

8 UNIVERSAL SERIAL BUS

ETRAX 100LX includes an on-chip Universal Serial Bus (USB) interface that complies with the Universal Serial Bus Revision 1.1 specification.

The interface is equipped with two USB ports designated p1 and p2 respectively, the inputs and outputs of which are multiplexed on to the same pins as other interface applications (see chapter 19 *Electrical Information*). The characteristics and operational principles of both ports are similar. The electrical interface is compatible with a Philips model PDIUSBP11A transceiver (or equivalent), as illustrated below.

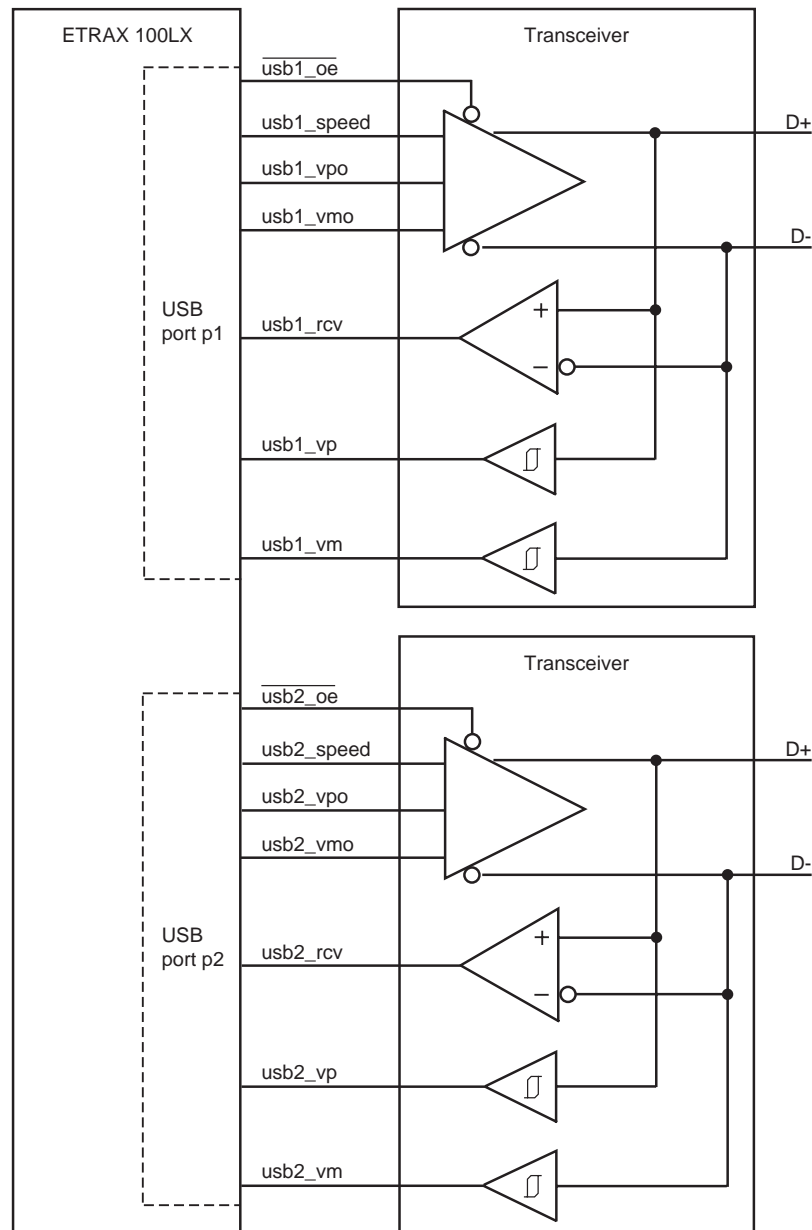


Figure 8-1 USB Transceiver Interface

8.1 Principle of Operation

8.1.1 Basic Architecture of the USB Interface

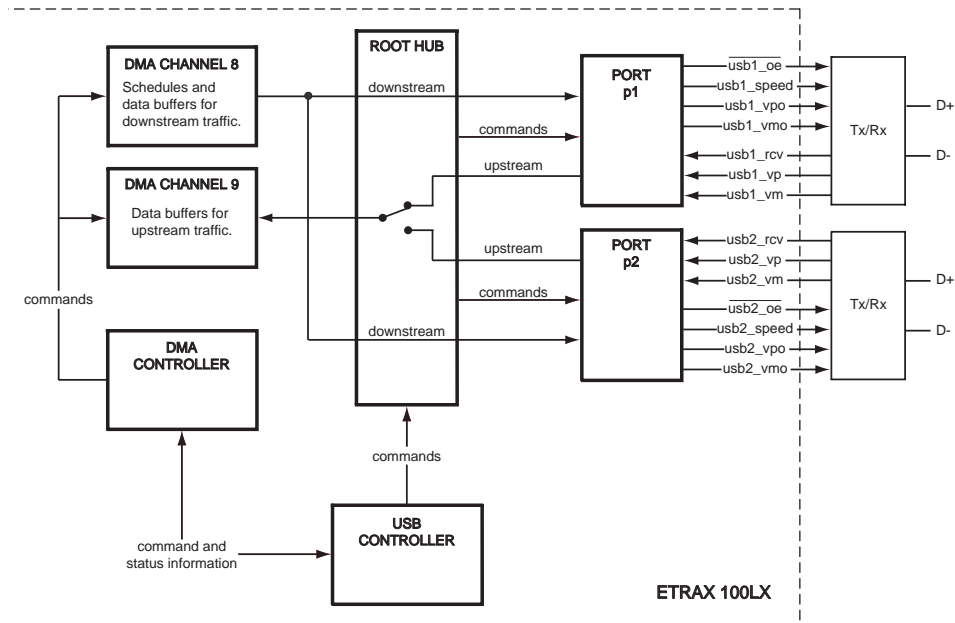


Figure 8-2 Basic USB Architecture

The USB is comprised of a root hub with ports p1 and p2 and the USB controller. The ports are duplex, and each port is connected to a dedicated transceiver (via I/O multiplexing not shown on the diagram). The root hub in the data stream controls the two USB ports and is itself managed by the USB controller, which handles frame control and timing, transaction protocol, port management and error recovery. In device mode, the root hub is only used as a port controller.

The USB interface operates in cooperation with the Direct Memory Access (DMA) controller, which manages all traffic schedules and communicates with the USB via two DMA channels. DMA channel 8 handles the traffic schedules and data buffers for downstream (OUT) traffic; DMA channel 9 handles the data buffers for upstream (IN) traffic. Please refer to chapter 7 *DMA* for detailed information on DMA.

8.1.2 Modes of Operation of the USB Interface

The USB interface can be configured to operate in one of two modes:

Host mode - in which ETRAX 100LX acts as the host for devices connected to the USB. In this mode, the USB interface can use either or both ports.

