filesystem.
 * @suspend: Called when the device is going to be suspended by the system.
 * @resume: Called when the device is being resumed by the system.
 * @reset_resume: Called when the suspended device has been reset instead
 *      of being resumed.
 * @pre_reset: Called by usb_reset_device() when the device
 *      is about to be reset.
 * @post_reset: Called by usb_reset_device() after the device
 *      has been reset
 * @id_table: USB drivers use ID table to support hotplugging.
 *      Export this with MODULE_DEVICE_TABLE(usb,...).  This must be set
 *      or your driver's probe function will never get called.
 * @dynids: used internally to hold the list of dynamically added device
 *      ids for this driver.
 * @drvwrap: Driver-model core structure wrapper.
 * @no_dynamic_id: if set to 1, the USB core will not allow dynamic ids to be
 *      added to this driver by preventing the sysfs file from being created.
 * @supports_autosuspend: if set to 0, the USB core will not allow autosuspend
 *      for interfaces bound to this driver.
 * @soft_unbind: if set to 1, the USB core will not kill URBs and disable
 *      endpoints before calling the driver's disconnect method.
 *
 * USB interface drivers must provide a name, probe() and disconnect()
 * methods, and an id_table.  Other driver fields are optional.
 *
```

```
 * The id_table is used in hotplugging.  It holds a set of descriptors,
 * and specialized data may be associated with each entry.  That table
 * is used by both user and kernel mode hotplugging support.
 *
 * The probe() and disconnect() methods are called in a context where
 * they can sleep, but they should avoid abusing the privilege.  Most
 * work to connect to a device should be done when the device is opened,
 * and undone at the last close.  The disconnect code needs to address
 * concurrency issues with respect to open() and close() methods, as
 * well as forcing all pending I/O requests to complete (by unlinking
 * them as necessary, and blocking until the unlinks complete).
 */
struct usb_driver {
        const char *name;

        int (*probe) (struct usb_interface *intf,
                      const struct usb_device_id *id);

        void (*disconnect) (struct usb_interface *intf);

        int (*ioctl) (struct usb_interface *intf, unsigned int code,
                        void *buf);

        int (*suspend) (struct usb_interface *intf, pm_message_t message);
        int (*resume) (struct usb_interface *intf);
        int (*reset_resume)(struct usb_interface *intf);

        int (*pre_reset)(struct usb_interface *intf);
        int (*post_reset)(struct usb_interface *intf);

        const struct usb_device_id *id_table;

        struct usb_dynids dynids;
        struct usbdrv_wrap drvwrap;
        unsigned int no_dynamic_id:1;
        unsigned int supports_autosuspend:1;
        unsigned int soft_unbind:1;
};
#define  to_usb_driver(d) container_of(d, struct usb_driver, drvwrap.driver)

/**
 * struct usb_device_driver - identifies USB device driver to usbcore
 * @name: The driver name should be unique among USB drivers,
 *      and should normally be the same as the module name.
 * @probe: Called to see if the driver is willing to manage a particular
 *      device.  If it is, probe returns zero and uses dev_set_drvdata()
 *      to associate driver-specific data with the device.  If unwilling
 *      to manage the device, return a negative errno value.
 * @disconnect: Called when the device is no longer accessible, usually
 *      because it has been (or is being) disconnected or the driver's
 *      module is being unloaded.
 * @suspend: Called when the device is going to be suspended by the system.
 * @resume: Called when the device is being resumed by the system.
 * @drvwrap: Driver-model core structure wrapper.
 * @supports_autosuspend: if set to 0, the USB core will not allow autosuspend
 *      for devices bound to this driver.
 *
 * USB drivers must provide all the fields listed above except drvwrap.
```

```
 */
struct usb_device_driver {
	const char *name;

	int (*probe) (struct usb_device *udev);
	void (*disconnect) (struct usb_device *udev);

	int (*suspend) (struct usb_device *udev, pm_message_t message);
	int (*resume) (struct usb_device *udev);
	struct usbdrv_wrap drvwrap;
	unsigned int supports_autosuspend:1;
};
#define  to_usb_device_driver(d) container_of(d, struct usb_device_driver, \
		drvwrap.driver)

extern struct bus_type usb_bus_type;

