Revised: 2000-Dec-05.
Again: 2002-Jul-06
Again: 2005-Sep-19

NOTE:

The USB subsystem now has a substantial section in "The Linux Kernel API" guide (in Documentation/DocBook), generated from the current source code. This particular documentation file isn't particularly current or complete; don't rely on it except for a quick overview.

1.1. Basic concept or 'What is an URB?'

The basic idea of the new driver is message passing, the message itself is called USB Request Block, or URB for short.

- An URB consists of all relevant information to execute any USB transaction and deliver the data and status back.

- Execution of an URB is inherently an asynchronous operation, i.e. the usb_submit_urb(urb) call returns immediately after it has successfully queued the requested action.

- Transfers for one URB can be canceled with usb_unlink_urb(urb) at any time.

- Each URB has a completion handler, which is called after the action has been successfully completed or canceled. The URB also contains a context-pointer for passing information to the completion handler.

- Each endpoint for a device logically supports a queue of requests. You can fill that queue, so that the USB hardware can still transfer data to an endpoint while your driver handles completion of another. This maximizes use of USB bandwidth, and supports seamless streaming of data to (or from) devices when using periodic transfer modes.

1.2. The URB structure

Some of the fields in an URB are:

```
struct urb
{
// (IN) device and pipe specify the endpoint queue
	struct usb_device *dev;         // pointer to associated USB device
	unsigned int pipe;              // endpoint information

	unsigned int transfer_flags;    // ISO_ASAP, SHORT_NOT_OK, etc.

// (IN) all urbs need completion routines
	void *context;                  // context for completion routine
	void (*complete)(struct urb *); // pointer to completion routine

// (OUT) status after each completion
	int status;                     // returned status
```

```
// (IN) buffer used for data transfers
        void *transfer_buffer;          // associated data buffer
        int transfer_buffer_length;     // data buffer length
        int number_of_packets;          // size of iso_frame_desc

// (OUT) sometimes only part of CTRL/BULK/INTR transfer_buffer is used
        int actual_length;              // actual data buffer length

// (IN) setup stage for CTRL (pass a struct usb_ctrlrequest)
        unsigned char* setup_packet;    // setup packet (control only)

// Only for PERIODIC transfers (ISO, INTERRUPT)
    // (IN/OUT) start_frame is set unless ISO_ASAP isn't set
        int start_frame;                // start frame
        int interval;                   // polling interval

    // ISO only: packets are only "best effort"; each can have errors
        int error_count;                // number of errors
        struct usb_iso_packet_descriptor iso_frame_desc[0];
};
```

Your driver must create the "pipe" value using values from the appropriate endpoint descriptor in an interface that it's claimed.

1.3. How to get an URB?

URBs are allocated with the following call

```
        struct urb *usb_alloc_urb(int isoframes, int mem_flags)
```

Return value is a pointer to the allocated URB, 0 if allocation failed. The parameter isoframes specifies the number of isochronous transfer frames you want to schedule. For CTRL/BULK/INT, use 0. The mem_flags parameter holds standard memory allocation flags, letting you control (among other things) whether the underlying code may block or not.

To free an URB, use

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

You may free an urb that you've submitted, but which hasn't yet been returned to you in a completion callback. It will automatically be deallocated when it is no longer in use.

1.4. What has to be filled in?

Depending on the type of transaction, there are some inline functions defined in <linux/usb.h> to simplify the initialization, such as fill_control_urb() and fill_bulk_urb(). In general, they need the usb device pointer, the pipe (usual format from usb.h), the transfer buffer, the desired transfer length, the completion handler, and its context. Take a look at the some existing drivers to see how they're used.

Flags:

For ISO there are two startup behaviors: Specified start_frame or ASAP.
For ASAP set URB_ISO_ASAP in transfer_flags.

If short packets should NOT be tolerated, set URB_SHORT_NOT_OK in
transfer_flags.

1.5. How to submit an URB?

Just call

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

The mem_flags parameter, such as SLAB_ATOMIC, controls memory allocation,
such as whether the lower levels may block when memory is tight.