Device mode - in which ETRAX 100LX acts as a device communicating with a host elsewhere. Either of the two USB ports can be used in this mode, but not both simultaneously.

8.2 Operational States of the USB Controller

The USB controller has six operational states which are illustrated below.

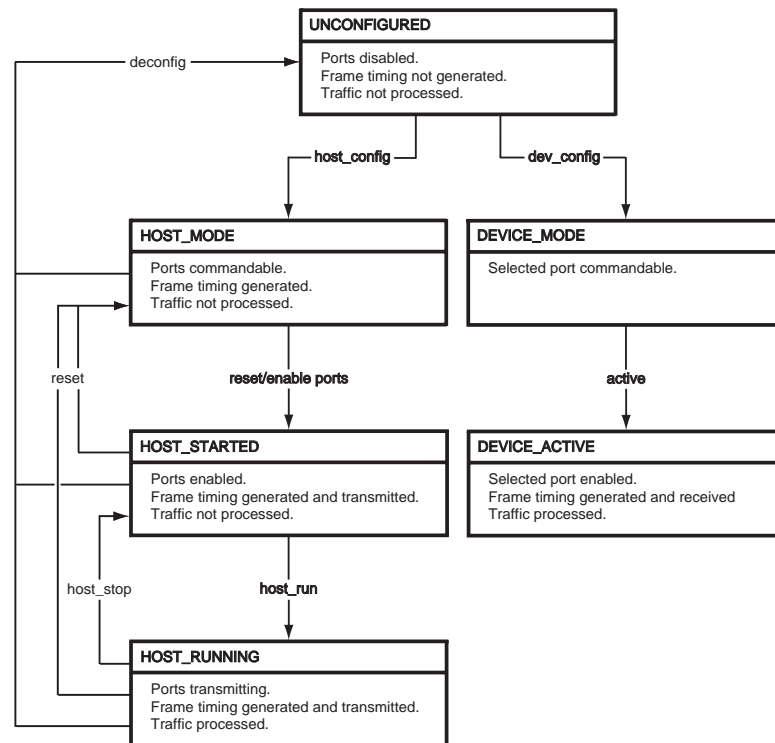


Figure 8-3 Operational States of the USB Controller

The UNCONFIGURED state prevails immediately after the execution of a **deconfig** command or system reset. The USB controller is stopped, the ports are disabled, frame timing is not generated and no traffic is processed.

HOST_MODE prevails in response to a **host_config** command while in the UNCONFIGURED state, or a **reset** command while in the HOST_STARTED or HOST_RUNNING state. In HOST_MODE the USB controller is prepared to act as a host. Commands can be issued to the ports and frame timing is generated, but no traffic is processed.

The HOST_STARTED state prevails when one or both ports have been enabled in HOST_MODE, or in response to a **host_stop** command in the HOST_RUNNING state. In the HOST_STARTED state, frame timing is generated and transmitted but traffic is not processed.

The HOST_RUNNING state prevails in response to a **host_run** command in the HOST_STARTED state. This is the fully operational state of the USB controller in Host mode, with the configured port(s) handling processed traffic.

DEVICE_MODE prevails in response to a **dev_config** command while in the UNCONFIGURED state. The USB controller is prepared to act as a device and commands can be issued to the selected port.

The DEVICE_ACTIVE state prevails in response to an **active** command in the DEVICE_MODE. This is the fully operational state of the USB controller in Device mode, with frame timing generated and the configured port handling traffic.

Please refer to section 8.4.1 *USB Controller Commands in Host Mode* for information on the USB controller commands.

8.3 USB Registers

8.3.1 Register Access Timing

There is a short delay between writing to a mode register and its effect on the hardware. Consequently, a **NOP** (no operation) instruction must be inserted between a write operation and a succeeding read operation to the same or affected register in order to read the updated value. This needs to be considered, for example, when accessing the EP Table.

8.3.2 USB Mode Registers

The USB interface is served by a set of dedicated mode registers. Table 8-1 below introduces these registers and summarizes their functions. For more detailed information on the USB registers, please refer to chapter 18.16 *Universal Serial Bus Interface Control Registers*.

Register	Function
R_USB_COMMAND	A byte-wide, read/write register that controls the USB commands in Host mode. Its functions include port selection, host and device configuration/deconfiguration, reset, run and stop commands.
R_USB_COMMAND_DEV	A byte-wide, read/write register that controls the USB commands in Device mode. The commands are similar to those used in Host mode.
R_USB_STATUS	A byte-wide, read-only register containing controller status information. The fields show whether the controller is busy, a Host or Device mode indicator, and whether Host mode is started and running.
R_USB_IRQ_MASK_SET	A 16-bit wide, write-only register in which ten control bits are used to enable or mask separate USB interrupts in Host mode. They are: isochronous end of frame (iso_eof); interrupt end of frame (intr_eof); isochronous end of transfer (iso_eot); interrupt end of transfer (intr_eot); control end of transfer (ctl_eot); bulk end of transfer (bulk_eot); endpoint attention (epid_attn); start of frame (sof); port status (port_status); controller status (ctl_status).
R_USB_IRQ_MASK_SET_DEV	A 16-bit wide, write-only register in which nine control bits are used to enable or mask separate USB interrupts in Device mode. They are: end of transfer/transaction for OUT (out_eot); endpoint3 end of transfer (ep3_in_eot); endpoint2 end of transfer (ep2_in_eot); endpoint1 end of transfer (ep1_in_eot); endpoint0 end of transfer (ep0_in_eot); endpoint attention (epid_attn); start of frame (sof); port status (port_status); controller status (ctl_status).