/**
 * struct usb_class_driver - identifies a USB driver that wants to use the USB major number
 * @name: the usb class device name for this driver.  Will show up in sysfs.
 * @fops: pointer to the struct file_operations of this driver.
 * @minor_base: the start of the minor range for this driver.
 *
 * This structure is used for the usb_register_dev() and
 * usb_unregister_dev() functions, to consolidate a number of the
 * parameters used for them.
 */
struct usb_class_driver {
	char *name;
	const struct file_operations *fops;
	int minor_base;
};

/*
 * use these in module_init()/module_exit()
 * and don't forget MODULE_DEVICE_TABLE(usb, ...)
 */
extern int usb_register_driver(struct usb_driver *, struct module *,
				const char *);
static inline int usb_register(struct usb_driver *driver)
{
	return usb_register_driver(driver, THIS_MODULE, KBUILD_MODNAME);
}
extern void usb_deregister(struct usb_driver *);

extern int usb_register_device_driver(struct usb_device_driver *,
			struct module *);
extern void usb_deregister_device_driver(struct usb_device_driver *);

extern int usb_register_dev(struct usb_interface *intf,
				struct usb_class_driver *class_driver);
extern void usb_deregister_dev(struct usb_interface *intf,
				struct usb_class_driver *class_driver);

extern int usb_disabled(void);

/* ------------------------------------------------------------------------ */
```

```
/*
 * URB support, for asynchronous request completions
 */

/*
 * urb->transfer_flags:
 *
 * Note: URB_DIR_IN/OUT is automatically set in usb_submit_urb().
 */
#define URB_SHORT_NOT_OK	0x0001	/* report short reads as errors */
#define URB_ISO_ASAP		0x0002	/* iso-only, urb->start_frame
					 * ignored */
#define URB_NO_TRANSFER_DMA_MAP		0x0004	/* urb->transfer_dma valid on submit */
#define URB_NO_SETUP_DMA_MAP	0x0008	/* urb->setup_dma valid on submit */
#define URB_NO_FSBR		0x0020	/* UHCI-specific */
#define URB_ZERO_PACKET		0x0040	/* Finish bulk OUT with short packet */
#define URB_NO_INTERRUPT	0x0080	/* HINT: no non-error interrupt
					 * needed */
#define URB_FREE_BUFFER		0x0100	/* Free transfer buffer with the URB */

#define URB_DIR_IN		0x0200	/* Transfer from device to host */
#define URB_DIR_OUT		0
#define URB_DIR_MASK		URB_DIR_IN

struct usb_iso_packet_descriptor {
	unsigned int offset;
	unsigned int length;		/* expected length */
	unsigned int actual_length;
	int status;
};

struct urb;

struct usb_anchor {
	struct list_head urb_list;
	wait_queue_head_t wait;
	spinlock_t lock;
	unsigned int poisoned:1;
};

static inline void init_usb_anchor(struct usb_anchor *anchor)
{
	INIT_LIST_HEAD(&anchor->urb_list);
	init_waitqueue_head(&anchor->wait);
	spin_lock_init(&anchor->lock);
}

typedef void (*usb_complete_t)(struct urb *);

/**
 * struct urb - USB Request Block
 * @urb_list: For use by current owner of the URB.
 * @anchor_list: membership in the list of an anchor
 * @anchor: to anchor URBs to a common mooring
 * @ep: Points to the endpoint's data structure.  Will eventually
 *	replace @pipe.
 * @pipe: Holds endpoint number, direction, type, and more.
 *	Create these values with the eight macros available;
```

