LECCIÓN USB

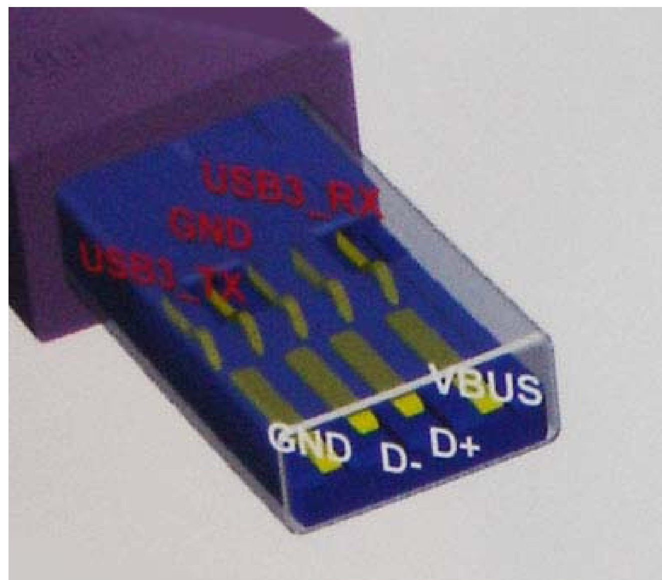
Contenido

Introducción ... 3
Fundamentos de los dispositivos USB ... 4
EndPoints (Línea de Transmisión) ... 4
struct usb_endpoint_descriptor ... 6
Interfaz ... 7
struct usb_interface ... 8
Configuraciones ... 9
USB y /sys ... 9
USB Urbs ... 11
Struct urb ... 12
Funciones de creación y destrucción de urb ... 18
Funciónes de inicialización de una urb ... 19
Tipo Interrupción ... 19
usb_fill_int_urb ... 19
Tipo masivo (Bulk) ... 20
usb_fill_bulk_urb ... 20
Tipo control ... 20
Tipo síncrono ... 21
Funciones de envío de urbs ... 21
usb_submit_urb ... 21
usb_hcd_submit_urb ... 27
Función para finalizar la transmisión de una urb ... 28
Cancelación de Urbs ... 28
Estructura usb_device_id ... 29
Macros para inicializar la estructura usb_device_id ... 31
Estructura usb_driver ... 32
Registrar un manejador USB ... 34
Envió y control de una Urb para un tipo de transmisión masivo ... 35
Transferencias Usb sin urbs ... 38
usb_bulk_msg ... 38
usb_control_msg ... 39
Otras funciones para USB ... 41
Recursos ... 45

Introducción

El Universal Serial Bus (bus universal en serie) es un puerto que sirve para conectar periféricos a una computadora. Fue creado en 1996 por siete empresas: IBM, Intel, Northern Telecom, Compaq, Microsoft, Digital Equipment Corporation y NEC.

El estándar incluye la transmisión de energía eléctrica al dispositivo conectado. Algunos dispositivos requieren una potencia mínima, así que se pueden conectar varios sin necesitar fuentes de alimentación extra. La gran mayoría de los concentradores incluyen fuentes de alimentación que brindan energía a los dispositivos conectados a ellos, pero algunos dispositivos consumen tanta energía que necesitan su propia fuente de alimentación. Los concentradores con fuente de alimentación pueden proporcionarle corriente eléctrica a otros dispositivos sin quitarle corriente al resto de la conexión (dentro de ciertos límites). Las señales del USB son transmitidas por un cable de datos par trenzado con impedancia de 90Ω ±15% llamados D+ y D-. Utilizan señalización diferencial en half dúplex para combatir los efectos del ruido electromagnético en enlaces largos. D+ y D- usualmente operan en conjunto y no son conexiones simples. Las dos conexiones de los extremos son tierra y alimentación.



El diseño del USB tenía en mente eliminar la necesidad de adquirir tarjetas separadas para poner en los puertos bus ISA o PCI, y mejorar las capacidades plug-and-play permitiendo a esos dispositivos ser conectados o desconectados al sistema sin necesidad de reiniciar. Cuando se conecta un nuevo dispositivo, el servidor lo enumera y agrega el software necesario para que pueda funcionar.

El USB puede conectar los periféricos como mouse, teclados, escáneres, cámaras digitales, teléfonos celulares, reproductores multimedia, impresoras, discos duros externos, tarjetas de sonido, sistemas de adquisición de datos y componentes de red. Para dispositivos multimedia como escáneres y cámaras

digitales, el USB se ha convertido en el método estándar de conexión. Para impresoras, el USB ha crecido tanto en popularidad que ha empezado a desplazar a los puertos paralelos porque el USB hace sencillo el poder agregar más de una impresora a una computadora personal.

En el caso de los discos duros, el USB es poco probable que reemplace completamente a los buses como el ATA (IDE) y el SCSI porque el USB tiene un rendimiento un poco más lento que esos otros estándares. El nuevo estándar Serial ATA permite tasas de transferencia de hasta aproximadamente 150/300 MB por segundo. Sin embargo, el USB tiene una importante ventaja en su habilidad de poder instalar y desinstalar dispositivos sin tener que abrir el sistema, lo cual es útil para dispositivos de almacenamiento externo. Hoy en día, una gran parte de los fabricantes ofrece dispositivos USB portátiles que ofrecen un rendimiento casi indistinguible en comparación con los ATA (IDE).

Los dispositivos USB se clasifican en cuatro tipos según su velocidad de transferencia de datos:

- Baja Velocidad (1.0): Bit rate de 1.5Mbit/s (192KB/s). Utilizado en su mayor parte por Dispositivos de Interfaz Humana (HID) como los teclados, los ratones y los joysticks.
- Velocidad Completa (1.1): Bit rate de 12Mbit/s (1.5MB/s). Esta fue la más rápida antes de que se especificara la USB 2.0 y muchos dispositivos fabricados en la actualidad trabajan a esta velocidad. Estos dispositivos, dividen el ancho de banda de la conexión USB entre ellos basados en un algoritmo FIFO.
- Alta Velocidad (2.0): Bitrate de 480Mbit/s (60MB/s).
- Súper Velocidad (3.0) Actualmente en fase experimental. Bit rate de 4.8Gbit/s (600MB/s). Esta especificación será lanzada a mediados de 2008 por la compañía Intel, de acuerdo a información recabada de Internet. Las velocidades de los buses serán 10 veces más rápidas que la de USB 2.0 debido a la inclusión de un enlace de fibra óptica que trabaja con los conectores tradicionales de cobre. Se espera que los productos fabricados con esta tecnología lleguen al consumidor en 2009 o 2010.

Fundamentos de los dispositivos USB

Un dispositivo USB es algo complejo, como se describe en la documentación oficial del USB (www.usb.org). Afortunadamente, el núcleo de Linux provee un subsistema llamado USB core para manejar la mayor parte de la complejidad, a continuación se describiría la interacción entre un manejador y el USB core.

EndPoints (Línea de Transmisión).

La forma más básica de comunicación USB es a través de algo llamado endpoint. Un endpoint USB puede transportar datos en una sola dirección, ya sea desde el ordenador host hacia el dispositivo (llamado OUT endpoint) o desde el dispositivo al ordenador (llamado IN endpoint).

Los **tipos de transferencia de datos** son:

- **Control.**

En el tipo control los parámetros se utilizan para permitir el acceso a diferentes partes del dispositivo USB. Se utiliza para configurar el dispositivo, obtener información sobre el dispositivo, el envío de comandos al dispositivo, o recuperar los informes sobre la situación sobre el dispositivo. Los datos enviados son generalmente de pequeño tamaño. Cada dispositivo USB tiene un tipo de transferencia de control denominado "endpoint 0 "que es utilizado por el núcleo para configurar el dispositivo USB cuando es insertado. Estas transferencias están garantizadas a través del protocolo USB que siempre va a disponer de suficiente ancho de banda.

- **Interrupción**.

  Los endpoints de interrupción transfieren pequeñas cantidades de datos a una velocidad estable cada vez que el computador pide datos al dispositivo USB. Este tipo es el el principal método de comunicación para teclados USB y ratones. También son comúnmente usados para enviar datos a los dispositivos USB para controlar el dispositivo, pero no se utiliza generalmente para la transferencia de grandes cantidades de datos. Estas transferencias están garantizadas por el protocolo USB para que siempre tenga suficiente ancho de banda reservado.

- **Masivo (bulk)**.

  Este tipo transfiere grandes cantidades de datos. Estos datos son normalmente mucho más numerosos que los transmitidos por la transmisión por interrupción. Este tipo es usado normalmente por los dispositivos que necesitan transferir cualquier cantidad de información sin pérdida de datos. Estas transferencias no están garantizadas por el protocolo USB. Si no hay suficiente espacio en el bus para enviar el paquete en su totalidad, se separan a través de múltiples transferencias hacia o desde el dispositivo. Estos tipos son comunes en las impresoras, almacenamiento y dispositivos de red.

- **Síncrono**

  La transferencia síncrona de endpoints también es usada para la transferencia de grandes cantidades de datos pero la llegada de estos datos no siempre esta garantizada. Este tipo de transferencia es usada en dispositivos que pueden manejar la pérdida de datos y que interesa más que mantenga un flujo constante de datos. El este tipo de transferencia es muy usado en dispositivos de audio y video.

Los USB endpoints son descritos en el núcleo en el fichero linux/include/linux/usb.h con la estructura **struct usb_host_endpoint**. Esta estructura contiene a su vez, la información real del endpoint en otra estructura llamada **struct usb_endpoint_descriptor**.

Las peticiones a un USB son siempre encoladas a un endpoint particular, identificado por un descriptor con una interface activa en una configuración USB dada.
La estructura struct usb_host_endpoint, contien la cola de descriptors endpoint en el lado del host.

```
60struct usb_host_endpoint {
61        struct usb_endpoint_descriptor  desc;
62        struct list_head                urb_list;
63        void                            *hcpriv;
64        struct ep_device                *ep_dev;        /* For sysfs info */
65
66        unsigned char *extra;   /* Extra descriptors */
67        int extralen;
68        int enabled;
69};
```

desc: descriptor para este endpoint
urb_list: urbs encoladas a este endpoint; gestionado por usbcore
hcpriv: usado por HCD; mantiene una cola (QH) para el hardware dma con uno o más descriptores de transferencias (TDs) por urb
ep_dev: ep_device para colocar información en sysfs
extra: descriptores que siguen este endpoint en la configuración
extralen: cuantos bytes de "extra" son validos
enabled: endpoint abilitado, URBs pueden ser enviados a este endpoint

struct usb_endpoint_descriptor

La estructura **struct usb_endpoint_descriptor** contiene todos los USB endpoint en el formato exacto que el propio dispositivo ha especificado. linux/include/linux/usb.h/ch9.h

Los campos de esta estructura no poseen una nomenclatura tradicional al núcleo de Linux. Esto se debe a que estos campos corresponden directamente a los nombres de campo en la especificación USB. Los programadores del USB core consideraron que era más importante la utilización de los nombres especificados, a fin de reducir la confusión al leer las especificaciones.

```
313struct usb_endpoint_descriptor {
314        __u8  bLength;
315        __u8  bDescriptorType;
316
317        __u8  bEndpointAddress;
318        __u8  bmAttributes;
319        __le16 wMaxPacketSize;
320        __u8  bInterval;
321
322        /* NOTE:  these two are _only_ in audio endpoints. */
323        /* use USB_DT_ENDPOINT*_SIZE in bLength, not sizeof. */
324        __u8  bRefresh;
325        __u8  bSynchAddress;
326} __attribute__ ((packed));
```

Los campos de esta estructura que son usados por los manejadores son:

- bEndpointAddress

  Esta es la dirección USB de este endpoint. También se incluyen en este

valor de 8-bit la dirección del endpoint. La máscara de bits USB_DIR_OUT y USB_DIR_IN puede ponerse en este campo para determinar si los datos se dirigen al dispositivo o al computador.