Table 8-1 *USB Mode registers*

R_USB_IRQ_MASK_READ	A 16-bit wide, read-only register that shows the status of the USB interrupts after individual bit masking in Host mode. Its contents are controlled by R_USB_IRQ_MASK_SET and R_USB_IRQ_MASK_CLR.
R_USB_IRQ_MASK_READ_DEV	A 16-bit wide, read-only register that shows the status of the USB interrupts after individual bit masking in Device mode. Its contents are controlled by R_USB_IRQ_MASK_SET_DEV and R_USB_IRQ_MASK_CLR_DEV.
R_USB_IRQ_MASK_CLR	A 16-bit wide, write-only register with ten control bits that are used to clear the USB interrupt mask bits in Host mode.
R_USB_IRQ_MASK_CLR_DEV	A 16-bit wide, write-only register with nine control bits that are used to clear the USB interrupt mask bits in Device mode.
R_USB_IRQ_READ	A 16-bit wide, read-only register containing ten pending USB interrupt bits. It shows the status of the USB interrupts prior to individual bit masking in Host mode.
R_USB_IRQ_READ_DEV	A 16-bit wide, read-only register containing nine pending USB interrupt bits. It shows the status of the USB interrupts prior to individual bit masking in Device mode.
R_USB_FM_NUMBER	A 32-bit wide, read/write register that reads the number of the current USB frame in Host mode. The register is cleared when the USB controller is reset. Reading this register clears the sof interrupt condition.
R_USB_FM_NUMBER_DEV	A 32-bit wide read/write register that contains the current frame number and Host/Device frame synchronization information in USB Device mode. The lower 11 bits represent the number of the current frame. The 8 msb bits read the time difference between the Host start-of-frame and the Device frame timer. The register is cleared when the USB controller is reset. Reading this register clears the sof interrupt condition.
R_USB_FM_INTERVAL	A 16-bit wide, read/write register containing a 14 bit value that defines the bit time interval in a frame (the distance between two start-of-frames). The frame timer decrements from this value to zero. The value is reloaded into register R_USB_FM_REMAINING at each start-of-frame (SOF). The value in this register is the frame length minus 1.
R_USB_FM_REMAINING	A 16-bit wide, read-only register that holds the remaining number of bit times in the current frame. The lower 14 bits represent this value: the two msb are not used.
R_USB_FM_PSTART	A 16-bit wide, read/write register that holds the periodic start point. The lower 14 bits represent this value: the two msb are not used. The value of the periodic start point is compared with a value counted downwards.
R_USB_RH_STATUS	A byte-wide, read-only register that contains root hub status information about the USB ports. Two 2-bit fields hold the bus states of ports p1 and p2 as sampled at the EOF2 time in the frame. A third 2-bit field represents the general condition of the USB interface.
R_USB_RH_PORT_STATUS_1	A 16-bit wide, read-only register containing status information about USB port p1. It is compatible with the wPortStatus field of the get_status command to USB hubs. The parameters that are read are speed , reset , suspended , enabled and connected . Two fields are not implemented in hardware and must therefore be handled in software. They are over_current (bit 3) and port_power (bit 8). When this register and R_USB_RH_PORT_STATUS_2 are read, the root hub status change interrupt condition is cleared. The register must be read as an entire word (16-bits).
R_USB_RH_PORT_STATUS_2	A 16-bit wide, read-only register containing the same information as R_USB_RH_PORT_STATUS_1, but for USB port p2.

Table 8-1 USB Mode registers

R_USB_EPT_INDEX	A byte wide, read/write register in which the five lsb contain the index of the endpoint lookup table to be used when reading and writing via register R_USB_EPT_DATA. The endpoint lookup table contains 32 endpoint entries, each pointing at an endpoint on a USB device. The table is indexed by the endpoint ID (ep_id) number in the USB DMA descriptors.
R_USB_EPT_DATA	A 32-bit wide, read/write register. It is the general endpoint table data register for normal (non-isochronous) transfers in Host mode. Bit 18 of the register is unused (reserved). The other fields contain: valid - validity of the table entry; hold - software exclusion bit; error_count_in - error counter for incoming transactions; t_in - toggle bit for incoming transactions; low_speed - endpoint low speed marker; port - indicates the upstream device traffic port; error_code - error type indicator; t_out - toggle bit for outgoing transactions; error_count_out - error counter for outgoing transactions; max_len - maximum length of non-isochronous data packets; ep - endpoint number; dev - configured device address.
R_USB_EPT_DATA_ISO	A 32-bit wide, read/write register. It is the general endpoint table data register for isochronous transfers in Host mode. The register is similar to R_USB_EPT_DATA.
R_USB_EPT_DATA_DEV	A 32-bit wide, read/write register. It is the general endpoint table for Device mode transfers. The register is similar to R_USB_EPT_DATA.
R_USB_EPID_ATTN	A 32-bit wide, read-only register. It contains a value representing an EP table entry that merits software attention. Reading this register clears the epid_attn interrupt condition.
R_USB_PORT1_DISABLE	A byte-wide register in which one bit is used to disable USB port p1.
R_USB_PORT2_DISABLE	A byte-wide register in which one bit is used to disable USB port p2.

Table 8-1 USB Mode registers

8.4 USB Host mode

In Host mode the USB interface manages frame timing, transaction protocol, port management and error recovery. The four transfer types stipulated in the USB specification are supported, namely: control (CTRL), bulk (BULK), interrupt (INTR) and isochronous (ISO).

Either or both of the USB ports can be used. Selection of the port(s) to be used is made by asserting fields **usb1** (bit 29) and **usb2** (bit 30) in general configuration register R_GEN_CONFIG. The act of port selection also enables the USB interface. Prior to port selection, the interface and its registers are entirely inoperative.

The root hub controls the reset, disable, suspend, resume, receive and transmit activities of the USB ports, even if only one port is configured for use. It is commanded by the **port_sel** (bits 7 and 6) and **port_cmd** (bits 5 and 4) fields in register R_USB_COMMAND. The state of the ports is read in registers R_USB_RH_PORT_STATUS_1 and R_USB_RH_PORT_STATUS_2. The status of the root hub is read in register R_USB_RH_STATUS.

8.4.1 USB Controller Commands in Host Mode

Register `R_USB_COMMAND` is used to control the USB interface in Host mode. This register contains fields for commanding the root hub and the USB controller. All these fields must be written in one operation.

Each time a write operation to register `R_USB_COMMAND` is performed, a command interpretation is triggered. This sets the **busy** field (bit 7) in register `R_USB_STATUS`. When the USB controller has executed the command, the **busy** field is cleared. The current state of the USB interface is read in register `R_USB_STATUS`.

The commands to the USB controller in Host mode are:

nop - (no operation)

One or both ports can be commanded without issuing any commands to the USB controller. The **nop** command is issued by setting field `ctrl_cmd` (bits 2 to 0) in register `R_USB_COMMAND` to the value 0x0.

reset

When the status of the USB controller is `HOST_STARTED` or `HOST_RUNNING`, this command resets the USB controller. It overrides any port commands by disabling either or both configured ports. The state of the USB controller changes to `HOST_MODE`, which is functionally equivalent to executing the **deconfig** and **host_config** commands in succession. The **reset** command is issued by setting the `ctrl_cmd` field in `R_USB_COMMAND` to the value 0x1.

deconfig

This is an emergency stop command that immediately deconfigures the entire USB interface, returning it to the condition that immediately succeeds a reset. The state of the USB controller changes to `UNCONFIGURED`. The **deconfig** command is issued by setting the `ctrl_cmd` field in register `R_USB_COMMAND` to the value 0x2.

host_config

With the USB controller in the `UNCONFIGURED` state, this command configures it as a host controller. Any command to a port is overridden by a configuration request to the port to adopt Host mode. The state of the USB controller changes to `HOST_MODE` and, as soon as a port is reset and enabled, the status changes to `HOST_STARTED`. The **host_config** command is issued by setting the `ctrl_cmd` field in register `R_USB_COMMAND` to the value 0x3.

dev_config

Please refer to section 8.5 *USB Data Structures in Host Mode, Device mode*. The **dev_config** command is issued by setting the `ctrl_cmd` field in register `R_USB_COMMAND` to the value 0x4.

host_nop

The **host_nop (no operation)** command is issued by setting the `ctrl_cmd` field in `R_USB_COMMAND` to the value 0x5.

host_run

When the host is started, this command starts the processing of USB traffic, and the status of the controller changes to HOST_RUNNING. It is not necessary to set up the various data structures for USB traffic before issuing the **host_run** command, but no traffic processing will occur until this has been done. The **host_run** command is issued by setting the **ctrl_cmd** field in register R_USB_COMMAND to the value 0x6.

host_stop

If the controller status is HOST_RUNNING, this command stops all traffic processing and changes the status of the controller to HOST_STARTED. The **host_stop** command is issued by setting the **ctrl_cmd** field in register R_USB_COMMAND to the value 0x7.