```
*         usb_{snd,rcv}TYPEpipe(dev,endpoint), where the TYPE is "ctrl"
*         (control), "bulk", "int" (interrupt), or "iso" (isochronous).
*         For example usb_sndbulkpipe() or usb_rcvintpipe().  Endpoint
*         numbers range from zero to fifteen.  Note that "in" endpoint two
*         is a different endpoint (and pipe) from "out" endpoint two.
*         The current configuration controls the existence, type, and
*         maximum packet size of any given endpoint.
* @dev: Identifies the USB device to perform the request.
* @status: This is read in non-iso completion functions to get the
*         status of the particular request.  ISO requests only use it
*         to tell whether the URB was unlinked; detailed status for
*         each frame is in the fields of the iso_frame-desc.
* @transfer_flags: A variety of flags may be used to affect how URB
*         submission, unlinking, or operation are handled.  Different
*         kinds of URB can use different flags.
* @transfer_buffer:  This identifies the buffer to (or from) which
*         the I/O request will be performed (unless URB_NO_TRANSFER_DMA_MAP
*         is set).  This buffer must be suitable for DMA; allocate it with
*         kmalloc() or equivalent.  For transfers to "in" endpoints, contents
*         of this buffer will be modified.  This buffer is used for the data
*         stage of control transfers.
* @transfer_dma: When transfer_flags includes URB_NO_TRANSFER_DMA_MAP,
*         the device driver is saying that it provided this DMA address,
*         which the host controller driver should use in preference to the
*         transfer_buffer.
* @transfer_buffer_length: How big is transfer_buffer.  The transfer may
*         be broken up into chunks according to the current maximum packet
*         size for the endpoint, which is a function of the configuration
*         and is encoded in the pipe.  When the length is zero, neither
*         transfer_buffer nor transfer_dma is used.
* @actual_length: This is read in non-iso completion functions, and
*         it tells how many bytes (out of transfer_buffer_length) were
*         transferred.  It will normally be the same as requested, unless
*         either an error was reported or a short read was performed.
*         The URB_SHORT_NOT_OK transfer flag may be used to make such
*         short reads be reported as errors.
* @setup_packet: Only used for control transfers, this points to eight bytes
*         of setup data.  Control transfers always start by sending this data
*         to the device.  Then transfer_buffer is read or written, if needed.
* @setup_dma: For control transfers with URB_NO_SETUP_DMA_MAP set, the
*         device driver has provided this DMA address for the setup packet.
*         The host controller driver should use this in preference to
*         setup_packet.
* @start_frame: Returns the initial frame for isochronous transfers.
* @number_of_packets: Lists the number of ISO transfer buffers.
* @interval: Specifies the polling interval for interrupt or isochronous
*         transfers.  The units are frames (milliseconds) for for full and low
*         speed devices, and microframes (1/8 millisecond) for highspeed ones.
* @error_count: Returns the number of ISO transfers that reported errors.
* @context: For use in completion functions.  This normally points to
*         request-specific driver context.
* @complete: Completion handler. This URB is passed as the parameter to the
*         completion function.  The completion function may then do what
*         it likes with the URB, including resubmitting or freeing it.
* @iso_frame_desc: Used to provide arrays of ISO transfer buffers and to
*         collect the transfer status for each buffer.
*
* This structure identifies USB transfer requests.  URBs must be allocated by
```

* calling usb_alloc_urb() and freed with a call to usb_free_urb().
* Initialization may be done using various usb_fill_*_urb() functions. URBs
* are submitted using usb_submit_urb(), and pending requests may be canceled
* using usb_unlink_urb() or usb_kill_urb().
*
* Data Transfer Buffers:
*
* Normally drivers provide I/O buffers allocated with kmalloc() or otherwise
* taken from the general page pool. That is provided by transfer_buffer
* (control requests also use setup_packet), and host controller drivers
* perform a dma mapping (and unmapping) for each buffer transferred. Those
* mapping operations can be expensive on some platforms (perhaps using a dma
* bounce buffer or talking to an IOMMU),
* although they're cheap on commodity x86 and ppc hardware.
*
* Alternatively, drivers may pass the URB_NO_xxx_DMA_MAP transfer flags,
* which tell the host controller driver that no such mapping is needed since
* the device driver is DMA-aware. For example, a device driver might
* allocate a DMA buffer with usb_buffer_alloc() or call usb_buffer_map().
* When these transfer flags are provided, host controller drivers will
* attempt to use the dma addresses found in the transfer_dma and/or
* setup_dma fields rather than determining a dma address themselves. (Note
* that transfer_buffer and setup_packet must still be set because not all
* host controllers use DMA, nor do virtual root hubs).
*
* Initialization:
*
* All URBs submitted must initialize the dev, pipe, transfer_flags (may be
* zero), and complete fields. All URBs must also initialize
* transfer_buffer and transfer_buffer_length. They may provide the
* URB_SHORT_NOT_OK transfer flag, indicating that short reads are
* to be treated as errors; that flag is invalid for write requests.
*
* Bulk URBs may
* use the URB_ZERO_PACKET transfer flag, indicating that bulk OUT transfers
* should always terminate with a short packet, even if it means adding an
* extra zero length packet.
*
* Control URBs must provide a setup_packet. The setup_packet and
* transfer_buffer may each be mapped for DMA or not, independently of
* the other. The transfer_flags bits URB_NO_TRANSFER_DMA_MAP and
* URB_NO_SETUP_DMA_MAP indicate which buffers have already been mapped.
* URB_NO_SETUP_DMA_MAP is ignored for non-control URBs.
*
* Interrupt URBs must provide an interval, saying how often (in milliseconds
* or, for highspeed devices, 125 microsecond units)
* to poll for transfers. After the URB has been submitted, the interval
* field reflects how the transfer was actually scheduled.
* The polling interval may be more frequent than requested.
* For example, some controllers have a maximum interval of 32 milliseconds,
* while others support intervals of up to 1024 milliseconds.
* Isochronous URBs also have transfer intervals. (Note that for isochronous
* endpoints, as well as high speed interrupt endpoints, the encoding of
* the transfer interval in the endpoint descriptor is logarithmic.
* Device drivers must convert that value to linear units themselves.)
*
* Isochronous URBs normally use the URB_ISO_ASAP transfer flag, telling
* the host controller to schedule the transfer as soon as bandwidth