- bmAttributes

  Este es el tipo del endpoint. La máscara de bits USB_ENDPOINT_XFERTYPE_MASK puede ponerse en este valor con el fin de determinar si la variable es de tipo USB_ENDPOINT_XFER_ISOC, USB_ENDPOINT_XFER_BULK, o de tipo USB_ENDPOINT_XFER_INT. Estas macros definen los tipos síncrono, masivo, e interrupción de endpoint, respectivamente.

- wMaxPacketSize

  Este es el tamaño máximo en bytes que un endpoint puede manejar a la vez. Es posible que un manejador envié cantidades de datos a un endpoint y que este endpoint sea más pequeño que esa cantidad con lo que los datos se divide en trozos wMaxPacketSize cuando se está enviando al dispositivo. Para dispositivos de alta velocidad, este campo puede ser usado para soportar un alto ancho de banda para el modo endpoint mediante el uso de un par de bits en la parte superior del valor. Véase la especificación USB para obtener más detalles sobre cómo se hace esto.

- bInterval

  Si este parámetro es del tipo interrupción, este valor es el intervalo para el endpoint, es decir, el tiempo transcurrido entre las solicitudes de interrumpir el endpoint. El valor está representado en mili segundos.

Interfaz

Los manejadores USB asignan una interface a un dispositivo USB. Muchos dispositivos utilizan solo una interface pero algunos dispositivos USB tienen varias interfaces; por ejemplo un altavoz USB puede tener dos interfaces para el control de audio, también un teclado USB puede tener varias interfaces para el control de los botones. Debido a que una interfaz USB representa una funcionalidad básica, cada manejador USB controla una interfaz, por lo tanto, para el ejemplo del altavoz, Linux necesita dos controladores diferentes para un dispositivo de hardware.

Cada interface encapsula una función de alto nivel, como el envío de audio a un altavoz o reportar el estado del control de volumen.

Las interfaces pueden tener configuraciones alternativas, con opciones diferentes para los parámetros del interfaz. El estado inicial de una interfaz se encuentra en la primera configuración, con el número 0 pero que pueden cambiar utilizando la función usb_set_interfase(). La configuración alternativa se puede utilizar para controlar los distintos parámetros de diferentes maneras, tales como la reserva de diferentes cantidades de ancho de banda. Los dispositivos con un endpoint síncrono usan varias configuraciones alternativas para la misma interfaz.

struct usb_interface

Las interfaces USB están descritas en el núcleo con la estructura struct usb_interface. Esta estructura es lo que el núcleo USB pasa a los manejadores USB y es lo que los manejadores USB se encarga de controlar. linux/include/linux/usb.h

```
143struct usb_interface {
144       /* array of alternate settings for this interface,
145        * stored in no particular order */
146       struct usb_host_interface *altsetting;
147
148       struct usb_host_interface *cur_altsetting;      /* the currently
149                                         * active alternate setting */
150       unsigned num_altsetting;        /* number of alternate settings */
151
152       /* If there is an interface association descriptor then it will list
153        * the associated interfaces */
154       struct usb_interface_assoc_descriptor *intf_assoc;
155
156       int minor;                      /* minor number this interface is
157                                         * bound to */
158       enum usb_interface_condition condition;         /* state of binding */
159       unsigned is_active:1;           /* the interface is not suspended */
160       unsigned sysfs_files_created:1; /* the sysfs attributes exist */
161       unsigned needs_remote_wakeup:1; /* driver requires remote wakeup */
162
163       struct device dev;              /* interface specific device info */
164       struct device *usb_dev;
165       int pm_usage_cnt;               /* usage counter for autosuspend */
166};
```

Los campos más significativos de esta estructura son los siguientes:

- **struct usb_host_interface *altsetting**

  Vector de estructuras que contienen todas las configuraciones alternativas que pueden ser seleccionadas para esta interfaz. Cada estructura **usb_host_interface** consiste en un vector de configuraciones, tal como se define en la estructura **struct usb_host_endpoint** descrita anteriormente. Las estructuras de la interfaz dentro del vector no tienen ningún orden en particular.

- **struct usb_host_interface *cur_altsetting**

  Un puntero al vector altsetting, que indica la configuración activa en ese momento.

- **unsigned num_altsetting**

  El número de configuraciones alternativas.

- **int minor**

  Si el manejador USB vinculado a este interface utiliza el número mayor de USB, esta variable contiene el número menor asignado por el núcleo USB para la interfaz. La asignación de este número se realiza después

de la llamada a la función usb_register_dev (que se describe más adelante en este capítulo).

Configuraciones

El protocolo utilizado para comunicarse con una interfase endpoint, se puede definir en las especificaciones de la “clase” usb.

El control por defecto de cada endpoint, es parte de cada interfase, pero no viene descrito en el descriptor de la interfase.

El manejador de la interfase pude utilizar un modelo estándar de manejador, llamando a la función dev_get_drvdata(), en el campo dev de su estructura.

Un dispositivo USB puede tener varias configuraciones, la configuración por defecto es la cero, y puede cambiar entre ellas con el fin de cambiar el estado del dispositivo, llamando a la función usb_set_interface(). Solo una configuración puede estar habilitada en un momento dado. Se cambia de configuración cuando se quiere cambiar el ancho de banda reservado a un USB. Por ejemplo las interfases síncronas cambian para no utilizar la configuración por defecto. Linux no maneja una múltiple configuración de un dispositivo USB muy bien, afortunadamente estos dispositivos no son muy abundantes.

Linux describe las configuraciones USB con la estructura struct usb_host_config y todo los dispositivos USB con la estructura struct usb_device. Los manejadores de dispositivo USB no suelen tener la necesidad de leer o escribir los valores en estas estructuras, por lo que no definiremos estos valores en gran detalle.

Un manejador de dispositivo USB normalmente tiene que convertir los datos de una determinada estructura **struct usb_interface** en una estructura **usb_device** que el USB core necesita para una amplia gama de llamadas a funciones. Para ello, se dispone de la función **interface_to_usbdev**.

Por lo tanto, para resumir, los dispositivos USB son bastante complejos y están compuestos de varios lotes de unidades lógicas. Las relaciones entre estas unidades puede ser descrita como sigue:

- Los dispositivos suelen tener una o más configuraciones.
- Las configuraciones suelen tener una o más interfaces.
- Las interfaces generalmente, presentan una o más configuraciones.
- Las interfaces pueden tener varios endpoints, o no tener ninguno.

USB y /sys

Un dispositivo USB es muy complejo, Linux nos provee del /sys que es un sistema de archivos virtual que se encarga de mostrarnos todos los parámetros de los dispositivos, entre ellos los dispositivos USB en forma de ficheros y carpetas. Tanto el dispositivo físico USB (representado por una estructura

usb_device) y cada una de las interfaces USB (representadas por una estructura **usb_interface**) se muestran en /sys como dispositivos individuales. Como un ejemplo, se muestra el árbol bajo /sys de un simple ratón USB que contiene solo una interfaz USB, con los directorios para este dispositivo:

```
/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1
|-- 2-1:1.0
|   |-- bAlternateSetting
|   |-- bInterfaceClass
|   |-- bInterfaceNumber
|   |-- bInterfaceProtocol
|   |-- bInterfaceSubClass
|   |-- bNumEndpoints
|   |-- detach_state
|   |-- iInterface
|   `-- power
|       `-- state
|-- bConfigurationValue
|-- bDeviceClass
|-- bDeviceProtocol
|-- bDeviceSubClass
|-- bMaxPower
|-- bNumConfigurations
|-- bNumInterfaces
|-- bcdDevice
|-- bmAttributes
|-- detach_state
|-- devnum
|-- idProduct
|-- idVendor
|-- maxchild
|-- power
|   `-- state
|-- speed
`-- version
```

La estructura **usb_device** está representada en el árbol por:

/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1

mientras que la estructura interfaz USB del ratón, se representa en el directorio:

/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1/2-1:1.0

Para ayudar a comprender esta larga dirección de dispositivo, describimos a continuación la forma en la que el núcleo etiqueta los dispositivos USB.

El primer dispositivo USB es un hub raíz. Este es el manejador USB, generalmente contenido en un dispositivo PCI. El controlador es llamado así porque controla todo el bus USB. El controlador es un puente entre el bus PCI y el bus USB, y es el primer dispositivo USB en el bus.

A todos los hub raíz, el USB core les asigna un único número. En el ejemplo, el hub raíz se llama usb2, ya que es el segundo hub raiz que se registró con el USB core. No hay límite sobre el número de hubs raíz que pueden existir en cualquier momento en un sistema.

Cada dispositivo presente en el bus USB tiene el número del hub raíz como el

primer número de su nombre. Esto es seguido por un “ - “ y a continuación el número del puerto en donde el dispositivo está conectado. Como el dispositivo en nuestro ejemplo está conectado al primer puerto, se añadirá al nombre el “ 1 “. Así que el nombre del dispositivo USB es 2-1. Debido a que este dispositivo USB contiene una interfaz, implica que otro dispositivo es añadido en el árbol. El sistema de nombres de las interfaces USB, es el nombre del dispositivo que es 2-1 seguido de dos puntos y el número de configuración USB, luego de un punto y el número de interfaz. Por lo tanto, para este ejemplo, el nombre del dispositivo es 2-1:1.0 porque es la primera configuración y el número de interfaz es cero.

Por lo tanto, para resumir, el esquema del dispositivo USB dentro de sysfs es el siguiente:

root_hub - hub_port : config . interface

Como puede observarse en el arbol de directorios mostrado anteriormente todos los parámetros del dispositivo USB están disponibles directamente a través de sysfs como si de archivos se tratasen. Uno de estos archivos, **bConfigurationValue**, puede ser escrito con el fin de cambiar la configuración activa que se está utilizando en ese momento. Esto es útil para dispositivos que disponen de múltiples configuraciones.

Sysfs no expone todas las partes diferentes de un dispositivo USB. Cualquier configuración alternativa que el dispositivo puede contener no se muestra, así como los detalles de los parámetros asociados a las interfaces. Esta información se puede encontrar también en el sistema de ficheros usbfs, que está montado en el directorio **/proc/bus/usb/**. El archivo **/proc/bus/usb/devices** muestra la misma información expuesta en /sys, así como la configuración alternativa del endpoint y la información de todos los dispositivos USB presentes en el sistema. Usbfs también permite que el usuario asigne espacio para que los programas se comuniquen directamente con el dispositivo USB.

USB Urbs

El código USB en el núcleo de Linux se comunica con todos los dispositivos USB mediante una estructura denominada **urb** (petición de bloque USB/ USB request block). Una petición de bloque se describe con la estructura struct urb y el código de las funciones asociadas a la estructura se pueden encontrar en el archivo include/linux/usb.h .

La estructura urb se utiliza para enviar o recibir datos hacia o desde un determinado endpoint en un dispositivo USB de forma asíncrona. Un manejador de dispositivo USB puede asignar muchos urbs a una sola variable o también puede reutilizar para muchos parámetros diferentes solamente un urb, dependiendo de la necesidad. Cada endpoint en un dispositivo puede manejar una cola de urbs, de forma que múltiples urbs pueden ser enviados al mismo endpoint antes de que la cola esté vacía.

Las URBs deben crearse llamando a la función usb_alloc_urb(), y destruidas con la función usb_free_urb(). Se inicializan usando varias funciones como usb_fill_*_urb(). Las URBs son transmitidas usando la función

usb_submit_urb(), y pueden ser canceladas usando usb_unlink_urb() o usb_kill_urb().

