USB Request Block (URB)

Revised: 2000-Dec-05
Again: 2002-Jul-06
Again: 2005-Sep-19
Again: 2017-Mar-29

> **Note**
>
> The USB subsystem now has a substantial section at :ref:`usb-hostside-api` section, generated from the current source code. This particular documentation file isn't complete and may not be updated to the last version; don't rely on it except for a quick overview.
>
> > **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 14); *backlink*
> >
> > Unknown interpreted text role "ref".

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 :c:func:`usb_submit_urb` call returns immediately after it has successfully queued the requested action.

  > **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 29); *backlink*
  >
  > Unknown interpreted text role "c:func".

- Transfers for one URB can be canceled with :c:func:`usb_unlink_urb` at any time.

  > **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 33); *backlink*
  >
  > Unknown interpreted text role "c:func".

- 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.

The URB structure

Some of the fields in struct 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;    // URB_ISO_ASAP, URB_SHORT_NOT_OK, etc.

// (IN) all urbs need completion routines
      void *context;                  // context for completion routine
      usb_complete_t complete;        // 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
      u32 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
      u32 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 URB_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.

How to get an URB?

URBs are allocated by calling :c:func:`usb_alloc_urb`:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 95); *backlink***
>
> Unknown interpreted text role "c:func".

```
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 :c:func:`usb_free_urb`:

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 105); *backlink***
>
> Unknown interpreted text role "c:func".

```
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.

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 :c:func:`usb_fill_control_urb`, :c:func:`usb_fill_bulk_urb` and :c:func:`usb_fill_int_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.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 117); *backlink***
>
> Unknown interpreted text role "c:func".

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 117); *backlink***
>
> Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 117); *backlink*

Unknown interpreted text role "c:func".

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.

How to submit an URB?

Just call :c:func:`usb_submit_urb`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 137); *backlink*

Unknown interpreted text role "c:func".

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

The `mem_flags` parameter, such as `GFP_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 `URB_ISO_ASAP` flag, using multi-buffering, to get seamless ISO streaming.

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 :c:func:`usb_unlink_urb`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 166); *backlink*

Unknown interpreted text role "c:func".

```
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 :c:func:`usb_unlink_urb` returns; you must still wait for the completion handler to be called.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 172); *backlink*

Unknown interpreted text role "c:func".

To cancel an URB synchronously, call :c:func:`usb_kill_urb`:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 177); *backlink*

Unknown interpreted text role "c:func".

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

It does everything :c:func:`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 :c:func:`usb_kill_urb` returns, the URB is totally idle.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 181); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 181); *backlink*

Unknown interpreted text role "c:func".

There is a lifetime issue to consider. An URB may complete at any time, and the completion handler may free the URB. If this happens while :c:func:`usb_unlink_urb` or :c:func:`usb_kill_urb` is running, it will cause a memory-access violation. The driver is responsible for avoiding this, which often means some sort of lock will be needed to prevent the URB from being deallocated while it is still in use.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 188); *backlink*

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 188); *backlink*

Unknown interpreted text role "c:func".

On the other hand, since usb_unlink_urb may end up calling the completion handler, the handler must not take any lock that is held when usb_unlink_urb is invoked. The general solution to this problem is to increment the URB's reference count while holding the lock, then drop the lock and call usb_unlink_urb or usb_kill_urb, and then decrement the URB's reference count. You increment the reference count by calling :c:func`usb_get_urb`:

```
struct urb *usb_get_urb(struct urb *urb)
```

(ignore the return value; it is the same as the argument) and decrement the reference count by calling :c:func:`usb_free_urb`. Of course, none of this is necessary if there's no danger of the URB being freed by the completion handler.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 205); *backlink*

Unknown interpreted text role "c:func".

What about the completion handler?

The handler is of the following type:

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

I.e., it gets the URB that caused the completion call. 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.

> **Warning**
>
> NEVER SLEEP IN A COMPLETION HANDLER.
>
> These are often called in atomic context.

In the current kernel, completion handlers run with local interrupts disabled, but in the future this will be changed, so don't assume that local IRQs are always disabled inside completion handlers.

How to do isochronous (ISO) transfers?

Besides the fields present on a bulk transfer, for ISO, 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. You can use the :c:func:`usb_fill_int_urb` macro to fill most ISO transfer fields.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 242); *backlink***
>
> Unknown interpreted text role "c:func".

For ISO transfers you also have to fill a :c:type:`usb_iso_packet_descriptor` structure, allocated at the end of the URB by :c:func:`usb_alloc_urb`, for each packet you want to schedule.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 249); *backlink***
>
> Unknown interpreted text role "c:type".

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 249); *backlink***
>
> Unknown interpreted text role "c:func".

The :c:func:`usb_submit_urb` call modifies `urb->interval` to the implemented interval value that is less than or equal to the requested interval value. If `URB_ISO_ASAP` scheduling is used, `urb->start_frame` is also updated.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 253); *backlink***
>
> Unknown interpreted text role "c:func".

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 `URB_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.

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. You can use the :c:func:`usb_fill_int_urb` macro to fill INT transfer fields.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 278); *backlink***

Unknown interpreted text role "c:func".

The :c:func:`usb_submit_urb` call modifies `urb->interval` to the implemented interval value that is less than or equal to the requested interval value.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\usb\[linux-master][Documentation][driver-api][usb]URB.rst, line 283); *backlink*

Unknown interpreted text role "c:func".

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. s