```
 * utilization allows, and then set start_frame to reflect the actual frame
 * selected during submission.  Otherwise drivers must specify the start_frame
 * and handle the case where the transfer can't begin then.  However, drivers
 * won't know how bandwidth is currently allocated, and while they can
 * find the current frame using usb_get_current_frame_number () they can't
 * know the range for that frame number.  (Ranges for frame counter values
 * are HC-specific, and can go from 256 to 65536 frames from "now".)
 *
 * Isochronous URBs have a different data transfer model, in part because
 * the quality of service is only "best effort".  Callers provide specially
 * allocated URBs, with number_of_packets worth of iso_frame_desc structures
 * at the end.  Each such packet is an individual ISO transfer.  Isochronous
 * URBs are normally queued, submitted by drivers to arrange that
 * transfers are at least double buffered, and then explicitly resubmitted
 * in completion handlers, so
 * that data (such as audio or video) streams at as constant a rate as the
 * host controller scheduler can support.
 *
 * Completion Callbacks:
 *
 * The completion callback is made in_interrupt(), and one of the first
 * things that a completion handler should do is check the status field.
 * The status field is provided for all URBs.  It is used to report
 * unlinked URBs, and status for all non-ISO transfers.  It should not
 * be examined before the URB is returned to the completion handler.
 *
 * The context field is normally used to link URBs back to the relevant
 * driver or request state.
 *
 * When the completion callback is invoked for non-isochronous URBs, the
 * actual_length field tells how many bytes were transferred.  This field
 * is updated even when the URB terminated with an error or was unlinked.
 *
 * ISO transfer status is reported in the status and actual_length fields
 * of the iso_frame_desc array, and the number of errors is reported in
 * error_count.  Completion callbacks for ISO transfers will normally
 * (re)submit URBs to ensure a constant transfer rate.
 *
 * Note that even fields marked "public" should not be touched by the driver
 * when the urb is owned by the hcd, that is, since the call to
 * usb_submit_urb() till the entry into the completion routine.
 */
struct urb {
        /* private: usb core and host controller only fields in the urb */
        struct kref kref;               /* reference count of the URB */
        void *hcpriv;                   /* private data for host controller */
        atomic_t use_count;             /* concurrent submissions counter */
        u8 reject;                      /* submissions will fail */
        int unlinked;                   /* unlink error code */

        /* public: documented fields in the urb that can be used by drivers */
        struct list_head urb_list;      /* list head for use by the urb's
                                         * current owner */
        struct list_head anchor_list;   /* the URB may be anchored */
        struct usb_anchor *anchor;
        struct usb_device *dev;         /* (in) pointer to associated device */
        struct usb_host_endpoint *ep;   /* (internal) pointer to endpoint */
        unsigned int pipe;              /* (in) pipe information */
```

