In Linux 2.5 kernels (and later), USB device drivers have additional control over how DMA may be used to perform I/O operations. The APIs are detailed in the kernel usb programming guide (kerneldoc, from the source code).

API OVERVIEW

The big picture is that USB drivers can continue to ignore most DMA issues, though they still must provide DMA-ready buffers (see Documentation/PCI/PCI-DMA-mapping.txt). That's how they've worked through the 2.4 (and earlier) kernels.

OR: they can now be DMA-aware.

- New calls enable DMA-aware drivers, letting them allocate dma buffers and manage dma mappings for existing dma-ready buffers (see below).

- URBs have an additional "transfer_dma" field, as well as a transfer_flags bit saying if it's valid. (Control requests also have "setup_dma", but drivers must not use it.)

- "usbcore" will map this DMA address, if a DMA-aware driver didn't do it first and set URB_NO_TRANSFER_DMA_MAP. HCDs don't manage dma mappings for URBs.

- There's a new "generic DMA API", parts of which are usable by USB device drivers. Never use dma_set_mask() on any USB interface or device; that would potentially break all devices sharing that bus.

ELIMINATING COPIES

It's good to avoid making CPUs copy data needlessly. The costs can add up, and effects like cache-trashing can impose subtle penalties.

- If you're doing lots of small data transfers from the same buffer all the time, that can really burn up resources on systems which use an IOMMU to manage the DMA mappings. It can cost MUCH more to set up and tear down the IOMMU mappings with each request than perform the I/O!

  For those specific cases, USB has primitives to allocate less expensive memory. They work like kmalloc and kfree versions that give you the right kind of addresses to store in urb->transfer_buffer and urb->transfer_dma. You'd also set URB_NO_TRANSFER_DMA_MAP in urb->transfer_flags:

```
        void *usb_alloc_coherent (struct usb_device *dev, size_t size,
                int mem_flags, dma_addr_t *dma);

        void usb_free_coherent (struct usb_device *dev, size_t size,
                void *addr, dma_addr_t dma);
```

  Most drivers should *NOT* be using these primitives; they don't need to use this type of memory ("dma-coherent"), and memory returned from kmalloc() will work just fine.

  The memory buffer returned is "dma-coherent"; sometimes you might need to

force a consistent memory access ordering by using memory barriers. It's not using a streaming DMA mapping, so it's good for small transfers on systems where the I/O would otherwise thrash an IOMMU mapping. (See Documentation/PCI/PCI-DMA-mapping.txt for definitions of "coherent" and "streaming" DMA mappings.)

Asking for 1/Nth of a page (as well as asking for N pages) is reasonably space-efficient.

On most systems the memory returned will be uncached, because the semantics of dma-coherent memory require either bypassing CPU caches or using cache hardware with bus-snooping support. While x86 hardware has such bus-snooping, many other systems use software to flush cache lines to prevent DMA conflicts.

- Devices on some EHCI controllers could handle DMA to/from high memory.

Unfortunately, the current Linux DMA infrastructure doesn't have a sane way to expose these capabilities ... and in any case, HIGHMEM is mostly a design wart specific to x86_32. So your best bet is to ensure you never pass a highmem buffer into a USB driver. That's easy; it's the default behavior. Just don't override it; e.g. with NETIF_F_HIGHDMA.

This may force your callers to do some bounce buffering, copying from high memory to "normal" DMA memory. If you can come up with a good way to fix this issue (for x86_32 machines with over 1 GByte of memory), feel free to submit patches.

WORKING WITH EXISTING BUFFERS

Existing buffers aren't usable for DMA without first being mapped into the DMA address space of the device. However, most buffers passed to your driver can safely be used with such DMA mapping. (See the first section of Documentation/PCI/PCI-DMA-mapping.txt, titled "What memory is DMA-able?")

- When you're using scatterlists, you can map everything at once. On some systems, this kicks in an IOMMU and turns the scatterlists into single DMA transactions:

```
int usb_buffer_map_sg (struct usb_device *dev, unsigned pipe,
        struct scatterlist *sg, int nents);

void usb_buffer_dmasync_sg (struct usb_device *dev, unsigned pipe,
        struct scatterlist *sg, int n_hw_ents);

void usb_buffer_unmap_sg (struct usb_device *dev, unsigned pipe,
        struct scatterlist *sg, int n_hw_ents);
```

It's probably easier to use the new usb_sg_*() calls, which do the DMA mapping and apply other tweaks to make scatterlist i/o be fast.

- Some drivers may prefer to work with the model that they're mapping large buffers, synchronizing their safe re-use. (If there's no re-use, then let usbcore do the map/unmap.) Large periodic transfers make good examples here, since it's cheaper to just synchronize the buffer than to unmap it

each time an urb completes and then re-map it on during resubmission.

These calls all work with initialized urbs: urb->dev, urb->pipe, urb->transfer_buffer, and urb->transfer_buffer_length must all be valid when these calls are used (urb->setup_packet must be valid too if urb is a control request):

```
struct urb *usb_buffer_map (struct urb *urb);

void usb_buffer_dmasync (struct urb *urb);

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

The calls manage urb->transfer_dma for you, and set URB_NO_TRANSFER_DMA_MAP so that usbcore won't map or unmap the buffer. They cannot be used for setup_packet buffers in control requests.

Note that several of those interfaces are currently commented out, since they don't have current users. See the source code. Other than the dmasync calls (where the underlying DMA primitives have changed), most of them can easily be commented back in if you want to use them.