It immediately returns, either with status 0 (request queued) or some
error code, usually caused by the following:

- Out of memory (-ENOMEM)
- Unplugged device (-ENODEV)
- Stalled endpoint (-EPIPE)
- Too many queued ISO transfers (-EAGAIN)
- Too many requested ISO frames (-EFBIG)
- Invalid INT interval (-EINVAL)
- More than one packet for INT (-EINVAL)

After submission, urb->status is -EINPROGRESS; however, you should never
look at that value except in your completion callback.

For isochronous endpoints, your completion handlers should (re)submit
URBs to the same endpoint with the ISO_ASAP flag, using multi-buffering,
to get seamless ISO streaming.

1.6. How to cancel an already running URB?

There are two ways to cancel an URB you've submitted but which hasn't
been returned to your driver yet. For an asynchronous cancel, call

```
	int usb_unlink_urb(struct urb *urb)
```

It removes the urb from the internal list and frees all allocated
HW descriptors. The status is changed to reflect unlinking. Note
that the URB will not normally have finished when usb_unlink_urb()
returns; you must still wait for the completion handler to be called.

To cancel an URB synchronously, call

```
	void usb_kill_urb(struct urb *urb)
```

It does everything usb_unlink_urb does, and in addition it waits
until after the URB has been returned and the completion handler
has finished. It also marks the URB as temporarily unusable, so
that if the completion handler or anyone else tries to resubmit it
they will get a -EPERM error. Thus you can be sure that when

usb_kill_urb() returns, the URB is totally idle.

1.7. What about the completion handler?

The handler is of the following type:

```
        typedef void (*usb_complete_t)(struct urb *, struct pt_regs *)
```

I.e., it gets the URB that caused the completion call, plus the register values at the time of the corresponding interrupt (if any). In the completion handler, you should have a look at urb->status to detect any USB errors. Since the context parameter is included in the URB, you can pass information to the completion handler.

Note that even when an error (or unlink) is reported, data may have been transferred. That's because USB transfers are packetized; it might take sixteen packets to transfer your 1KByte buffer, and ten of them might have transferred successfully before the completion was called.

NOTE: ***** WARNING *****
NEVER SLEEP IN A COMPLETION HANDLER. These are normally called during hardware interrupt processing. If you can, defer substantial work to a tasklet (bottom half) to keep system latencies low. You'll probably need to use spinlocks to protect data structures you manipulate in completion handlers.

1.8. How to do isochronous (ISO) transfers?

For ISO transfers you have to fill a usb_iso_packet_descriptor structure, allocated at the end of the URB by usb_alloc_urb(n,mem_flags), for each packet you want to schedule. You also have to set urb->interval to say how often to make transfers; it's often one per frame (which is once every microframe for highspeed devices). The actual interval used will be a power of two that's no bigger than what you specify.

The usb_submit_urb() call modifies urb->interval to the implemented interval value that is less than or equal to the requested interval value. If ISO_ASAP scheduling is used, urb->start_frame is also updated.

For each entry you have to specify the data offset for this frame (base is transfer_buffer), and the length you want to write/expect to read. After completion, actual_length contains the actual transferred length and status contains the resulting status for the ISO transfer for this frame. It is allowed to specify a varying length from frame to frame (e.g. for audio synchronisation/adaptive transfer rates). You can also use the length 0 to omit one or more frames (striping).

For scheduling you can choose your own start frame or ISO_ASAP. As explained earlier, if you always keep at least one URB queued and your completion keeps (re)submitting a later URB, you'll get smooth ISO streaming (if usb bandwidth utilization allows).

If you specify your own start frame, make sure it's several frames in advance

of the current frame. You might want this model if you're synchronizing
ISO data with some other event stream.

1.9. How to start interrupt (INT) transfers?

Interrupt transfers, like isochronous transfers, are periodic, and happen in intervals that are powers of two (1, 2, 4 etc) units. Units are frames for full and low speed devices, and microframes for high speed ones.
The usb_submit_urb() call modifies urb->interval to the implemented interval value that is less than or equal to the requested interval value.

In Linux 2.6, unlike earlier versions, interrupt URBs are not automagically restarted when they complete. They end when the completion handler is called, just like other URBs. If you want an interrupt URB to be restarted, your completion handler must resubmit it.