```
	int status;				/* (return) non-ISO status */
	unsigned int transfer_flags;		/* (in) URB_SHORT_NOT_OK | ...*/
	void *transfer_buffer;			/* (in) associated data buffer */
	dma_addr_t transfer_dma;/* (in) dma addr for transfer_buffer */
	int transfer_buffer_length;/* (in) data buffer length */
	int actual_length;		/* (return) actual transfer length */
	unsigned char *setup_packet;	/* (in) setup packet (control only) */
	dma_addr_t setup_dma;		/* (in) dma addr for setup_packet */
	int start_frame;		/* (modify) start frame (ISO) */
	int number_of_packets;		/* (in) number of ISO packets */
	int interval;			/* (modify) transfer interval
					 * (INT/ISO) */
	int error_count;		/* (return) number of ISO errors */
	void *context;			/* (in) context for completion */
	usb_complete_t complete;/* (in) completion routine */
	struct usb_iso_packet_descriptor iso_frame_desc[0];
					/* (in) ISO ONLY */
};

/* -------------------------------------------------------------------- */

/**
 * usb_fill_control_urb - initializes a control urb
 * @urb: pointer to the urb to initialize.
 * @dev: pointer to the struct usb_device for this urb.
 * @pipe: the endpoint pipe
 * @setup_packet: pointer to the setup_packet buffer
 * @transfer_buffer: pointer to the transfer buffer
 * @buffer_length: length of the transfer buffer
 * @complete_fn: pointer to the usb_complete_t function
 * @context: what to set the urb context to.
 *
 * Initializes a control urb with the proper information needed to submit
 * it to a device.
 */
static inline void usb_fill_control_urb(struct urb *urb,
					struct usb_device *dev,
					unsigned int pipe,
					unsigned char *setup_packet,
					void *transfer_buffer,
					int buffer_length,
					usb_complete_t complete_fn,
					void *context)
{
	urb->dev = dev;
	urb->pipe = pipe;
	urb->setup_packet = setup_packet;
	urb->transfer_buffer = transfer_buffer;
	urb->transfer_buffer_length = buffer_length;
	urb->complete = complete_fn;
	urb->context = context;
}

/**
 * usb_fill_bulk_urb - macro to help initialize a bulk urb
 * @urb: pointer to the urb to initialize.
 * @dev: pointer to the struct usb_device for this urb.
 * @pipe: the endpoint pipe
```

```
 * @transfer_buffer: pointer to the transfer buffer
 * @buffer_length: length of the transfer buffer
 * @complete_fn: pointer to the usb_complete_t function
 * @context: what to set the urb context to.
 *
 * Initializes a bulk urb with the proper information needed to submit it
 * to a device.
 */
static inline void usb_fill_bulk_urb(struct urb *urb,
                                     struct usb_device *dev,
                                     unsigned int pipe,
                                     void *transfer_buffer,
                                     int buffer_length,
                                     usb_complete_t complete_fn,
                                     void *context)
{
        urb->dev = dev;
        urb->pipe = pipe;
        urb->transfer_buffer = transfer_buffer;
        urb->transfer_buffer_length = buffer_length;
        urb->complete = complete_fn;
        urb->context = context;
}

/**
 * usb_fill_int_urb - macro to help initialize a interrupt urb
 * @urb: pointer to the urb to initialize.
 * @dev: pointer to the struct usb_device for this urb.
 * @pipe: the endpoint pipe
 * @transfer_buffer: pointer to the transfer buffer
 * @buffer_length: length of the transfer buffer
 * @complete_fn: pointer to the usb_complete_t function
 * @context: what to set the urb context to.
 * @interval: what to set the urb interval to, encoded like
 *      the endpoint descriptor's bInterval value.
 *
 * Initializes a interrupt urb with the proper information needed to submit
 * it to a device.
 * Note that high speed interrupt endpoints use a logarithmic encoding of
 * the endpoint interval, and express polling intervals in microframes
 * (eight per millisecond) rather than in frames (one per millisecond).
 */
static inline void usb_fill_int_urb(struct urb *urb,
                                    struct usb_device *dev,
                                    unsigned int pipe,
                                    void *transfer_buffer,
                                    int buffer_length,
                                    usb_complete_t complete_fn,
                                    void *context,
                                    int interval)
{
        urb->dev = dev;
        urb->pipe = pipe;
        urb->transfer_buffer = transfer_buffer;
        urb->transfer_buffer_length = buffer_length;
        urb->complete = complete_fn;
        urb->context = context;
        if (dev->speed == USB_SPEED_HIGH)
```