El típico ciclo de vida de un urb es el siguiente:

- Se crea por un manejador de dispositivo USB.
- Se asigna a un determinado endpoint de un determinado dispositivo USB.
- Enviada al USB core, por el manejador de dispositivo USB.
- Enviada al manejador del dispositivo USB, por el USB core.
- Procesada por el manejador del host USB que hace una transferencia al dispositivo USB.
- Cuando la urb se ha completado, el manejador del USB notifica al manejador del dispositivo USB.

Los urbs pueden ser cancelados en cualquier momento por el manejador que ha enviado la urb, o por el USB core si el dispositivo es eliminado del sistema. Los urbs son creados dinámicamente y contienen una referencia interna que les permite liberarse automáticamente cuando el usuario final termina las emisiones urb.

El número de endpoints va de cero a quince. Notar que IN endpoint 2 es un endpoint diferente de un OUT endpoint 2.

La configuración de un endpoint, controla la existencia, el tipo, y el máximo tamaño del paquete.

Struct urb

Se encuentra en el fichero include/linux/usb.h

Bufers para la transferencias de datos:

Normalmente los manejadores proporcionan bufers de I/O creados con kmalloc(). Asignados en el campo transfer_buffer (o setup_packet en transmisiones tipo control), y los manejadores del controlador host, realizan un mapeo (o desmapeo) dma para cada buffer. Estos mapeos pueden ser caros en algunas plataformas hardware.

Alternativamente, los manejadores pueden pasar el flag URB_NO_xxx_DMA_MAP, que le dice al manejador del host que no haga mapeo. Cuando se suministra este flag de transferencia, el host intenta usar la dirección de dma encontrada en los campos transfer_dma y o setup_dma en vez de determinar una dirección para dma.

Inicialización:
Todas las URBs antes de transmitirse deben inicializar los campos dev, pipe, transfer_flags, (pueden ser cero). También deben inicializar los campos transfer_buffer y transfer_buffer_length. Pueden activar el flag de transferencia URB_SHORT_NOT_OK, para indicar que las lecturas cortas van a ser tratadas como errores; este flag es inválido para escrituras.

Las URBs de tipo masivo pueden usar el flag de transferencia URB_ZERO_PACKET, para indicar que la transferencia OUT de tipo masivo debe terminar siempre con un paquete corto, incluso si se añade un paquete extra de ceros.

Las URBs de tipo control deben proporcionar un setup_packet. El setup_packet y el transfer_buffer pueden ser mapeados o no para DMA, independientemente uno de otro. Los flags de transferencia URB_NO_TRANSFER_DMA_MAP y URB_NO_SETUP_DMA_MAP, indican que bufers han sido ya mapeados. En URBs que no son de control el flag URB_NO_SETUP_DMA_MAP se ignora.

Para interrupciones, las URBs deben proporcionar un intervalo de tiempo, después del cual se debe interrumpir la transferencia (en milisegundos, o para dispositivos rápidos, en unidades de 125 microsegundos). Después que la URB ha sido transmitida, el campo intervalo indica cuanto tiempo tiene la transferencia antes de ser interrumpida. Por ejemplo, algunos controladores tienen un intervalo máximo de 32 milisegundos mientras que otros tienen 1024 milisegundos. Transferencias síncronas también utilizan intervalo. (Note que para endpoints síncronos, también como para interrupciones de endpoints de alta velocidad, la codificación del intervalo de transferencia en el descriptor del endpoint es logarítmico. Los manejadores de los dispositivos deben convertir este valor a forma lineal.)

Las URBs sincronas normalmente utilizan el flag de transmisión URB_ISO_ASAP, para decirle al host que conmute a otra transferencia tan pronto como el ancho de banda lo permita, y actualice el campo start_frame para que indique el urb que se está transmitiendo. Si no, el manejador debe especificar el start_frame y manejar el caso en el que la transferencia no puede comenzar. Los manejadores no conocen el ancho de banda que ha sido asignado actualmente, ellos pueden conocer el número del rango actual mediante la función usb_get_current_frame_number (), ellos no pueden conocer el rango de este frame.

Las transmisiones sincronas de URBs, tienen un modelo diferente para transferencia de datos. Los llamadores proporcionan URBs especiales, con número de paquetes en vez de estructuras iso_frame_desc. Esta clase de paquete es una transferencia individual ISO. Transferencias síncronas de URBs normalmente son puestas en colas, con buffers dobles, de forma que los datos (como audio o video) tengan una velocidad de transferencia constante.

Función manejadora para finalización de la petición

La función manejadora para la finalización es invocada por in_interrupt(), una de las primeras cosas que hace el manejador es chequear el campo status. Todas las URBs tienen un campo estatus para testear el estado de la transferencia.
El campo context normalmente se utiliza para enlazar la URB con el manejador o con el estado de la petición.

Cuando se invoca una manejador de terminación para una transmisión no síncrona, el campo actual_length, dice cuantos bytes fueron transmitidos. Este campo es actualizado incluso si una URB termina con un error.

El estado de una transferencia ISO es reportado por los campos status y actual_length y el vector iso_frame_desc, y el número de errores en error_count. La función de terminación en ISO, trata de conseguir u una velocidad de transmisión constante.
Notar que incluso los campos marcados como públicos no deben ser tocados por el manejador del dispositivo cuando la urb pertenece al hcd, y la urb ha sido enviada por medio de la función usb_submit_urb().

```
1267struct urb {
1268        /* private: usb core and host controller only fields in the urb */
1269        struct kref kref;               /* reference count of the URB */
1270        void *hcpriv;                   /* private data for host controller */
1271        atomic_t use_count;             /* concurrent submissions counter */
1272        u8 reject;                      /* submissions will fail */
1273        int unlinked;                   /* unlink error code */
```

```
1274
1275      /* public: documented fields in the urb that can be used by drivers */
1276      struct list_head urb_list;      /* list head for use by the urb's
1277                                       * current owner */
1278      struct list_head anchor_list;   /* the URB may be anchored */
1279      struct usb_anchor *anchor;
1280      struct usb_device *dev;         /* (in) pointer to associated device */
1281      struct usb_host_endpoint *ep;   /* (internal) pointer to endpoint */
1282      unsigned int pipe;              /* (in) pipe information */
1283      int status;                     /* (return) non-ISO status */
1284      unsigned int transfer_flags;    /* (in) URB_SHORT_NOT_OK | ...*/
1285      void *transfer_buffer;          /* (in) associated data buffer */
1286      dma_addr_t transfer_dma;        /* (in) dma addr for transfer_buffer */
1287      int transfer_buffer_length;     /* (in) data buffer length */
1288      int actual_length;              /* (return) actual transfer length */
1289      unsigned char *setup_packet;    /* (in) setup packet (control only) */
1290      dma_addr_t setup_dma;           /* (in) dma addr for setup_packet */
1291      int start_frame;                /* (modify) start frame (ISO) */
1292      int number_of_packets;          /* (in) number of ISO packets */
1293      int interval;                   /* (modify) transfer interval
1294                                       * (INT/ISO) */
1295      int error_count;                /* (return) number of ISO errors */
1296      void *context;                  /* (in) context for completion */
1297      usb_complete_t complete;        /* (in) completion routine */
1298      struct usb_iso_packet_descriptor iso_frame_desc[0];
1299                                      /* (in) ISO ONLY */
1300};
1301
```

Los campos de la estructura **struct urb** son los siguientes:

- **urb_list**

  Para uso del propietario actual de la URB.

- **anchor_list**

  miembros de la lista anchor.

- **ep**

  Puntero a la estructura de un endpoint. Reenplaza al campo pipe.

- **struct usb_device *dev**

  Puntero a la estructura **usb_device** a la que la urb es enviada, identifica al dispositivo que hace la petición. Esta variable debe ser inicializada por el manejador USB antes de que la urb sea enviada.

- **unsigned int pipe**

  Información del endpoint (númer, dirección, tipo).

  Para definir estos valores de esta estructura, el manejador utiliza las siguientes macros:

  - **unsigned int usb_sndctrlpipe(struct usb_device *dev, unsigned int endpoint)**

    Especifica un OUT endpoint de control para el dispositivo USB.

  - **unsigned int usb_rcvctrlpipe(struct usb_device *dev, unsigned**

int endpoint)

Especifica un IN endpoint de control para el dispositivo USB.

- **unsigned int usb_sndbulkpipe(struct usb_device *dev, unsigned int endpoint)**

  Especifica un OUT endpoint masivo para el dispositivo USB.

- **unsigned int usb_rcvbulkpipe(struct usb_device *dev, unsigned int endpoint)**

  Especifica un IN endpoint masivo para el dispositivo USB.

- **unsigned int usb_sndintpipe(struct usb_device *dev, unsigned int endpoint)**

  Especifica un OUT endpoint de interrupción para el dispositivo USB.

- **unsigned int usb_rcvintpipe(struct usb_device *dev, unsigned int endpoint)**

  Especifica un IN endpoint de interrupción para un dispositivo USB.

- **unsigned int usb_sndisocpipe(struct usb_device *dev, unsigned int endpoint)**

  Especifica un OUT endpoint sincrono para el dispositivo USB.

- **unsigned int usb_rcvisocpipe(struct usb_device *dev, unsigned int endpoint)**

  Especifica un IN endpoint sincrono para el dispositivo USB.

- **unsigned int transfer_flags**

  Una serie de bits a diferentes valores, definen el modo de transmisión, la terminación y las operaciones sobre una URB, que el manejador USB quiera hacer. Diferentes clases de URB pueden usar flags diferentes. Los valores posibles son los siguientes:

  - **URB_SHORT_NOT_OK**

    Cuando se establece, especifica que una lectura corta en un IN endpoint sea tratada como un error por el USB core. Este valor sólo es útil para urbs que se van a leer desde el dispositivo USB, no para escribir urbs.

  - **URB_ISO_ASAP**

    Si este bit se activa, significa que el urb será atendido tan pronto como sea posible.

  - **URB_NO_TRANSFER_DMA_MAP**

    Puede ser activado cuando el urb contiene un buffer DMA. El USB core usa el buffer apuntado por transfer_dma.

  - **URB_NO_SETUP_DMA_MAP**

    Como el bit URB_NO_TRANSFER_DMA_MAP, es usado para controlar urbs que tienen un buffer DMA listo para crear.

  - **URB_ASYNC_UNLINK**

Si es activado, la llamada usb_unlink_urb es devuelta casi de inmediato para este urb y el urb es desvinculado. Este flag hay que utilizarlo con mucho cuidado.

- **URB_NO_FSBR**

  Usado sólo por el controlador de host USB UHCI. Este bit generalmente no es usado.

- **URB_ZERO_PACKET**

  Si es usado este bit un urb de tipo masivo, termina la transferencia de datos con una serie de datos vacíos.

- **URB_NO_INTERRUPT**

  Si es usado este bit, el hardware no genera ninguna interrupción cuando el urb termina la transferencia.

- **void *transfer_buffer**

  Puntero al buffer al que el dispositivo envía o recibe datos. Se crea con kmalloc() o equivalente. Es incompatible con DMA. Para transferencias a IN endpoints, los contenidos del bufer serán modificados.

- **dma_addr_t transfer_dma**

  Buffer a ser usado para transferir datos al dispositivo USB utilizando DMA. Si se ha activado el flag URB_NO_TRANSFER_DMA_MAP, la dirección del bufer la suministra el manejador del dispositivo y debe usarse en lugar del transfer_buffer.

- **int transfer_buffer_length**

  La longitud del buffer apuntado por el transfer_buffer. La transferencia puede partirse en trozos acordes al máximo tamaño del paquete del endpoint, como se especifica en la configuración del endpoint y está en el campo pipe. Cuando es cero no se utiliza ni el bufer ni el DMA.

