# https://www.kernel.org/doc/html/v5.4/driver-api/usb/URB.html

# USB Request Block (URB)

**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()](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html" \l "c.usb_submit_urb" \o "usb_submit_urb)** call returns immediately after it has successfully queued the requested action.
* Transfers for one URB can be canceled with **[usb\_unlink\_urb()](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html" \l "c.usb_unlink_urb" \o "usb_unlink_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.

Основная идея нового драйвера - передача сообщений, само сообщение называется USB Request Block, или для краткости URB.

• URB состоит из всей соответствующей информации для выполнения любой транзакции USB и доставки данных и статуса обратно.

• Выполнение URB по своей сути является асинхронной операцией, то есть вызов usb\_submit\_urb () возвращается сразу после того, как он успешно поставил в очередь запрашиваемое действие.

• Передачи для одного URB могут быть отменены с помощью usb\_unlink\_urb () в любое время.

• Каждый URB имеет обработчик завершения, который вызывается после того, как действие было успешно завершено или отменено. URB также содержит указатель контекста для передачи информации обработчику завершения.

• Каждая конечная точка для устройства логически поддерживает очередь запросов. Вы можете заполнить эту очередь, чтобы оборудование USB все еще могло передавать данные в конечную точку, пока ваш драйвер обрабатывает завершение другой. Это максимизирует использование полосы пропускания USB и поддерживает плавную потоковую передачу данных на (или с) устройства при использовании периодических режимов передачи.

**The URB structure**

Some of the fields in struct **[urb](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html" \l "c.urb" \o "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 **[usb\_alloc\_urb()](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html" \l "c.usb_alloc_urb" \o "usb_alloc_urb)**:

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.

Возвращаемое значение - указатель на выделенный URB, 0, если выделение не удалось. Параметр isoframes указывает количество кадров изохронной передачи, которые вы хотите запланировать. Для CTRL / BULK / INT используйте 0. Параметр mem\_flags содержит стандартные флаги выделения памяти, что позволяет вам (помимо прочего) контролировать, может ли базовый код блокироваться или нет.

To free an URB, use [**usb\_free\_urb()**](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html#c.usb_free_urb):

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 [**usb\_fill\_control\_urb()**](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html#c.usb_fill_control_urb), [**usb\_fill\_bulk\_urb()**](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html#c.usb_fill_bulk_urb) and [**usb\_fill\_int\_urb()**](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html#c.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.

В зависимости от типа транзакции в linux / usb.h определены некоторые встроенные функции, упрощающие инициализацию, такие как usb\_fill\_control\_urb (), usb\_fill\_bulk\_urb () и usb\_fill\_int\_urb (). В общем, они нуждаются в указателе устройства usb, канале (обычный формат от usb.h), буфере передачи, желаемой длине передачи, обработчике завершения и его контексте. Посмотрите на некоторые существующие драйверы, чтобы увидеть, как они используются.

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 **[usb\_submit\_urb()](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html" \l "c.usb_submit_urb" \o "usb_submit_urb)**:

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 **[usb\_unlink\_urb()](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html" \l "c.usb_unlink_urb" \o "usb_unlink_urb)**:

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()](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html" \l "c.usb_unlink_urb" \o "usb_unlink_urb)** returns; you must still wait for the completion handler to be called.

To cancel an URB synchronously, call **[usb\_kill\_urb()](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html" \l "c.usb_kill_urb" \o "usb_kill_urb)**:

void usb\_kill\_urb(struct urb \*urb)

It does everything **[usb\_unlink\_urb()](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html" \l "c.usb_unlink_urb" \o "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()](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html" \l "c.usb_kill_urb" \o "usb_kill_urb)** returns, the URB is totally idle.

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 **[usb\_unlink\_urb()](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html" \l "c.usb_unlink_urb" \o "usb_unlink_urb)** or **[usb\_kill\_urb()](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html" \l "c.usb_kill_urb" \o "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.

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 **[usb\_free\_urb()](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html" \l "c.usb_free_urb" \o "usb_free_urb)**. Of course, none of this is necessary if there’s no danger of the URB being freed by the completion handler.

**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 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 **[usb\_fill\_int\_urb()](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html" \l "c.usb_fill_int_urb" \o "usb_fill_int_urb)** macro to fill most ISO transfer fields.

For ISO transfers you also have to fill a **usb\_iso\_packet\_descriptor** structure, allocated at the end of the URB by **[usb\_alloc\_urb()](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html" \l "c.usb_alloc_urb" \o "usb_alloc_urb)**, for each packet you want to schedule.

The **[usb\_submit\_urb()](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html" \l "c.usb_submit_urb" \o "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.

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 **[usb\_fill\_int\_urb()](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html" \l "c.usb_fill_int_urb" \o "usb_fill_int_urb)** macro to fill INT transfer fields.

The **[usb\_submit\_urb()](https://www.kernel.org/doc/html/v5.4/driver-api/usb/usb.html" \l "c.usb_submit_urb" \o "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. s

[Next](https://www.kernel.org/doc/html/v5.4/driver-api/usb/power-management.html)[Previous](https://www.kernel.org/doc/html/v5.4/driver-api/usb/dma.html)