```
                urb->interval = 1 << (interval - 1);
        else
                urb->interval = interval;
        urb->start_frame = -1;
}

extern void usb_init_urb(struct urb *urb);
extern struct urb *usb_alloc_urb(int iso_packets, gfp_t mem_flags);
extern void usb_free_urb(struct urb *urb);
#define usb_put_urb usb_free_urb
extern struct urb *usb_get_urb(struct urb *urb);
extern int usb_submit_urb(struct urb *urb, gfp_t mem_flags);
extern int usb_unlink_urb(struct urb *urb);
extern void usb_kill_urb(struct urb *urb);
extern void usb_poison_urb(struct urb *urb);
extern void usb_unpoison_urb(struct urb *urb);
extern void usb_kill_anchored_urbs(struct usb_anchor *anchor);
extern void usb_poison_anchored_urbs(struct usb_anchor *anchor);
extern void usb_unlink_anchored_urbs(struct usb_anchor *anchor);
extern void usb_anchor_urb(struct urb *urb, struct usb_anchor *anchor);
extern void usb_unanchor_urb(struct urb *urb);
extern int usb_wait_anchor_empty_timeout(struct usb_anchor *anchor,
                                         unsigned int timeout);
extern struct urb *usb_get_from_anchor(struct usb_anchor *anchor);
extern void usb_scuttle_anchored_urbs(struct usb_anchor *anchor);
extern int usb_anchor_empty(struct usb_anchor *anchor);

/**
 * usb_urb_dir_in - check if an URB describes an IN transfer
 * @urb: URB to be checked
 *
 * Returns 1 if @urb describes an IN transfer (device-to-host),
 * otherwise 0.
 */
static inline int usb_urb_dir_in(struct urb *urb)
{
        return (urb->transfer_flags & URB_DIR_MASK) == URB_DIR_IN;
}

/**
 * usb_urb_dir_out - check if an URB describes an OUT transfer
 * @urb: URB to be checked
 *
 * Returns 1 if @urb describes an OUT transfer (host-to-device),
 * otherwise 0.
 */
static inline int usb_urb_dir_out(struct urb *urb)
{
        return (urb->transfer_flags & URB_DIR_MASK) == URB_DIR_OUT;
}

void *usb_buffer_alloc(struct usb_device *dev, size_t size,
        gfp_t mem_flags, dma_addr_t *dma);
void usb_buffer_free(struct usb_device *dev, size_t size,
        void *addr, dma_addr_t dma);

#if 0
struct urb *usb_buffer_map(struct urb *urb);
```

```
void usb_buffer_dmasync(struct urb *urb);
void usb_buffer_unmap(struct urb *urb);
#endif

struct scatterlist;
int usb_buffer_map_sg(const struct usb_device *dev, int is_in,
                              struct scatterlist *sg, int nents);
#if 0
void usb_buffer_dmasync_sg(const struct usb_device *dev, int is_in,
                                    struct scatterlist *sg, int n_hw_ents);
#endif
void usb_buffer_unmap_sg(const struct usb_device *dev, int is_in,
                                   struct scatterlist *sg, int n_hw_ents);

/*-------------------------------------------------------------------*
 *                    SYNCHRONOUS CALL SUPPORT                       *
 *-------------------------------------------------------------------*/

extern int usb_control_msg(struct usb_device *dev, unsigned int pipe,
            __u8 request, __u8 requesttype, __u16 value, __u16 index,
            void *data, __u16 size, int timeout);
extern int usb_interrupt_msg(struct usb_device *usb_dev, unsigned int pipe,
            void *data, int len, int *actual_length, int timeout);
extern int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe,
            void *data, int len, int *actual_length,
            int timeout);