- **unsigned char *setup_packet**

  Solo utilizado por las transferencias de tipo control. Contiene 8 bytes de configuración. Una transferencia de control siempre comienza enviando estos datos al dispositivo si se necesita se lee o escribe del transfer bufer.

- **dma_addr_t setup_dma**

  Direccion del buffer a usar para la transferencia. Se utiliza en transferencias de tipo control con el flag URB_NO_SETUP_DMA_MAP activado. El manejador del dispositivo suministra la dirección del DMA para el setup_packet. Se debe usar DMA preferentemente a setup_paquet

- **usb_complete_t complete**

  Puntero a la función manejadora que se debe ejecutar cuando la transferencia está hecha. Este puntero se pasa como parámetro en la función de terminación, la función puede seguir con la transmisión o terminar.

- **void *context**

  Puntero a una variable contexto de un manejador para una petición especifica. Se utiliza en la finalización de una petición.

- **int actual_length**

  Longitud de la transferencia actual en bytes, cuantos bytes son leídos en la transferencia, debe coincidir con la longitud de la solicitud,

- **int status**

  Variable de lectura, cuando la urb ha finalizado, o se esté procesando por el USB core, esta variable indica el estado actual de la urb

  Los valores válidos para esta variable son:

  \- **0** La transferencia Urb fue un éxito.

  -ENOENT La urb fue detenida por una llamada a usb_kill_urb.

  -ECONNRESET La urb fue desvinculada por una llamada a usb_unlink_urb y la variable transfer_flags del urb fue puesta a URB_ASYNC_UNLINK .

  -EINPROGRESS La urb está siendo procesada por el controlador USB. Si el manejador nunca ve este valor, hay un fallo en el manejador.

  -EPROTO Ha ocurrido un error irrecuperable.

  -EILSEQ Un error en la transmisión.

  -EPIPE El endpoint está atascado.

  -ECOMM Los datos fueron transmitidos muy rápido y no dio tiempo a almacenarlos en memoria. Este error solo ocurre con IN urb.

  -ENOSR Los datos no son enviados con la suficiente rapidez.

  -EOVERFLOW Ocurre cuando se han recibidos más datos que los enviados.

  -EREMOTEIO Ocurre cuando no se han recibido todos los datos enviados.

  -ENODEV El dispositivo ya no se encuentra en el sistema.

  -EXDEV Ocurre solo en urb síncrono, y significa que la transferencia no se completo.

  -EINVAL Ha sucedido un error crítico con el urb.

  -ESHUTDOWN Significa que el urb llego cuando el dispositivo fue retirado del sistema.

- **int start_frame**

  Establece o devuelve los datos iníciales en una transmisión síncrona.

- **int interval**

  Especifica el intervalo de tiempo en milisegundos para interrumpir en transmisiones síncronas. Se utiliza en dispositivos lentos. El valor de esta variable varía dependiendo de la velocidad del dispositivo.

- **int number_of_packets**

  Especifica el número de buffers en una transferencia síncrona.

- **int error_count**

  Informa el número de transferencias síncronas que reportaron algún error.

- **struct usb_iso_packet_descriptor iso_frame_desc[0]**

  Esta variable es un vector de estructuras usb_iso_packet_descriptor. Se utiliza para recoger el estado de cada bufer de la transferencia.

  La estructura usb_iso_packet_descriptor se compone de los siguientes campos:

  - **unsigned int offset**

    El desplazamiento en el buffer de transferencia.

  - **unsigned int length**

    La longitud del buffer de transferencia.

  - **unsigned int actual_length**

    La longitud de los datos recibidos en el buffer.

  - **unsigned int status**

    El estado individual de la transferencia síncrona de este paquete.

Funciones de creación y destrucción de urb

La estructura struct urb nunca debe ser creada estática mente en un manejador o dentro de otra estructura, ya que rompería el sistema de conteo de referencia utilizado por el USB core para los urbs. Debe ser creada con una llamada a la función usb_alloc_urb. Esta función tiene el prototipo:

```
struct urb *usb_alloc_urb(int iso_packets, int mem_flags);
```

El primer parámetro, **iso_packets**, es el número de paquetes síncrono que este urb puede contener. Si no desea crear un urb síncrono, esta variable debe ser fijada a 0. El segundo parámetro se usa para la asignación de memoria por el núcleo. Si la función tiene éxito en la asignación de espacio, se devuelve un puntero al urb creado. Si el valor de retorno es NULL, entonces se ha producido un error en el núcleo.

Después de que una urb se haya creado correctamente, debe ser inicializada correctamente antes de que pueda ser utilizada por el USB core.

Con el fin de decirle al USB core que el manejador ha terminado con la urb, el manejador debe llamar a la función **usb_free_urb**. Esta función tiene un solo argumento:

```
void usb_free_urb(struct urb *urb);
```

El argumento es un puntero a la struct urb que se desea liberar. Después de que esta función sea llamada, la estructura urb se ha eliminado y el manejador

ya no puede acceder a ella.

Funciónes de inicialización de una urb

Tipo Interrupción

usb_fill_int_urb

La función usb_fill_int_urb es una función para inicializar correctamente una urb para ser enviada a un endpoint de tipo interrupción de un dispositivo USB:

```
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,
                      void *context,
                      int interval);
```

Esta función contiene una gran cantidad de parámetros:

- struct urb *urb

  Un puntero a la urb a ser inicializado.
- struct usb_device *dev

  Puntero a la estructura usb_device del dispositivo USB para el que se crea esta urb.
- unsigned int pipe

  El endpoint específico de los dispositivo USB para el que se crea este urb.
- void *transfer_buffer

  Un puntero a un buffer para la transferencia de datos. Este no puede ser un búfer estático y debe ser creado con una llamada a kmalloc.
- int buffer_length

  La longitud del buffer apuntado por el puntero transfer_buffer.
- usb_complete_t complete_fn

  Puntero al manejador que se llama cuando se completa una urb.
- void *context

  Puntero que se añade a la estructura urb apuntando a un contexto,

usado por el manejador de terminación para su posterior recuperación.

- int interval

  El intervalo de tiempo tras el cual esta urb debe ser sheduled.

  Endpoints de tipo interrupción, de alta velocidad usan una codificación logarítmica para el intervalo, y expresan el tiempo de intervalo en micro frames (ocho por milisegundo) en vez de frames (una por milisegundo).

```
1383static inline void usb_fill_int_urb(struct urb *urb,
1384                              struct usb_device *dev,
1385                              unsigned int pipe,
1386                              void *transfer_buffer,
1387                              int buffer_length,
1388                              usb_complete_t complete_fn,
1389                              void *context,
1390                              int interval)
1391{
1392        urb->dev = dev;
1393        urb->pipe = pipe;
1394        urb->transfer_buffer = transfer_buffer;
1395        urb->transfer_buffer_length = buffer_length;
1396        urb->complete = complete_fn;
1397        urb->context = context;
1398        if (dev->speed == USB_SPEED_HIGH)
1399                urb->interval = 1 << (interval - 1);
1400        else
1401                urb->interval = interval;
1402        urb->start_frame = -1;
1403}
```

Tipo masivo (Bulk)

usb_fill_bulk_urb

La función que inicializa la estructura es **usb_fill_bulk_urb**, y se muestra como:

```
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,
                       void *context)
```

Los parámetros de la función son todos los mismos que la funcion **usb_fill_int_urb**. Sin embargo, no tiene el parámetro de intervalo.

La función usb_fill_int_urb no establece la variable **transfer_flags** en el urb, por lo que cualquier modificación de este campo debe de ser realizada por el propio manejador.

Tipo control

En el tipo control se inicializa la estructura urb casi del mismo modo que bulk, con una llamada a la función:

```
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,
                          void *context);
```

Los parámetros de la función son todos los mismos que en el **usb_fill_bulk_urb**, excepto que hay un nuevo parámetro, **unsigned char * setup_packet**, que debe apuntar a la configuración de paquetes de datos que se enviará al final.

La función **usb_fill_control_urb** no establece la variable **transfer_flags** en la urb, por lo que cualquier modificación a este campo tiene que ser realizada por el propio manejador.

Tipo síncrono

Lamentablemente, no tiene una función para inicializar como en los tipos interrupción, control y masivo. Así que debe ser inicializado "a mano" en el manejador antes de que pueda ser enviado al USB core.

El siguiente es un ejemplo de cómo inicializar correctamente este tipo de urb. Este ejemplo ha sido tomado de **konicawc.c** situado en driver/usb/media:

```
urb->dev = dev;
urb->context = uvd;
urb->pipe = usb_rcvisocpipe(dev, uvd->video_endp-1);
urb->interval = 1;
urb->transfer_flags = URB_ISO_ASAP;
urb->transfer_buffer = cam->sts_buf[i];
urb->complete = konicawc_isoc_irq;
urb->number_of_packets = FRAMES_PER_DESC;
urb->transfer_buffer_length = FRAMES_PER_DESC;
for (j=0; j < FRAMES_PER_DESC; j++) {
        urb->iso_frame_desc[j].offset = j;
        urb->iso_frame_desc[j].length = 1;
}
```

Funciones de envío de urbs

usb_submit_urb

Una vez que la estructura urb ha sido creada e inicializada por el manejador USB, esta lista para ser enviada hacia el USB core. Esto se hace con una llamada a la función **usb_submit_urb**, su código se encuentra en drivers/usb/core/urb.c

```
int usb_submit_urb(struct urb *urb, int mem_flags);
```

El parámetro urb es un puntero a la urb. El parametro **mem_flags** es

equivalente con el parámetro que se le pasa a la llamada **kmalloc** y se usa para decirle al núcleo la manera de asignar los buffers de memoria en este momento.

Después de que una urb se ha enviado con éxito, no se debe tratar de acceder a ningún campo de la estructura urb hasta completar la llamada a la función.

Debido a que la función **usb_submit_urb** puede ser llamada en cualquier momento, la especificación de la variable **mem_flags** debe ser correcta. Realmente hay sólo tres valores válidos que deben utilizarse, dependiendo de cuando se realiza la llamada a **usb_submit_urb**:

- GFP_ATOMIC

  Este valor debe ser utilizado cuando se den las siguientes circunstancias:

  - La llamada está dentro de un manejador de urb finalizado, una interrupción, una tasklet.
  - El llamador está utilizando un spinlock o rwlock. Note que si un semáforo está siendo utilizado, este valor no es necesario.
  - El campo current->state no está en TASK_RUNNING . El estado del proceso es siempre TASK_RUNNING a menos que el manejador haya cambiado el estado el mismo.

- GFP_NOIO

  Este valor debe utilizarse si el manejador se encuentra en el bloque I/O. También debe ser utilizado en el manejador de todos los dispositivos de almacenamiento.