Most commands to the USB controller are effective in certain states only. The table below summarizes the relationship between the commands and the state of the USB controller.

Status	ctrl_cmd							
	nop	reset	deconfig	host_config	dev_config	host_no_p	host_ru_n	host_stop
UNCONFIGURED	nop	nop	OVR	OVR	OVR	nop	nop	nop
HOST_MODE	nop	nop	OVR	nop	nop	nop	nop	nop
HOST_STARTED	nop	OVR	OVR	nop	nop	nop	OK	nop
HOST_RUNNING	nop	OVR	OVR	nop	nop	nop	nop	OK

In the table above:

OVR indicates that the **ctrl_cmd** field in register R_USB_COMMAND overrides the **port_sel** and **port_cmd** fields.

OK and **nop** indicate that the **port_sel** and **port_cmd** commands are executed.

Note 1: If the USB controller is in the UNCONFIGURED state, then the **port_cmd** command cannot be executed.

If a command is written to register R_USB_COMMAND while the **busy** bit (3) in register R_USB_COMMAND is set, the result is unpredictable.

8.4.2 USB Port (Root Hub) Commands in Host Mode

The root hub operates in conjunction with the USB controller. Commands to the root hub are given in the **port_cmd** field of register R_USB_COMMAND. Any combination of the **ctrl_cmd** and **port_cmd** values can be used, but some are inappropriate. For instance do not command the USB controller to assume the HOST_RUNNING state whilst simultaneously disabling both of the USB ports. Instead, set the **ctrl_cmd** field of register R_USB_COMMAND to 0 (**nop**), and then use the **port_sel** and **port_cmd** fields to command the root hub.

The commands to the USB ports in Host mode are issued to the port(s) selected in the **port_sel** field (bits 7 to 6) in register R_USB_COMMAND as follows:

reset

The selected port is reset and enabled. A full port initialization of the port is performed and a bus reset is signalled according to the USB protocol. This occupies approximately 3 ms, depending upon the point in the frame cycle at which the command is issued. The **reset** command is issued by setting the **port_cmd** field in register R_USB_COMMAND to the value 0x0.

disable

The selected port is disabled. The **disable** command is issued by setting the **port_cmd** field in register R_USB_COMMAND to the value 0x1.

suspend

The selected port is placed in a suspended state. The port will not forward outbound traffic but will detect remote wakeup signalling or disconnects on the bus. The **suspend** command is issued by setting the **port_cmd** field in register R_USB_COMMAND to the value 0x2.

resume

The selected port is forced to resume operation. It starts a resume signalling sequence on the USB, after which the forwarding of USB traffic commences. The **resume** command is issued by setting the **port_cmd** field in register R_USB_COMMAND to the value 0x3.

The root hub handles most of the signalling on the bus. Functions such as connect/disconnect detection, reset signalling, suspend/resume and speed detection are automatic and accord with the USB specification. Some device attachment and configuration timeouts must be handled in software.

8.5 USB Data Structures in Host Mode

8.5.1 Transfer Frames

A USB transfer frame has the basic structure shown below:

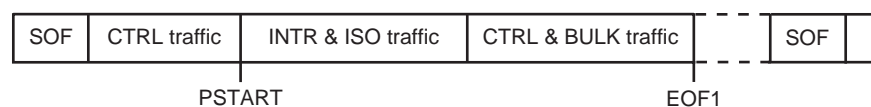


Figure 8-4 Basic Structure of USB Transfer Frame in Host Mode

The significant points in the transfer frame are the start-of-frame (SOF), periodic start (PSTART) and end-of-frame (EOF) events.

Start of Frame

The SOF point signifies the start of a transfer frame. The period from the beginning of one SOF to the next is the bit time interval of the frame. This is loaded into the **value** field (bits 13:0) of register R_USB_FM_REMAINING at the start of each frame. The frame timer decrements from the bit time interval to zero, at which point the next SOF occurs.

The bit time interval is a 14-bit value contained in the **fixed** and **adj** (adjustable) fields of register R_USB_FM_INTERVAL. The default value is 0x2EDF. The **fixed** field (bits 13:6) contains the upper 8-bits of the frame interval and, as the field name suggests, they are read-only.

The **adj** field (bits 5:0) contains the lower 6-bits of the frame interval. These bits can be used to adjust the bit time interval if necessary. Revision 1.1 of the USB specification requires that the interval must not deviate from the nominal (12000 bit times), by more than 15 bit times. Moreover the software must not adjust the frame interval by more than one bit time over six frames, and only one bit time is permitted in each adjustment.

Note 2: The value in register R_USB_FM_INTERVAL is actually the frame length minus 1.

Periodic Start Point

PSTART is where the transfer of interrupt and isochronous traffic can commence. Prior to the PSTART mark, only control transfers are performed. The position of PSTART is set by the 14-bits of the **value** field (13:0) in register R_USB_FM_PSTART.

To determine when to start sending periodic traffic, the USB controller compares R_USB_FM_PSTART with R_USB_FM_REMAINING, which is counted downwards. The USB driver should set R_USB_FM_PSTART to 0x2A30 (Periodic traffic starts 10% after **sof**).

More control traffic is permitted when all interrupt and isochronous transactions in the current frame are finished. When all remaining control traffic is finished, then bulk traffic transactions are performed.

End of Frame

At the EOF point, all traffic for the current frame is stopped and the entire procedure is repeated in the next frame. The register R_USB_FM_REMAINING shows the number of bit times remaining in the current frame. Register R_USB_FM_NUMBER gives the 32-bit frame number of the current frame.

Note 3: A 32-bit frame number is used in Host mode only. The lower 11 bits of register R_USB_FM_NUMBER are sent at the SOF.

Transfers and Toggle Bits

The USB interface does not set or clear the toggle bits at the end of a transfer. In fact the toggle bits are initialized only during the setup transaction in a CTRL transfer.

This feature can be used to concatenate multiple short transfers (as presented to the USB interface), into one long transfer (as seen by the device). It also means that the software may have to preset the toggle bits between transfers. This affects the insertion of transfers in the SB descriptor lists.