/* wrappers around usb_control_msg() for the most common standard requests */
extern int usb_get_descriptor(struct usb_device *dev, unsigned char desctype,
            unsigned char descindex, void *buf, int size);
extern int usb_get_status(struct usb_device *dev,
            int type, int target, void *data);
extern int usb_string(struct usb_device *dev, int index,
            char *buf, size_t size);

/* wrappers that also update important state inside usbcore */
extern int usb_clear_halt(struct usb_device *dev, int pipe);
extern int usb_reset_configuration(struct usb_device *dev);
extern int usb_set_interface(struct usb_device *dev, int ifnum, int alternate);

/* this request isn't really synchronous, but it belongs with the others */
extern int usb_driver_set_configuration(struct usb_device *udev, int config);

/*
 * timeouts, in milliseconds, used for sending/receiving control messages
 * they typically complete within a few frames (msec) after they're issued
 * USB identifies 5 second timeouts, maybe more in a few cases, and a few
 * slow devices (like some MGE Ellipse UPSes) actually push that limit.
 */
#define USB_CTRL_GET_TIMEOUT    5000
#define USB_CTRL_SET_TIMEOUT    5000


/**
 * struct usb_sg_request - support for scatter/gather I/O
 * @status: zero indicates success, else negative errno
 * @bytes: counts bytes transferred.
 *
```

```
 * These requests are initialized using usb_sg_init(), and then are used
 * as request handles passed to usb_sg_wait() or usb_sg_cancel().  Most
 * members of the request object aren't for driver access.
 *
 * The status and bytecount values are valid only after usb_sg_wait()
 * returns.  If the status is zero, then the bytecount matches the total
 * from the request.
 *
 * After an error completion, drivers may need to clear a halt condition
 * on the endpoint.
 */
struct usb_sg_request {
        int                     status;
        size_t                  bytes;

        /*
         * members below are private: to usbcore,
         * and are not provided for driver access!
         */
        spinlock_t              lock;

        struct usb_device       *dev;
        int                     pipe;
        struct scatterlist      *sg;
        int                     nents;

        int                     entries;
        struct urb              **urbs;

        int                     count;
        struct completion       complete;
};

int usb_sg_init(
        struct usb_sg_request   *io,
        struct usb_device       *dev,
        unsigned                pipe,
        unsigned                period,
        struct scatterlist      *sg,
        int                     nents,
        size_t                  length,
        gfp_t                   mem_flags
);
void usb_sg_cancel(struct usb_sg_request *io);
void usb_sg_wait(struct usb_sg_request *io);


/* ------------------------------------------------------------------- */

/*
 * For various legacy reasons, Linux has a small cookie that's paired with
 * a struct usb_device to identify an endpoint queue.  Queue characteristics
 * are defined by the endpoint's descriptor.  This cookie is called a "pipe",
 * an unsigned int encoded as:
 *
 *  - direction:     bit 7                (0 = Host-to-Device [Out],
 *                                         1 = Device-to-Host [In] ...
 *                                        like endpoint bEndpointAddress)
```

```
 *  - device address:       bits 8-14       ... bit positions known to uhci-hcd
 *  - endpoint:             bits 15-18      ... bit positions known to uhci-hcd
 *  - pipe type:    bits 30-31          (00 = isochronous, 01 = interrupt,
 *                                       10 = control, 11 = bulk)
 *
 * Given the device address and endpoint descriptor, pipes are redundant.
 */

/* NOTE:  these are not the standard USB_ENDPOINT_XFER_* values!! */
/* (yet ... they're the values used by usbfs) */
#define PIPE_ISOCHRONOUS                0
#define PIPE_INTERRUPT                  1
#define PIPE_CONTROL                    2
#define PIPE_BULK                       3

#define usb_pipein(pipe)  ((pipe) & USB_DIR_IN)
#define usb_pipeout(pipe) (!usb_pipein(pipe))

#define usb_pipedevice(pipe)        (((pipe) >> 8) & 0x7f)
#define usb_pipeendpoint(pipe)      (((pipe) >> 15) & 0xf)