- GFP_KERNEL

  Este caso se debe usar para las demás situaciones.

```
169/**
170 * usb_submit_urb - issue an asynchronous transfer request for an endpoint
171 * @urb: pointer to the urb describing the request
172 * @mem_flags: the type of memory to allocate, see kmalloc() for a list
173 *      of valid options for this.
174 *
175 * This submits a transfer request, and transfers control of the URB
176 * describing that request to the USB subsystem.  Request completion will
177 * be indicated later, asynchronously, by calling the completion handler.
178 * The three types of completion are success, error, and unlink
179 * (a software-induced fault, also called "request cancellation").
180 *
181 * URBs may be submitted in interrupt context.
182 *
183 * The caller must have correctly initialized the URB before submitting
184 * it.  Functions such as usb_fill_bulk_urb() and usb_fill_control_urb() are
185 * available to ensure that most fields are correctly initialized, for
186 * the particular kind of transfer, although they will not initialize
187 * any transfer flags.
188 *
189 * Successful submissions return 0; otherwise this routine returns a
190 * negative error number.  If the submission is successful, the complete()
191 * callback from the URB will be called exactly once, when the USB core and
192 * Host Controller Driver (HCD) are finished with the URB.  When the completion
```

```
193 * function is called, control of the URB is returned to the device
194 * driver which issued the request.  The completion handler may then
195 * immediately free or reuse that URB.
196 *
197 * With few exceptions, USB device drivers should never access URB fields
198 * provided by usbcore or the HCD until its complete() is called.
199 * The exceptions relate to periodic transfer scheduling.  For both
200 * interrupt and isochronous urbs, as part of successful URB submission
201 * urb->interval is modified to reflect the actual transfer period used
202 * (normally some power of two units).  And for isochronous urbs,
203 * urb->start_frame is modified to reflect when the URB's transfers were
204 * scheduled to start.  Not all isochronous transfer scheduling policies
205 * will work, but most host controller drivers should easily handle ISO
206 * queues going from now until 10-200 msec into the future.
207 *
208 * For control endpoints, the synchronous usb_control_msg() call is
209 * often used (in non-interrupt context) instead of this call.
210 * That is often used through convenience wrappers, for the requests
211 * that are standardized in the USB 2.0 specification.  For bulk
212 * endpoints, a synchronous usb_bulk_msg() call is available.
213 *
214 * Request Queuing:
215 *
216 * URBs may be submitted to endpoints before previous ones complete, to
217 * minimize the impact of interrupt latencies and system overhead on data
218 * throughput.  With that queuing policy, an endpoint's queue would never
219 * be empty.  This is required for continuous isochronous data streams,
220 * and may also be required for some kinds of interrupt transfers. Such
221 * queuing also maximizes bandwidth utilization by letting USB controllers
222 * start work on later requests before driver software has finished the
223 * completion processing for earlier (successful) requests.
224 *
225 * As of Linux 2.6, all USB endpoint transfer queues support depths greater
226 * than one.  This was previously a HCD-specific behavior, except for ISO
227 * transfers.  Non-isochronous endpoint queues are inactive during cleanup
228 * after faults (transfer errors or cancellation).
229 *
230 * Reserved Bandwidth Transfers:
231 *
232 * Periodic transfers (interrupt or isochronous) are performed repeatedly,
233 * using the interval specified in the urb.  Submitting the first urb to
234 * the endpoint reserves the bandwidth necessary to make those transfers.
235 * If the USB subsystem can't allocate sufficient bandwidth to perform
236 * the periodic request, submitting such a periodic request should fail.
237 *
238 * Device drivers must explicitly request that repetition, by ensuring that
239 * some URB is always on the endpoint's queue (except possibly for short
240 * periods during completion callacks).  When there is no longer an urb
241 * queued, the endpoint's bandwidth reservation is canceled.  This means
242 * drivers can use their completion handlers to ensure they keep bandwidth
243 * they need, by reinitializing and resubmitting the just-completed urb
244 * until the driver longer needs that periodic bandwidth.
245 *
246 * Memory Flags:
247 *
248 * The general rules for how to decide which mem_flags to use
249 * are the same as for kmalloc.  There are four
250 * different possible values; GFP_KERNEL, GFP_NOFS, GFP_NOIO and
251 * GFP_ATOMIC.
252 *
```

```
253 * GFP_NOFS is not ever used, as it has not been implemented yet.
254 *
255 * GFP_ATOMIC is used when
256 *   (a) you are inside a completion handler, an interrupt, bottom half,
257 *       tasklet or timer, or
258 *   (b) you are holding a spinlock or rwlock (does not apply to
259 *       semaphores), or
260 *   (c) current->state != TASK_RUNNING, this is the case only after
261 *       you've changed it.
262 *
263 * GFP_NOIO is used in the block io path and error handling of storage
264 * devices.
265 *
266 * All other situations use GFP_KERNEL.
267 *
268 * Some more specific rules for mem_flags can be inferred, such as
269 *  (1) start_xmit, timeout, and receive methods of network drivers must
270 *      use GFP_ATOMIC (they are called with a spinlock held);
271 *  (2) queuecommand methods of scsi drivers must use GFP_ATOMIC (also
272 *      called with a spinlock held);
273 *  (3) If you use a kernel thread with a network driver you must use
274 *      GFP_NOIO, unless (b) or (c) apply;
275 *  (4) after you have done a down() you can use GFP_KERNEL, unless (b) or (c)
276 *      apply or your are in a storage driver's block io path;
277 *  (5) USB probe and disconnect can use GFP_KERNEL unless (b) or (c) apply; and
278 *  (6) changing firmware on a running storage or net device uses
279 *      GFP_NOIO, unless b) or c) apply
280 *
281 */
282int usb_submit_urb(struct urb *urb, gfp_t mem_flags)
283{
284        int                             xfertype, max;
285        struct usb_device               *dev;
286        struct usb_host_endpoint        *ep;
287        int                             is_out;
288
289        if (!urb || urb->hcpriv || !urb->complete)
290                return -EINVAL;
291        dev = urb->dev;
292        if ((!dev) || (dev->state < USB_STATE_DEFAULT))
293                return -ENODEV;
294
295        /* For now, get the endpoint from the pipe.  Eventually drivers
296         * will be required to set urb->ep directly and we will eliminate
297         * urb->pipe.
298         */
299        ep = (usb_pipein(urb->pipe) ? dev->ep_in : dev->ep_out)
300                        [usb_pipeendpoint(urb->pipe)];
301        if (!ep)
302                return -ENOENT;
303
304        urb->ep = ep;
305        urb->status = -EINPROGRESS;
306        urb->actual_length = 0;
307
308        /* Lots of sanity checks, so HCDs can rely on clean data
309         * and don't need to duplicate tests
310         */
311        xfertype = usb_endpoint_type(&ep->desc);
312        if (xfertype == USB_ENDPOINT_XFER_CONTROL) {
```

```
313             struct usb_ctrlrequest *setup =
314                     (struct usb_ctrlrequest *) urb->setup_packet;
315
316             if (!setup)
317                 return -ENOEXEC;
318             is_out = !(setup->bRequestType & USB_DIR_IN) ||
319                     !setup->wLength;
320         } else {
321             is_out = usb_endpoint_dir_out(&ep->desc);
322         }
323
324         /* Cache the direction for later use */
325         urb->transfer_flags = (urb->transfer_flags & ~URB_DIR_MASK) |
326                 (is_out ? URB_DIR_OUT : URB_DIR_IN);
327
328         if (xfertype != USB_ENDPOINT_XFER_CONTROL &&
329                 dev->state < USB_STATE_CONFIGURED)
330             return -ENODEV;
331
332         max = le16_to_cpu(ep->desc.wMaxPacketSize);
333         if (max <= 0) {
334             dev_dbg(&dev->dev,
335                 "bogus endpoint ep%d%s in %s (bad maxpacket %d)\n",
336                 usb_endpoint_num(&ep->desc), is_out ? "out" : "in",
337                 __FUNCTION__, max);
338             return -EMSGSIZE;
339         }
340
341         /* periodic transfers limit size per frame/uframe,
342          * but drivers only control those sizes for ISO.
343          * while we're checking, initialize return status.
344          */
345         if (xfertype == USB_ENDPOINT_XFER_ISOC) {
346             int     n, len;
347
348             /* "high bandwidth" mode, 1-3 packets/uframe? */
349             if (dev->speed == USB_SPEED_HIGH) {
350                 int     mult = 1 + ((max >> 11) & 0x03);
351                 max &= 0x07ff;
352                 max *= mult;
353             }
354
355             if (urb->number_of_packets <= 0)
356                 return -EINVAL;
357             for (n = 0; n < urb->number_of_packets; n++) {
358                 len = urb->iso_frame_desc[n].length;
359                 if (len < 0 || len > max)
360                     return -EMSGSIZE;
361                 urb->iso_frame_desc[n].status = -EXDEV;
362                 urb->iso_frame_desc[n].actual_length = 0;
363             }
364         }
365
366         /* the I/O buffer must be mapped/unmapped, except when length=0 */
367         if (urb->transfer_buffer_length < 0)
368             return -EMSGSIZE;
369
370#ifdef DEBUG
371         /* stuff that drivers shouldn't do, but which shouldn't
372          * cause problems in HCDs if they get it wrong.
```

```
373	 */
374	{
375	unsigned int	orig_flags = urb->transfer_flags;
376	unsigned int	allowed;
377
378	/* enforce simple/standard policy */
379	allowed = (URB_NO_TRANSFER_DMA_MAP | URB_NO_SETUP_DMA_MAP |
380			URB_NO_INTERRUPT | URB_DIR_MASK | URB_FREE_BUFFER);
381	switch (xfertype) {
382	case USB_ENDPOINT_XFER_BULK:
383		if (is_out)
384			allowed |= URB_ZERO_PACKET;
385		/* FALLTHROUGH */
386	case USB_ENDPOINT_XFER_CONTROL:
387		allowed |= URB_NO_FSBR; /* only affects UHCI */
388		/* FALLTHROUGH */
389	default:			/* all non-iso endpoints */
390		if (!is_out)
391			allowed |= URB_SHORT_NOT_OK;
392		break;
393	case USB_ENDPOINT_XFER_ISOC:
394		allowed |= URB_ISO_ASAP;
395		break;
396	}
397	urb->transfer_flags &= allowed;
398
399	/* fail if submitter gave bogus flags */
400	if (urb->transfer_flags != orig_flags) {
401		err("BOGUS urb flags, %x --> %x",
402			orig_flags, urb->transfer_flags);
403		return -EINVAL;
404	}
405	}
406#endif
407	/*
408	 * Force periodic transfer intervals to be legal values that are
409	 * a power of two (so HCDs don't need to).
410	 *
411	 * FIXME want bus->{intr,iso}_sched_horizon values here.  Each HC
412	 * supports different values... this uses EHCI/UHCI defaults (and
413	 * EHCI can use smaller non-default values).
414	 */
415	switch (xfertype) {
416	case USB_ENDPOINT_XFER_ISOC:
417	case USB_ENDPOINT_XFER_INT:
418		/* too small? */
419		if (urb->interval <= 0)
420			return -EINVAL;
421		/* too big? */
422		switch (dev->speed) {
423		case USB_SPEED_HIGH:	/* units are microframes */
424			/* NOTE usb handles 2^15 */
425			if (urb->interval > (1024 * 8))
426				urb->interval = 1024 * 8;
427			max = 1024 * 8;
428			break;
429		case USB_SPEED_FULL:	/* units are frames/msec */
430		case USB_SPEED_LOW:
431			if (xfertype == USB_ENDPOINT_XFER_INT) {
432				if (urb->interval > 255)
```

```
433                             return -EINVAL;
434                     /* NOTE ohci only handles up to 32 */
435                     max = 128;
436             } else {
437                     if (urb->interval > 1024)
438                             urb->interval = 1024;
439                     /* NOTE usb and ohci handle up to 2^15 */
440                     max = 1024;
441             }
442             break;
443     default:
444             return -EINVAL;
445     }
446     /* Round down to a power of 2, no more than max */
447     urb->interval = min(max, 1 << ilog2(urb->interval));
448 }
449
450 return usb_hcd_submit_urb(urb, mem_flags);
451}
```

usb_hcd_submit_urb

```
1292/* may be called in any context with a valid urb->dev usecount
1293 * caller surrenders "ownership" of urb
1294 * expects usb_submit_urb() to have sanity checked and conditioned all
1295 * inputs in the urb
1296 */
1297int usb_hcd_submit_urb (struct urb *urb, gfp_t mem_flags)
1298{
1299    int                     status;
1300    struct usb_hcd          *hcd = bus_to_hcd(urb->dev->bus);
1301
1302    /* increment urb's reference count as part of giving it to the HCD
1303     * (which will control it).  HCD guarantees that it either returns
1304     * an error or calls giveback(), but not both.
1305     */
1306    usb_get_urb(urb);
1307    atomic_inc(&urb->use_count);
1308    atomic_inc(&urb->dev->urbnum);
1309    usbmon_urb_submit(&hcd->self, urb);
1310
1311    /* NOTE requirements on root-hub callers (usbfs and the hub
1312     * driver, for now):  URBs' urb->transfer_buffer must be
1313     * valid and usb_buffer_{sync,unmap}() not be needed, since
1314     * they could clobber root hub response data.  Also, control
1315     * URBs must be submitted in process context with interrupts
1316     * enabled.
1317     */
1318    status = map_urb_for_dma(hcd, urb, mem_flags);
1319    if (unlikely(status)) {
1320            usbmon_urb_submit_error(&hcd->self, urb, status);
1321            goto error;
1322    }
1323
1324    if (is_root_hub(urb->dev))
1325            status = rh_urb_enqueue(hcd, urb);
1326    else
1327            status = hcd->driver->urb_enqueue(hcd, urb, mem_flags);
```

```
1328
1329        if (unlikely(status)) {
1330                usbmon_urb_submit_error(&hcd->self, urb, status);
1331                unmap_urb_for_dma(hcd, urb);
1332 error:
1333                urb->hcpriv = NULL;
1334                INIT_LIST_HEAD(&urb->urb_list);
1335                atomic_dec(&urb->use_count);
1336                atomic_dec(&urb->dev->urbnum);
1337                if (urb->reject)
1338                        wake_up(&usb_kill_urb_queue);
1339                usb_put_urb(urb);
1340        }
1341        return status;
1342}
```