Maximum Transfer Length

Revision 1.1 of the USB specification does not specify a maximum transfer length. The USB interface is designed to construct longer transfers by concatenation.

Please refer to Universal Serial Bus Revision 1.1 specification for more detailed information about toggle bits, transfers and transfer length.

8.5.2 DMA Descriptors

The DMA generates the traffic for the USB interface, using linked lists of descriptors to track the USB traffic data and schedules. For further information regarding DMA refer to chapter 7 *DMA*.

DMA channel 8 is used for the traffic schedule and outgoing traffic data. DMA channel 9 is used for incoming traffic data. All incoming traffic terminates in the same buffer list in DMA channel 9.

Each traffic type has its own data structure, and the DMA divides channel 8 into four sub-channels (8.0 to 8.3), one for each traffic type, each of which is a linked list of linked lists. Sub-channel 8.0 is used for bulk traffic, sub-channel 8.1 for control traffic, sub-channel 8.2 for interrupts and sub-channel 8.3 for isochronous traffic.

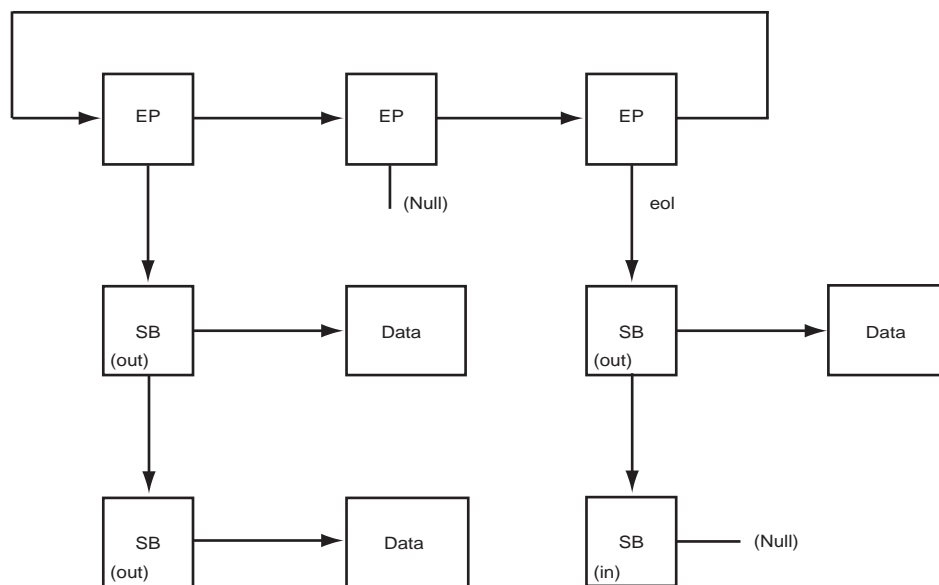


Figure 8-5 DMA List Structure for USB

In the upper dimension, the endpoint (EP) descriptors are aligned. From each EP descriptor, a number of sublist (SB) descriptors descend to form a transfer list for that endpoint. The EP list has exactly one descriptor with the *eol* (end-of-list) flag set.

An EP descriptor can be enabled only by software, but the hardware and the software can both disable an endpoint. When the hardware disables an EP descriptor, the software is notified by means of the *epid_attn* interrupt (See section 8.5.4 *Host Mode Interrupts*).

The SB descriptors describe the transfer to be performed. There are four types of transfer:

ZOUT - a special transfer used by the control, interrupt and isochronous traffic types, but not for bulk traffic.

SETUP - this transfer type is used for control transfers only.

IN and OUT - these transfer types are used in all four types of traffic. The SB descriptor for the OUT transfer has an associated buffer for the outgoing data, whereas the IN transfer data is received by DMA channel 9.

DMA Descriptors for IN Transfers

The format of a DMA descriptor for an IN transfer, used by DMA channel 9, is shown below. It is recommended that all DMA descriptors for USB are 32-bit aligned due to performance.

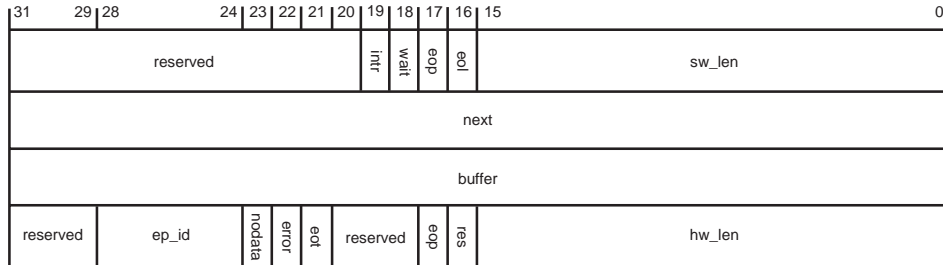


Figure 8-6 Format of DMA Descriptor

For a detailed description of the DMA descriptor refer to chapter 7.4.3 *DMA Descriptor Format for USB*.

Endpoint Descriptors

The format of an EP descriptor, used by DMA channel 8, is shown below. The **ep_id** field reports the EP identifier with which this buffer is associated, and is used to demultiplex the input stream into the corresponding endpoint pipe streams. All EP descriptors must be 32-bit aligned.

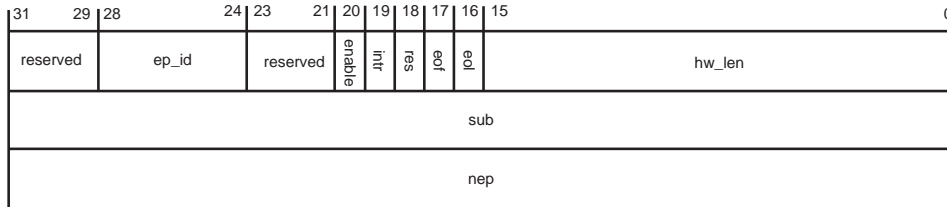


Figure 8-7 Format of an EP Descriptor

For a detailed description of the EP descriptor refer to chapter 7.4.3 *DMA Descriptor Format for USB*.

Sublist Descriptors

The format of an SB descriptor, used by DMA channel 8, is shown below.

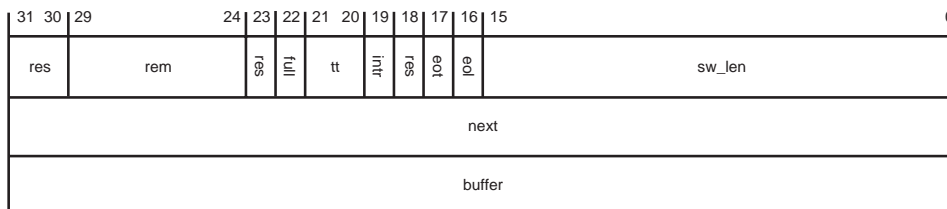


Figure 8-8 Format of SB Descriptor

In the special case where the transfer length is evenly divisible by the packet length, the **full** field is used to prevent the USB controller from sending an empty packet at the end of the transfer.