#define usb_pipetype(pipe)          (((pipe) >> 30) & 3)
#define usb_pipeisoc(pipe)          (usb_pipetype((pipe)) == PIPE_ISOCHRONOUS)
#define usb_pipeint(pipe) (usb_pipetype((pipe)) == PIPE_INTERRUPT)
#define usb_pipecontrol(pipe)       (usb_pipetype((pipe)) == PIPE_CONTROL)
#define usb_pipebulk(pipe)          (usb_pipetype((pipe)) == PIPE_BULK)

/* The D0/D1 toggle bits ... USE WITH CAUTION (they're almost hcd-internal) */
#define usb_gettoggle(dev, ep, out) (((dev)->toggle[out] >> (ep)) & 1)
#define  usb_dotoggle(dev, ep, out)  ((dev)->toggle[out] ^= (1 << (ep)))
#define usb_settoggle(dev, ep, out, bit) \
                        ((dev)->toggle[out] = ((dev)->toggle[out] & ~(1 << (ep))) | \
                         ((bit) << (ep)))


static inline unsigned int __create_pipe(struct usb_device *dev,
                        unsigned int endpoint)
{
        return (dev->devnum << 8) | (endpoint << 15);
}

/* Create various pipes... */
#define usb_sndctrlpipe(dev,endpoint)       \
        ((PIPE_CONTROL << 30) | __create_pipe(dev, endpoint))
#define usb_rcvctrlpipe(dev,endpoint)       \
        ((PIPE_CONTROL << 30) | __create_pipe(dev, endpoint) | USB_DIR_IN)
#define usb_sndisocpipe(dev,endpoint)       \
        ((PIPE_ISOCHRONOUS << 30) | __create_pipe(dev, endpoint))
#define usb_rcvisocpipe(dev,endpoint)       \
        ((PIPE_ISOCHRONOUS << 30) | __create_pipe(dev, endpoint) | USB_DIR_IN)
#define usb_sndbulkpipe(dev,endpoint)       \
        ((PIPE_BULK << 30) | __create_pipe(dev, endpoint))
#define usb_rcvbulkpipe(dev,endpoint)       \
        ((PIPE_BULK << 30) | __create_pipe(dev, endpoint) | USB_DIR_IN)
#define usb_sndintpipe(dev,endpoint)        \
        ((PIPE_INTERRUPT << 30) | __create_pipe(dev, endpoint))
#define usb_rcvintpipe(dev,endpoint)        \
        ((PIPE_INTERRUPT << 30) | __create_pipe(dev, endpoint) | USB_DIR_IN)
```

```
/*-------------------------------------------------------------------------*/

static inline __u16
usb_maxpacket(struct usb_device *udev, int pipe, int is_out)
{
	struct usb_host_endpoint  *ep;
	unsigned                  epnum = usb_pipeendpoint(pipe);

	if (is_out) {
		WARN_ON(usb_pipein(pipe));
		ep = udev->ep_out[epnum];
	} else {
		WARN_ON(usb_pipeout(pipe));
		ep = udev->ep_in[epnum];
	}
	if (!ep)
		return 0;

	/* NOTE:  only 0x07ff bits are for packet size... */
	return le16_to_cpu(ep->desc.wMaxPacketSize);
}

/* -------------------------------------------------------------------------- */

/* Events from the usb core */
#define USB_DEVICE_ADD		0x0001
#define USB_DEVICE_REMOVE	0x0002
#define USB_BUS_ADD		0x0003
#define USB_BUS_REMOVE		0x0004
extern void usb_register_notify(struct notifier_block *nb);
extern void usb_unregister_notify(struct notifier_block *nb);

#ifdef DEBUG
#define dbg(format, arg...) printk(KERN_DEBUG "%s: " format "\n" , \
	__FILE__ , ## arg)
#else
#define dbg(format, arg...) do {} while (0)
#endif

#define err(format, arg...) printk(KERN_ERR KBUILD_MODNAME ": " \
	format "\n" , ## arg)
#define info(format, arg...) printk(KERN_INFO KBUILD_MODNAME ": " \
	format "\n" , ## arg)
#define warn(format, arg...) printk(KERN_WARNING KBUILD_MODNAME ": " \
	format "\n" , ## arg)

#endif  /* __KERNEL__ */

#endif
```