Función para finalizar la transmisión de una urb

Si la función usb_submit_urb, de envío de una urb, fue un éxito, la función devuelve 0, si la función falla devolverá un número negativo. Si el envío tiene éxito, el manejador llama a la función "complete()", para retornar. Cuando se llama a esta función, el USB core (Host Controller Driver (HCD)) ha terminado con el urb y el control se devuelve al manejador del dispositivo que origino la transmisión de la urb.

Sólo hay tres maneras de que una urb pueda ser terminada:

- La urb se ha enviado al dispositivo, y el dispositivo devuelve la señal adecuada. Si esto ha ocurrido, la variable de estado de la urb estará a 0.
- Algún tipo de error ocurre cuando se realiza un envío o recepción de datos desde el dispositivo.
- La urb fue "desasignada" del USB core.

Cancelación de Urbs

Para detener una urb que se ha enviado al USB core, debería llamarse a las funciones **usb_kill_urb** o **usb_unlink_urb**:

int usb_kill_urb(struct urb *urb);

int usb_unlink_urb(struct urb *urb);

El parámetro urb para estas dos funciones es un puntero al urb que se ha cancelado.

Cuando la función es **usb_kill_urb**, el ciclo de vida del urb se detiene. Esta función se usa generalmente cuando el dispositivo está desconectado del sistema, en la llamada de desconexión.

Estructura usb_device_id

La estructura struct usb_device_id proporciona una lista de los diferentes tipos de dispositivos USB que soporta este controlador. Esta lista es utilizada por el USB core para decidir que manejador asignar a un dispositivo y si se usa la conexión en caliente, para decidir qué controlador cargara automáticamente.

```
41/*
42 * Device table entry for "new style" table-driven USB drivers.
43 * User mode code can read these tables to choose which modules to load.
44 * Declare the table as a MODULE_DEVICE_TABLE.
45 *
46 * A probe() parameter will point to a matching entry from this table.
47 * Use the driver_info field for each match to hold information tied
48 * to that match:  device quirks, etc.
49 *
50 * Terminate the driver's table with an all-zeroes entry.
51 * Use the flag values to control which fields are compared.
52 */
53
54/**
55 * struct usb_device_id - identifies USB devices for probing and hotplugging
56 * @match_flags: Bit mask controlling of the other fields are used to match
57 *      against new devices.  Any field except for driver_info may be used,
58 *      although some only make sense in conjunction with other fields.
59 *      This is usually set by a USB_DEVICE_*() macro, which sets all
60 *      other fields in this structure except for driver_info.
61 * @idVendor: USB vendor ID for a device; numbers are assigned
62 *      by the USB forum to its members.
63 * @idProduct: Vendor-assigned product ID.
64 * @bcdDevice_lo: Low end of range of vendor-assigned product version numbers.
65 *      This is also used to identify individual product versions, for
66 *      a range consisting of a single device.
67 * @bcdDevice_hi: High end of version number range.  The range of product
68 *      versions is inclusive.
69 * @bDeviceClass: Class of device; numbers are assigned
70 *      by the USB forum.  Products may choose to implement classes,
71 *      or be vendor-specific.  Device classes specify behavior of all
72 *      the interfaces on a devices.
73 * @bDeviceSubClass: Subclass of device; associated with bDeviceClass.
74 * @bDeviceProtocol: Protocol of device; associated with bDeviceClass.
75 * @bInterfaceClass: Class of interface; numbers are assigned
76 *      by the USB forum.  Products may choose to implement classes,
77 *      or be vendor-specific.  Interface classes specify behavior only
78 *      of a given interface; other interfaces may support other classes.
79 * @bInterfaceSubClass: Subclass of interface; associated with bInterfaceClass.
80 * @bInterfaceProtocol: Protocol of interface; associated with bInterfaceClass.
81 * @driver_info: Holds information used by the driver.  Usually it holds
82 *      a pointer to a descriptor understood by the driver, or perhaps
83 *      device flags.
84 *
85 * In most cases, drivers will create a table of device IDs by using
86 * USB_DEVICE(), or similar macros designed for that purpose.
87 * They will then export it to userspace using MODULE_DEVICE_TABLE(),
88 * and provide it to the USB core through their usb_driver structure.
```

```
 89 *
 90 * See the usb_match_id() function for information about how matches are
 91 * performed.  Briefly, you will normally use one of several macros to help
 92 * construct these entries.  Each entry you provide will either identify
 93 * one or more specific products, or will identify a class of products
 94 * which have agreed to behave the same.  You should put the more specific
 95 * matches towards the beginning of your table, so that driver_info can
 96 * record quirks of specific products.
 97 */
 98struct usb_device_id {
 99         /* which fields to match against? */
100         __u16           match_flags;
101 
102         /* Used for product specific matches; range is inclusive */
103         __u16           idVendor;
104         __u16           idProduct;
105         __u16           bcdDevice_lo;
106         __u16           bcdDevice_hi;
107 
108         /* Used for device class matches */
109         __u8            bDeviceClass;
110         __u8            bDeviceSubClass;
111         __u8            bDeviceProtocol;
112 
113         /* Used for interface class matches */
114         __u8            bInterfaceClass;
115         __u8            bInterfaceSubClass;
116         __u8            bInterfaceProtocol;
117 
118         /* not matched against */
119         kernel_ulong_t  driver_info;
120};
121 
```

La estructura **struct usb_device_id** tiene su código en /include/linux/mod_devicetable.h con los siguientes campos:

- _ _u16 match_flags

  Determina cual de los siguientes campos en la estructura del dispositivo deben ser activados. Este campo es definido por diferentes valores usb_device_id_match_* especificados en **include/linux/mod_devicetable.h**. Este campo normalmente no es inicializado por las macros del tipo usb_device.

- _ _u16 idVendor

  La identificación del vendedor para el dispositivo. Este número es asignado por el foro USB y sus miembros y no se puede asignar por nadie más.

- _ _u16 idProduct

  La identificacion del producto para el dispositivo.

- _ _u16 bcdDevice_lo
- _ _u16 bcdDevice_hi

  Define el extremo inferior y el superior del número de version de un

dispositivo. Estos valores se expresan en binario con código decimal (BCD). Estas variables, junto con el idVendor y idProduct, se utilizan para definir una versión específica de un dispositivo.

- _ _u8 bDeviceClass
- _ _u8 bDeviceSubClass
- _ _u8 bDeviceProtocol

  Define la clase, subclase, y el protocolo del dispositivo, respectivamente. Estos valores especifican el comportamiento de todo el dispositivo, incluyendo todas las interfaces de este dispositivo.

- _ _u8 bInterfaceClass
- _ _u8 bInterfaceSubClass
- _ _u8 bInterfaceProtocol

  Definen la clase, subclase, y el protocolo de la interfaz individual, respectivamente.

Macros para inicializar la estructura usb_device_id

Al igual que con los dispositivos PCI, hay una serie de macros que se utilizan para inicializar esta estructura, definidas en linux/include/linux/usb.h

- USB_DEVICE(vendor, product)

  Crea una estructura usb_device_id que especifica los valores del vendedor y los valores de ID de producto. Esto es utilizado por dispositivos USB que necesitan un controlador específico.

- USB_DEVICE_VER(vendor, product, lo, hi)

  Crea una estructura usb_device_id que pueden utilizarse para especificar la identificacion del vendedor del dispositivo y los valores del rango de versiones.

- USB_DEVICE_INFO(class, subclass, protocol)

  Crea una estructura usb_device_id que se usa para clasificar un dispositivos USB.

- USB_INTERFACE_INFO(class, subclass, protocol)

  Crea una estructura usb_device_id que se usa para clasificar un interfaz USB.

Así, para un manejador simple de un dispositivo USB que controla un único dispositivo USB de un solo proveedor, la estructura usb_device_id tabla se define como:

```
/* table of devices that work with this manejador */
static struct usb_device_id skel_table [ ] = {
  { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
```

```
    { }               /* Terminating entry */
};
MODULE_DEVICE_TABLE (usb, skel_table);
```

La macro **MODULE_DEVICE_TABLE** es necesaria para permitir que las herramientas del espacio de usuario configuren que dispositivos puede controlar el manejador.