The **rem** field is used by IN transfers to count the bytes in the last packet. If the last packet is expected to be full, then the **rem** field is set to zero. For a detailed description of the SB descriptor refer to chapter 7.4.3 *DMA Descriptor Format for USB*.

SB Descriptors for Bulk Traffic

The transfers in the sublists can be IN or OUT. Each transfer can be constructed from one or more SB descriptors, but the last descriptor in a transfer must have the **eot** flag set. All descriptors in a single transfer must be of the same transfer type with the last descriptor in a sublist having the **eot** flag set.

In the examples that follow, the maximum packet size is set in the EP table (See section 8.5.3 *Endpoint Table in Host Mode*).

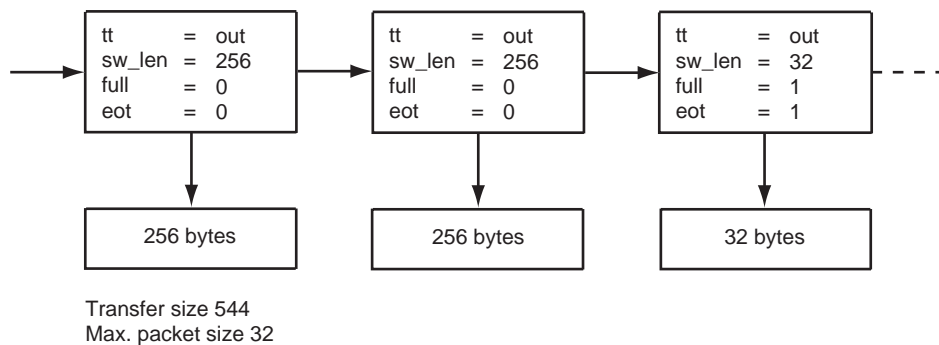


Figure 8-9 Example 1 of an OUT Transfer

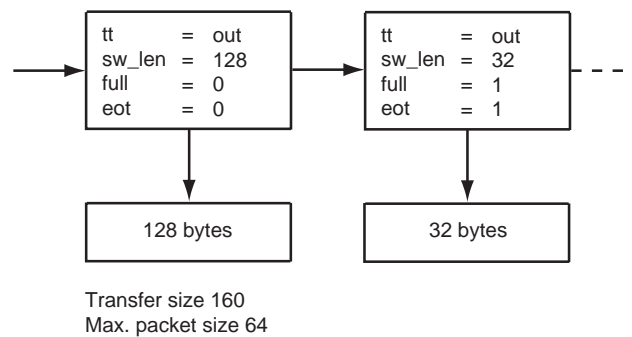


Figure 8-10 Example 2 of an OUT transfer

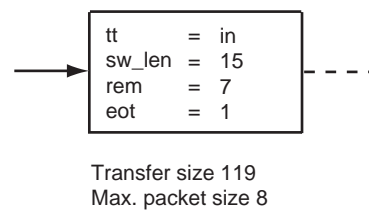


Figure 8-11 Example of an IN transfer

The calculations are:

```
sw_len = size ? (size - 1) / max packet size + 1 : 0;
rem = size % max packet size;
```

Note 4: For IN transfers, the `sw_len` field counts in packets.

SB Descriptors for Control (CTRL) Traffic

The format of a control transfer is more complicated than that of other traffic types. The USB specification requires that a control transfer must have either two or three phases: *setup*, *data* (optional), and *status*- and each phase uses one SB descriptor. For example, to build a control write it is necessary to have three SB descriptors: one SETUP, one OUT and one IN.

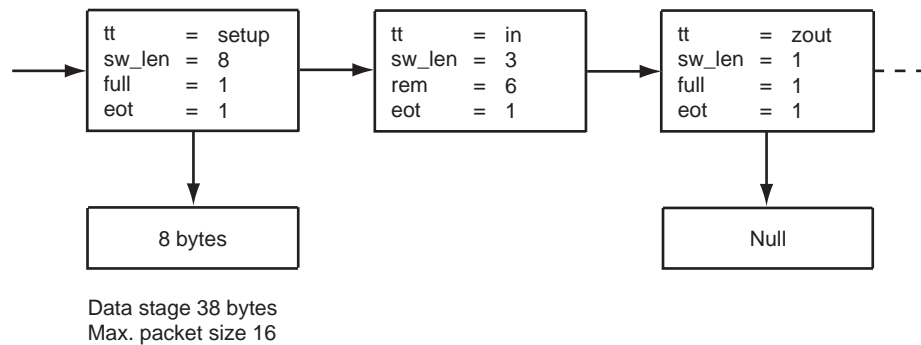


Figure 8-12 Example of a Control Read Transfer

Note 5: According to the Universal Serial Bus Revision 1.1 specification, the setup transaction must be exactly 8 bytes long.

The ZOUT descriptor buffer pointer should be set to NULL.

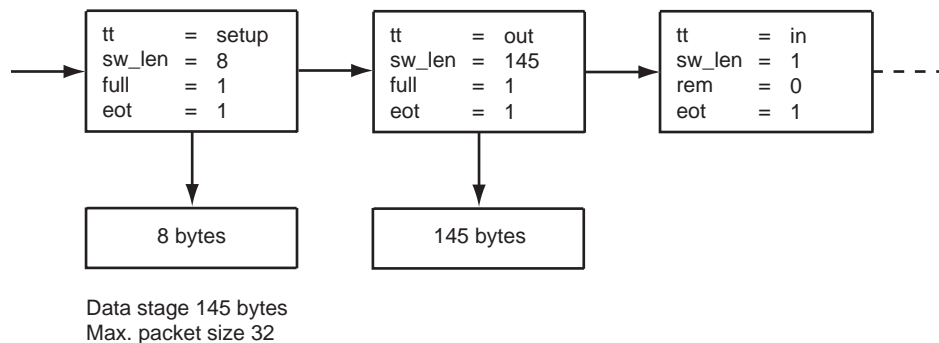


Figure 8-13 Example of a Control Write Transfer

Note 6: The `sw_len` field in the last descriptor has to be exactly 1. This is due to a requirement of the USB

specification that a host must be prepared to receive data (from a bogus device) in the status phase.