Estructura usb_driver

Es la estructura principal que todos los manejadores USB deben crear. Esta estructura debe ser inicializada por el manejador USB y contiene una serie de funciones y variables que describen el manejador USB. Su código se encuentra en /linux/include/linux/usb.h

```
895/**
896 * struct usb_driver - identifies USB interface driver to usbcore
897 * @name: The driver name should be unique among USB drivers,
898 *      and should normally be the same as the module name.
899 * @probe: Called to see if the driver is willing to manage a particular
900 *      interface on a device.  If it is, probe returns zero and uses
901 *      dev_set_drvdata() to associate driver-specific data with the
902 *      interface.  It may also use usb_set_interface() to specify the
903 *      appropriate altsetting.  If unwilling to manage the interface,
904 *      return a negative errno value.
905 * @disconnect: Called when the interface is no longer accessible, usually
906 *      because its device has been (or is being) disconnected or the
907 *      driver module is being unloaded.
908 * @ioctl: Used for drivers that want to talk to userspace through
909 *      the "usbfs" filesystem.  This lets devices provide ways to
910 *      expose information to user space regardless of where they
911 *      do (or don't) show up otherwise in the filesystem.
912 * @suspend: Called when the device is going to be suspended by the system.
913 * @resume: Called when the device is being resumed by the system.
914 * @reset_resume: Called when the suspended device has been reset instead
915 *      of being resumed.
916 * @pre_reset: Called by usb_reset_composite_device() when the device
917 *      is about to be reset.
918 * @post_reset: Called by usb_reset_composite_device() after the device
919 *      has been reset, or in lieu of @resume following a reset-resume
920 *      (i.e., the device is reset instead of being resumed, as might
921 *      happen if power was lost).  The second argument tells which is
922 *      the reason.
923 * @id_table: USB drivers use ID table to support hotplugging.
924 *      Export this with MODULE_DEVICE_TABLE(usb,...).  This must be set
925 *      or your driver's probe function will never get called.
926 * @dynids: used internally to hold the list of dynamically added device
927 *      ids for this driver.
928 * @drvwrap: Driver-model core structure wrapper.
929 * @no_dynamic_id: if set to 1, the USB core will not allow dynamic ids to be
930 *      added to this driver by preventing the sysfs file from being created.
931 * @supports_autosuspend: if set to 0, the USB core will not allow autosuspend
932 *      for interfaces bound to this driver.
933 *
```

```
934 * USB interface drivers must provide a name, probe() and disconnect()
935 * methods, and an id_table.  Other driver fields are optional.
936 *
937 * The id_table is used in hotplugging.  It holds a set of descriptors,
938 * and specialized data may be associated with each entry.  That table
939 * is used by both user and kernel mode hotplugging support.
940 *
941 * The probe() and disconnect() methods are called in a context where
942 * they can sleep, but they should avoid abusing the privilege.  Most
943 * work to connect to a device should be done when the device is opened,
944 * and undone at the last close.  The disconnect code needs to address
945 * concurrency issues with respect to open() and close() methods, as
946 * well as forcing all pending I/O requests to complete (by unlinking
947 * them as necessary, and blocking until the unlinks complete).
948 */
949struct usb_driver {
950       const char *name;
951
952       int (*probe) (struct usb_interface *intf,
953                     const struct usb_device_id *id);
954
955       void (*disconnect) (struct usb_interface *intf);
956
957       int (*ioctl) (struct usb_interface *intf, unsigned int code,
958                       void *buf);
959
960       int (*suspend) (struct usb_interface *intf, pm_message_t message);
961       int (*resume) (struct usb_interface *intf);
962       int (*reset_resume)(struct usb_interface *intf);
963
964       int (*pre_reset)(struct usb_interface *intf);
965       int (*post_reset)(struct usb_interface *intf);
966
967       const struct usb_device_id *id_table;
968
969       struct usb_dynids dynids;
970       struct usbdrv_wrap drvwrap;
971       unsigned int no_dynamic_id:1;
972       unsigned int supports_autosuspend:1;
973};
```

El significado de sus campos:

- **struct module *owner**

  Puntero al propietario de este manejador.
- **const char *name**

  Puntero al nombre del manejador.
- **const struct usb_device_id *id_table**

  Puntero a la estructura **usb_device_id,** tabla que contiene una lista de todos los diferentes tipos de dispositivos USB que este manejador puede controlar. Si esta variable no se establece, la función de búsqueda nunca es llamada. Si se quiere que un manejador siempre sea llamado por cada dispositivo en el sistema, se debe crear una entrada en el campo **driver_info**:

  ```
  static struct usb_device_id usb_ids[  ] = {
          {.driver_info = 42},
  ```

```
        { }
};
```

- **int (*probe) (struct usb_interface *intf, const struct usb_device_id *id)**
  Puntero a la función que interroga al manejador USB. Esta función es llamada por el USB core cuando se piensa que posee una estructura **usb_interface** que este controlador puede manejar.
- **void (*disconnect) (struct usb_interface *intf)**

  Puntero a la función de desconexión del controlador USB. Esta función es llamada por el USB core cuando la estructura **usb_interface** ha sido eliminada del sistema.

  Por lo tanto, para crear un valor **struct usb_driver** , sólo cinco de los campos tienen que ser inicializados:

```
static struct usb_driver skel_driver = {
        .owner = THIS_MODULE,
        .name = "skeleton",
        .id_table = skel_table,
        .probe = skel_probe,
        .disconnect = skel_disconnect,
};
```

La estructura usb_driver contiene un par de funciones callbacks, que generalmente no se utilizan mucho y no se requieren para que un manejador USB pueda funcionar correctamente:

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

  Puntero a una función ioctl, permite a una aplicación controlar o comunicarse con un manejador de dispositivo en el manejador USB. En la práctica, sólo el manejador hub USB utiliza esta función.
- **int (*suspend) (struct usb_interface *intf, u32 state)**

  Puntero a la función que se encarga de suspender el manejador USB. Se llama cuando el USB core ha suspendido el dispositivo.
- **int (*resume) (struct usb_interface *intf)**

  Puntero a la función reanudar el manejador USB.

Registrar un manejador USB

Para registrar la estructura usb_driver con el USB core, una llamada a **usb_register_driver** se realiza con un puntero a la estructura **usb_driver**. Esto se realiza normalmente en el módulo de inicialización de código del manejador USB:

```
static int _ _init usb_skel_init(void){

        int result;
```

```
        /* register this driver with the USB subsystem */
        result = usb_register(&skel_driver);
        if (result)
        err("usb_register failed. Error number %d", result);

        return result;
}
```

Cuando el manejador USB se deshabilite, la estructura **usb_driver** tiene que ser deshabilitada desde el núcleo. Esto se hace con una llamada a usb_deregister_driver. Cuando esta llamada ocurre, cualquier interfaz USB que se encuentre actualmente vinculados a este manejador se desconectaran y la función disconnect sera llamada por ellos.

```
static void _ _exit usb_skel_exit(void){
        /* deregister this driver with the USB subsystem */
        usb_deregister(&skel_driver);
}
```

Envió y control de una Urb para un tipo de transmisión masivo

Cuando el manejador quiere enviar datos al dispositivo USB, debe crear una estructura urb para la transmisión de los datos:

drivers/usb/core/usb.c

```
47/**
48 * usb_alloc_urb - creates a new urb for a USB driver to use
49 * @iso_packets: number of iso packets for this urb
50 * @mem_flags: the type of memory to allocate, see kmalloc() for a list of
51 *      valid options for this.
52 *
53 * Creates an urb for the USB driver to use, initializes a few internal
54 * structures, incrementes the usage counter, and returns a pointer to it.
55 *
56 * If no memory is available, NULL is returned.
57 *
58 * If the driver want to use this urb for interrupt, control, or bulk
59 * endpoints, pass '0' as the number of iso packets.
60 *
61 * The driver must call usb_free_urb() when it is finished with the urb.
62 */
63struct urb *usb_alloc_urb(int iso_packets, gfp_t mem_flags)
64{
65       struct urb *urb;
66
67       urb = kmalloc(sizeof(struct urb) +
68               iso_packets * sizeof(struct usb_iso_packet_descriptor),
69               mem_flags);
70       if (!urb) {
71               err("alloc_urb: kmalloc failed");
72               return NULL;
73       }
74       usb_init_urb(urb);
```

```
 75        return urb;
```

Después de que una urb se ha creado con éxito, se debe crear también un buffer DMA para enviar los datos al dispositivo de manera más eficiente. Y los datos se copian desde el usuario al manejador en el buffer:

```
623/**
624 * usb_buffer_alloc - allocate dma-consistent buffer for URB_NO_xxx_DMA_MAP
625 * @dev: device the buffer will be used with
626 * @size: requested buffer size
627 * @mem_flags: affect whether allocation may block
628 * @dma: used to return DMA address of buffer
629 *
630 * Return value is either null (indicating no buffer could be allocated), or
631 * the cpu-space pointer to a buffer that may be used to perform DMA to the
632 * specified device.  Such cpu-space buffers are returned along with the DMA
633 * address (through the pointer provided).
634 *
635 * These buffers are used with URB_NO_xxx_DMA_MAP set in urb->transfer_flags
636 * to avoid behaviors like using "DMA bounce buffers", or thrashing IOMMU
637 * hardware during URB completion/resubmit.  The implementation varies between
638 * platforms, depending on details of how DMA will work to this device.
639 * Using these buffers also eliminates cacheline sharing problems on
640 * architectures where CPU caches are not DMA-coherent.  On systems without
641 * bus-snooping caches, these buffers are uncached.
642 *
643 * When the buffer is no longer used, free it with usb_buffer_free().
644 */
645void *usb_buffer_alloc(struct usb_device *dev, size_t size, gfp_t mem_flags,
646                        dma_addr_t *dma)
647{
648        if (!dev || !dev->bus)
649                return NULL;
650        return hcd_buffer_alloc(dev->bus, size, mem_flags, dma);
651}
```

Una vez que los datos están copiados correctamente desde el espacio de usuario al buffer local, la urb debe de ser inicializada correctamente antes de que pueda ser enviada al dispositivo USB, veamos el código en include/linux/usb.h

```
/**
1337 * usb_fill_bulk_urb - macro to help initialize a bulk urb
1338 * @urb: pointer to the urb to initialize.
1339 * @dev: pointer to the struct usb_device for this urb.
1340 * @pipe: the endpoint pipe
1341 * @transfer_buffer: pointer to the transfer buffer
1342 * @buffer_length: length of the transfer buffer
1343 * @complete_fn: pointer to the usb_complete_t function
1344 * @context: what to set the urb context to.
1345 *
1346 * Initializes a bulk urb with the proper information needed to submit it
1347 * to a device.
1348 */
1349static inline void usb_fill_bulk_urb(struct urb *urb,
1350                                     struct usb_device *dev,
```

```
1351                        unsigned int pipe,
1352                        void *transfer_buffer,
1353                        int buffer_length,
1354                        usb_complete_t complete_fn,
1355                        void *context)
1356{
1357        urb->dev = dev;
1358        urb->pipe = pipe;
1359        urb->transfer_buffer = transfer_buffer;
1360        urb->transfer_buffer_length = buffer_length;
1361        urb->complete = complete_fn;
1362        urb->context = context;
1363}
```

Ahora que la estructura urb esta correctamente inicializada, y los datos copiados correctamente, es cuando la urb puede ser enviada al USB core para que sea enviado al dispositivo.

retval = usb_submit_urb(urb, GFP_KERNEL);

Después de que la urb se ha transmitido correctamente hacia el dispositivo USB, una última llamada tiene lugar para terminar la transmisión. Para el manejador drivers/usb/usb-skeleton.c, se llama a la función skel_write_bulk_callback

```
211static void skel_write_bulk_callback(struct urb *urb)
212{
213        struct usb_skel *dev;
214
215        dev = (struct usb_skel *)urb->context;
216
217        /* sync/async unlink faults aren't errors */
218        if (urb->status) {
219                if(!(urb->status == -ENOENT ||
220                    urb->status == -ECONNRESET ||
221                    urb->status == -ESHUTDOWN))
222                        err("%s - nonzero write bulk status received: %d",
223                            __FUNCTION__, urb->status);
224
225                spin_lock(&dev->err_lock);
226                dev->errors = urb->status;
227                spin_unlock(&dev->err_lock);
228        }
229
230        /* free up our allocated buffer */
231        usb_buffer_free(urb->dev, urb->transfer_buffer_length,
232                        urb->transfer_buffer, urb->transfer_dma);
233        up(&dev->limit_sem);
234}
```

Lo primero que hace la función es comprobar el estado de la urb para

determinar si el urb ha sido completada con éxito, (los valores de errores, **-ENOENT**, -**ECONNRESET** y **-ESHUTDOWN** no son verdaderos errores de transmisión, sólo los informes sobre el éxito de la transmisión), luego libera los buffers.

Transferencias Usb sin urbs

A veces un manejador USB para enviar unos pocos datos no quiere pasar por todos los pasos de crear una estructura urb, iniciarla y luego esperar a que terminen las funciones de preparar la estructura.

Dos funciones están disponibles que nos posibilitan una transmisión sencilla.

usb_bulk_msg

Crea una estructura urb para un tipo masivo (bulk) y lo envía al dispositivo especificado y luego espera a que termine antes de regresar al que llama. Se define en: /linux/drivers/usb/core/message.c

```
189/**
190 * usb_bulk_msg - Builds a bulk urb, sends it off and waits for completion
191 * @usb_dev: pointer to the usb device to send the message to
192 * @pipe: endpoint "pipe" to send the message to
193 * @data: pointer to the data to send
194 * @len: length in bytes of the data to send
195 * @actual_length: pointer to a location to put the actual length transferred
196 *      in bytes
197 * @timeout: time in msecs to wait for the message to complete before
198 *      timing out (if 0 the wait is forever)
199 *
200 * Context: !in_interrupt ()
201 *
202 * This function sends a simple bulk message to a specified endpoint
203 * and waits for the message to complete, or timeout.
204 *
205 * If successful, it returns 0, otherwise a negative error number.  The number
206 * of actual bytes transferred will be stored in the actual_length paramater.
207 *
208 * Don't use this function from within an interrupt context, like a bottom half
209 * handler.  If you need an asynchronous message, or need to send a message
210 * from within interrupt context, use usb_submit_urb() If a thread in your
211 * driver uses this call, make sure your disconnect() method can wait for it to
212 * complete.  Since you don't have a handle on the URB used, you can't cancel
213 * the request.
214 *
215 * Because there is no usb_interrupt_msg() and no USBDEVFS_INTERRUPT ioctl,
216 * users are forced to abuse this routine by using it to submit URBs for
217 * interrupt endpoints.  We will take the liberty of creating an interrupt URB
218 * (with the default interval) if the target is an interrupt endpoint.
219 */
220int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe,
221                 void *data, int len, int *actual_length, int timeout)
222{
223         struct urb *urb;
224         struct usb_host_endpoint *ep;
225
226         ep = (usb_pipein(pipe) ? usb_dev->ep_in : usb_dev->ep_out)
227                         [usb_pipeendpoint(pipe)];
```

```
228     if (!ep || len < 0)
229             return -EINVAL;
230
231     urb = usb_alloc_urb(0, GFP_KERNEL);
232     if (!urb)
233             return -ENOMEM;
234
235     if ((ep->desc.bmAttributes & USB_ENDPOINT_XFERTYPE_MASK) ==
236                     USB_ENDPOINT_XFER_INT) {
237             pipe = (pipe & ~(3 << 30)) | (PIPE_INTERRUPT << 30);
238             usb_fill_int_urb(urb, usb_dev, pipe, data, len,
239                             usb_api_blocking_completion, NULL,
240                             ep->desc.bInterval);
241     } else
242             usb_fill_bulk_urb(urb, usb_dev, pipe, data, len,
243                             usb_api_blocking_completion, NULL);
244
245     return usb_start_wait_urb(urb, timeout, actual_length);
246}
```

Los parámetros de esta función son los siguientes:

- **struct usb_device * usb_dev**

  Un puntero al dispositivo USB para enviar el mensaje.

- **unsigned int pipe**

  El endpoint específico de los dispositivo USB para que este mensaje pueda ser enviado. Este valor se crea con una llamada a cualquiera de **usb_sndbulkpipe** o **usb_rcvbulkpipe**.

- **void *data**

  Un puntero a los datos a enviar.

- **int len**

  La longitud del buffer que apunta el parámetro de datos.

- **int *actual_length**

  Un puntero al lugar donde la función almacena el número de bytes transmitidos al dispositivo o recibidos del dispositivo.

- **int timeout**

  La cantidad de tiempo que debe esperar antes de que se agote el tiempo de espera. Si este valor es 0, la función siempre espera el mensaje al completo.

Si la función se termina con éxito devuelve 0, en otro caso un valor negativo.

usb_control_msg

Para transmitir pocos datos en un tipo control, existen funciones para permitir a un manejador enviar y recibir mensajes de control USB. La función

usb_control_msg funciona de forma parecida a la función usb_bulk_msg:

```
int usb_control_msg(struct usb_device *dev, unsigned int pipe, __u8 request,
 130                __u8 requesttype, __u16 value, __u16 index, void *data,
 131                __u16 size, int timeout)
 132{
 133        struct usb_ctrlrequest *dr;
 134        int ret;
 135
 136        dr = kmalloc(sizeof(struct usb_ctrlrequest), GFP_NOIO);
 137        if (!dr)
 138                return -ENOMEM;
 139
 140        dr->bRequestType = requesttype;
 141        dr->bRequest = request;
 142        dr->wValue = cpu_to_le16p(&value);
 143        dr->wIndex = cpu_to_le16p(&index);
 144        dr->wLength = cpu_to_le16p(&size);
 145
 146        /* dbg("usb_control_msg"); */
 147
 148        ret = usb_internal_control_msg(dev, pipe, dr, data, size, timeout);
 149
 150        kfree(dr);
 151
 152        return ret;
 153}
```

Los parámetros de esta función son casi los mismos que **usb_bulk_msg**, con algunas diferencias importantes:

- **struct usb_device *dev**

  Un puntero al dispositivo USB para enviar el mensaje de control.

- **unsigned int pipe**

  El endpoint específico del dispositivo USB donde este mensaje de control va a ser enviado. Este valor se crea con una llamada a cualquiera de **usb_sndctrlpipe** o **usb_rcvctrlpipe**.

- **_ _u8 request**

  El valor de la solicitud para el mensaje de control.

- **_ _u8 requesttype**

  El tipo de la solicitud para el mensaje de control.

- **_ _u16 value**

  El valor del mensaje USB para el mensaje de control.

- **_ _u16 index**

  El índice del mensaje USB para mensaje de control.

- **void *data**

  Un puntero a los datos a enviar al dispositivo, si es un OUT endpoint.

- **_ _u16 size**

El tamaño del buffer que indica el puntero a datos.

- **int timeout**

  La cantidad de tiempo que debe esperar antes de que se agote el tiempo de espera. Si este valor es 0, la función siempre espera el mensaje al completo.

Si la función tiene éxito, devuelve el número de bytes que fueron transferidos hacia o desde el dispositivo. Si no tiene éxito, devuelve un número de error negativo.

Otras funciones para USB

Una serie de funciones de ayuda en el USB core, pueden ser utilizadas para obtener información estándar de un dispositivo USB.

La función usb_get_descriptor recupera el descriptor USB de un dispositivo especificado. La función se define en: /linux/drivers/usb/core/message.c

```
int usb_get_descriptor(struct usb_device *dev, unsigned char type,
                    unsigned char index, void *buf, int size);
```

Esta función puede ser utilizada por un manejador USB para leer dentro de la estructura **struct usb_device** cualquiera de los campos descriptores del dispositivo. Los parámetros de la función son los siguientes:

- **struct usb_device *usb_dev**

  Un puntero al descriptor de dispositivo USB que debe ser leído.

- **unsigned char type**

  Este tipo se describe en la especificación USB y puede ser uno de los siguientes tipos:

  - SB_DT_DEVICE
  - USB_DT_CONFIG
  - USB_DT_STRING
  - USB_DT_INTERFACE
  - USB_DT_ENDPOINT
  - USB_DT_DEVICE_QUALIFIER
  - USB_DT_OTHER_SPEED_CONFIG
  - USB_DT_INTERFACE_POWER
  - USB_DT_OTG
  - USB_DT_DEBUG
  - USB_DT_INTERFACE_ASSOCIATION

- USB_DT_CS_DEVICE
- USB_DT_CS_CONFIG
- USB_DT_CS_STRING
- USB_DT_CS_INTERFACE
- USB_DT_CS_ENDPOINT

- **unsigned char index**

  El número del descriptor que debe ser recuperado del dispositivo.

- **void *buf**

  Un puntero al buffer para poder copiar el descriptor.

- **int size**

  El tamaño de la memoria apuntado por la variable buf.

```
613/**
614 * usb_get_descriptor - issues a generic GET_DESCRIPTOR request
615 * @dev: the device whose descriptor is being retrieved
616 * @type: the descriptor type (USB_DT_*)
617 * @index: the number of the descriptor
618 * @buf: where to put the descriptor
619 * @size: how big is "buf"?
620 * Context: !in_interrupt ()
621 *
622 * Gets a USB descriptor.  Convenience functions exist to simplify
623 * getting some types of descriptors.  Use
624 * usb_get_string() or usb_string() for USB_DT_STRING.
625 * Device (USB_DT_DEVICE) and configuration descriptors (USB_DT_CONFIG)
626 * are part of the device structure.
627 * In addition to a number of USB-standard descriptors, some
628 * devices also use class-specific or vendor-specific descriptors.
629 *
630 * This call is synchronous, and may not be used in an interrupt context.
631 *
632 * Returns the number of bytes received on success, or else the status code
633 * returned by the underlying usb_control_msg() call.
634 */
635int usb_get_descriptor(struct usb_device *dev, unsigned char type,
636                 unsigned char index, void *buf, int size)
637{
638     int i;
639     int result;
640
641     memset(buf, 0, size);   /* Make sure we parse really received data */
642
643     for (i = 0; i < 3; ++i) {
644             /* retry on length 0 or error; some devices are flakey */
645             result = usb_control_msg(dev, usb_rcvctrlpipe(dev, 0),
646                             USB_REQ_GET_DESCRIPTOR, USB_DIR_IN,
647                             (type << 8) + index, 0, buf, size,
648                             USB_CTRL_GET_TIMEOUT);
649             if (result <= 0 && result != -ETIMEDOUT)
650                     continue;
651             if (result > 1 && ((u8 *)buf)[1] != type) {
652                     result = -EPROTO;
```

```
653                     continue;
654             }
655             break;
656     }
657     return result;
658}
659
```

Si esta función tiene éxito, devuelve el número de bytes leídos del dispositivo. De lo contrario, devuelve un número negativo.

Uno de los usos más comunes para la llamada **usb_get_descriptor** es para recuperar una cadena desde el dispositivo USB. Debido a que este es bastante común, existe una función para tal fin llamada **usb_get_string**:

```
661/**
662 * usb_get_string - gets a string descriptor
663 * @dev: the device whose string descriptor is being retrieved
664 * @langid: code for language chosen (from string descriptor zero)
665 * @index: the number of the descriptor
666 * @buf: where to put the string
667 * @size: how big is "buf"?
668 * Context: !in_interrupt ()
669 *
670 * Retrieves a string, encoded using UTF-16LE (Unicode, 16 bits per character,
671 * in little-endian byte order).
672 * The usb_string() function will often be a convenient way to turn
673 * these strings into kernel-printable form.
674 *
675 * Strings may be referenced in device, configuration, interface, or other
676 * descriptors, and could also be used in vendor-specific ways.
677 *
678 * This call is synchronous, and may not be used in an interrupt context.
679 *
680 * Returns the number of bytes received on success, or else the status code
681 * returned by the underlying usb_control_msg() call.
682 */
683static int usb_get_string(struct usb_device *dev, unsigned short langid,
684                         unsigned char index, void *buf, int size)
685{
686     int i;
687     int result;
688
689     for (i = 0; i < 3; ++i) {
690             /* retry on length 0 or stall; some devices are flakey */
691             result = usb_control_msg(dev, usb_rcvctrlpipe(dev, 0),
692                     USB_REQ_GET_DESCRIPTOR, USB_DIR_IN,
693                     (USB_DT_STRING << 8) + index, langid, buf, size,
694                     USB_CTRL_GET_TIMEOUT);
695             if (!(result == 0 || result == -EPIPE))
696                     break;
697     }
698     return result;
699}
```

Si tiene éxito, esta función devuelve el número de bytes recibidos por el dispositivo. De lo contrario, devuelve un número negativo de error.

Recursos

- www.lrr.in.tum.de/Par/arch/usb/usbdoc/
- marc.info/?l=linux-usb-announce&m=92328122826889&w=2
- tali.admingilde.org/linux-docbook/usb/index.html
- www.linuxjournal.com/article/4786
- book.chinaunix.net/special//ebook/oreilly/ldd3/0596005903/linuxdrive3-CHP-13-SECT-3.html
- www.linux-usb.org