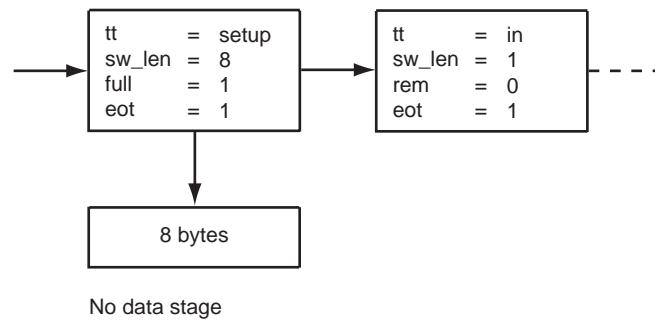


Figure 8-14 Example of a No-data Control Transfer

EP and SB Descriptors for Interrupt (INTR) Traffic

The EP descriptors for INTR traffic represent IN or OUT interrupt pipes. The technique is to run one transfer per endpoint and then change. If the endpoint has its **eof** flag set, then this signifies a *dummy* endpoint. Dummy endpoints are used to control the frequency of interrupt transfers: they must have only one SB descriptor attached, and that must be a ZOUT transfer.

ZOUT transfers are special. The ZOUT SB descriptor for periodic traffic (INTR and ISO) is not consumed and the DMA is instructed to move to the next endpoint. If the **eof** flag of the EP descriptor is not set, then an empty packet is sent. If the EP descriptor has its **eof** flag set, then the ZOUT transfer does nothing except force the hardware to stop processing interrupt traffic for that frame.

OUT transfers of INTR traffic resemble those of BULK traffic. One transfer, or an attempt to transfer, is performed in every frame (or desired period). OUT transfers of INTR traffic with different interrupt periods are complex to support.

In the first example below, the host will deliver 40 bytes in the first frame, after which it will send an empty packet in each frame. Note that if a device replies with NAK, or makes no reply, the host will attempt to re-send the packet in the next period (the next frame in this example).

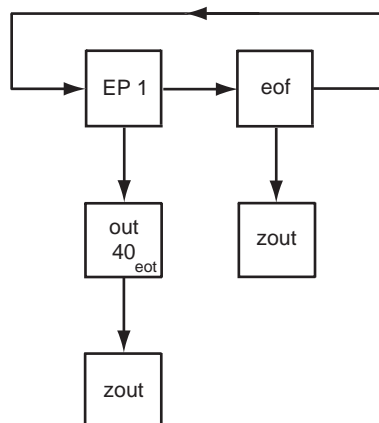


Figure 8-15 Example of Endpoint 1 (OUT) in Each Frame of INTR Traffic

In the next example below, the host will deliver 30 bytes in the first frame, then wait five frames and deliver 20 bytes. After this an empty packet will be transmitted every

fifth frame. Note that if a device replies with NAK, or makes no reply, the host will attempt to re-send the packet in the next period (after five frames in this example).

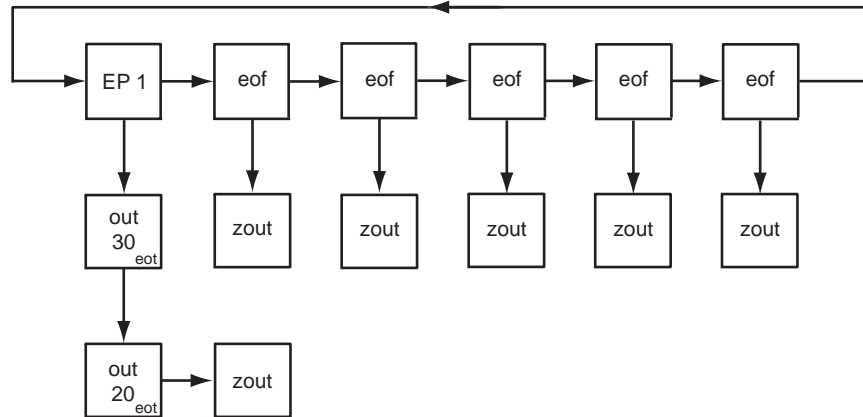


Figure 8-16 Example of Endpoint 1 (OUT) Every Fifth Frame of INTR Traffic

IN transfers of periodic traffic (INTR and ISO) are different from those of bulk and CTRL traffic. When the IN transfer is finished the SB descriptor is not consumed, which makes the IN transfer remain in the list. The first example below shows IN endpoint EP1 in each frame and IN endpoint EP2 in every second frame.

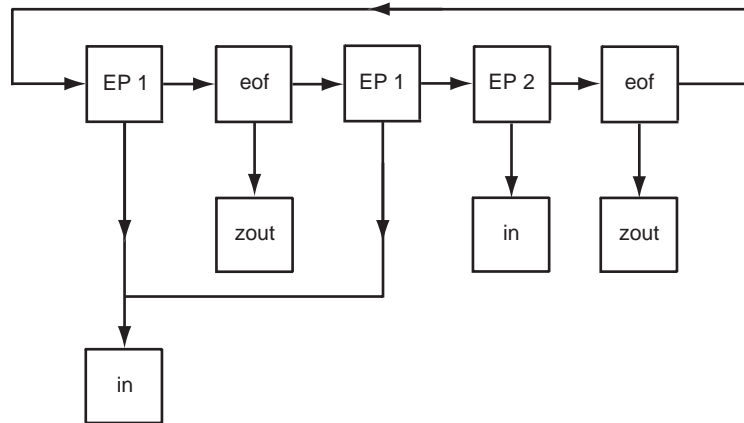


Figure 8-17 Example of Endpoint 1 (IN) Every Frame and Endpoint 2 (IN) Every 2nd Frame of INTR Traffic

The final example shows IN endpoint EP1 in every fifth frame.

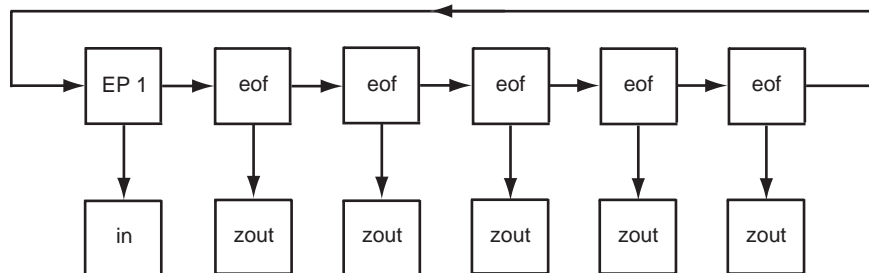


Figure 8-18 Example of Endpoint 1 (IN) Every Fifth Frame of INTR Traffic

EP and SB Descriptors for Isochronous (ISO) Traffic

Isochronous transfers are uni-directional, and the EP descriptors represent isochronous pipes. An EP descriptor with the `eof` flag set is a dummy endpoint used to mark the end of isochronous traffic. This `eof` endpoint must contain one SB descriptor with a ZOUT transfer.

For OUT transfers, only one transaction per endpoint is transmitted in each frame. If the SB list ends, then the endpoint is disabled. No re-transmissions are used. In the event of an underrun error, the USB controller forces the DMA to skip the transaction in which the underrun occurs. To make this behavior useful, it is recommended that there should be only one transaction per transfer (e.g. the data size should not be longer than the max packet size).

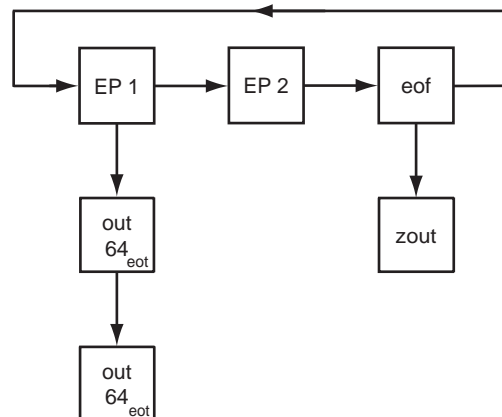


Figure 8-19 Example of ISO OUT traffic

For IN transfers, one request per endpoint is transmitted in each frame. As with interrupt traffic, the IN transfers never end. The `iso_eot` interrupt is given for all ISO IN requests.

8.5.3 Endpoint Table in Host Mode

The USB interface can handle 31 active endpoints in devices on the bus. The interface therefore includes a lookup (EP) table for translating endpoint identifiers into device/endpoint pairs. The EP table also contains status information for each valid `ep_id`. Most of this information is used by the hardware only, but some fields are used to exchange information between the hardware and the software.

The EP table contains 32 endpoint entries, each pointing at an endpoint on a device on the USB. The EP table for isochronous traffic differs slightly from that of all other traffic types: it requires fewer fields.

The EP table is accessed via three mode registers:

`R_USB_EPT_INDEX` is used to point to the desired entry in the EP table.

`R_USB_EPT_DATA` is used to read and write to the indexed entry in the EP table for all traffic types except isochronous.

`R_USB_EPT_DATA_ISO` is used to read and write to the indexed entry in the EP table for isochronous traffic.

After reset, all entries in the EP table are invalidated, which means that the **valid** field (bit 31), of each entry in register R_USB_EPT_DATA is cleared.

If no periodic transfer is ever scheduled, then all 32 entries in the EP table can be used. However, if interrupt or isochronous traffic are to be used, then one entry must be invalid, leaving 31 entries available.

Note 7: In software, the best way to create an invalid entry is to allocate one entry in the table; do not adjust it thereafter.

The software must never modify an entry in the EP table as long as the corresponding EP descriptor is enabled. If an entry is modified while the EP descriptor is enabled, the behavior is undefined and unpredictable. To avoid such errors a synchronization method must be used. Please refer to section 8.7 *Physical Interface* for details.

For a full definition of the fields in the EP table for all traffic types except isochronous, please refer to chapter 18.16.7 *R_USB_IRQ_MASK_CLR*.

For a full definition of the fields in the EP table for isochronous traffic, please refer to chapter 18.16.18 *R_USB_RH_STATUS*.

8.5.4 Host Mode Interrupts

A number of interrupts are generated by the USB interface: most of them indicate events in specific devices on the USB. All of the USB interrupts have the same internally-generated vector number (0x3F).

As noted in table 8-1 above, the USB interrupts are handled in four registers:

```
R_USB_IRQ_MASK_SET;  
R_USB_IRQ_MASK_READ;  
R_USB_IRQ_MASK_CLR;  
R_USB_IRQ_READ.
```

It is also necessary for the USB interface to identify any endpoint that triggers an interrupt. Register R_EPID_ATTEN is used for this purpose. Each bit in the register corresponds to an entry in the endpoint lookup table.

In summary the USB interrupts are:

iso_eof

This interrupt is triggered in response to an isochronous traffic end-of-frame flag. In Host mode the **iso_eof** interrupt occurs when the DMA reports a set **eof** flag in an isochronous EP descriptor to the USB interface. The interrupt is cleared when the **value** field in register R_USB_EPID_ATTEN is read.

intr_eof

This interrupt is triggered in response to an interrupt traffic end-of-frame flag. In Host mode the **intr_eof** interrupt occurs when the DMA reports a set **eof** flag in an interrupt EP descriptor to the USB interface. The interrupt is cleared when the **value** field in register R_USB_EPID_ATTEN is read.

iso_eot

In Host mode, this interrupt is triggered when an isochronous transaction is completed. The interrupt is cleared when the **value** field in register R_USB_EPID_ATTEN is read.

intr_eot

In Host mode, this interrupt is triggered in response to an end-of-transfer flag when an interrupt transfer is completed. The interrupt is cleared when the **value** field in register R_USB_EPID_ATTEN is read.

ctl_eot

In Host mode, this interrupt is triggered in response to an end-of-transfer flag when a control transfer is completed. The interrupt is cleared when the **value** field in register R_USB_EPID_ATTEN is read.

bulk_eot

In Host mode, this interrupt is triggered in response to an end-of-transfer flag when a bulk transfer is completed. The interrupt is cleared when the **value** field in register R_USB_EPID_ATTEN is read.

epid_attn

In Host mode, this interrupt is triggered whenever a significant event occurs at an endpoint. The interrupt condition is cleared when the R_USB_EPID_ATTEN register is read. The events that trigger an **epid_attn** interrupt are:

invalid ep_id - occurs when the USB interface receives a transfer with an invalid endpoint identifier in the EP descriptor. It is probably caused by a programming error and is a good reason to stop the affected endpoint at least. The EP descriptor will be disabled by the hardware.

stall - not strictly an error condition but the EP descriptor is disabled nevertheless. For CTRL transfers there are special precautions to be taken by the software. The hardware does not perform any special routine for CTRL stalls. They are handled in the same way as BULK or INTR stalls.

3rd error - occurs in response to three successive transaction error in a transfer, which disables the EP descriptor. Another severe error in a transaction may also trigger this event. Inspect the EP table.

buffer overrun - a buffer overrun or underrun occurs if the memory or DMA cannot handle the traffic load. The EP descriptor is disabled.

past eof1 - this event is triggered if an INTR or ISO transaction proceeds beyond the EOF1 mark in the frame.

near eof - this relatively rare event is triggered if an attempt is made to start an INTR or ISO transaction that would not fit inside the frame. It is handled in the same way as the **past eof1** event.

zout transfer - this error event is triggered if a ZOUT transfer is ever presented to any BULK endpoint. The EP descriptor is disabled.

setup transfer - this error event is triggered if a SETUP transfer is ever presented to any INTR, ISO or BULK endpoint. The EP descriptor is disabled.

sof

In Host mode, this interrupt is triggered whenever a start-of-frame flag leaves the USB interface. The interrupt is cleared when the **value** field in register R_USB_FM_NUMBER is read.

port_status

In Host mode, this interrupt signals any change in the status registers of the configured USB ports. The interrupt is cleared when these registers (R_USB_RH_PORT_STATUS_1 and R_USB_RH_PORT_STATUS_2) are read.

ctl_status

In Host mode, this interrupt indicates a change in the status of the USB interface controller. The interrupt is cleared when register R_USB_STATUS is read.

8.6 Device Mode

In Device mode, only one port can be configured for use, either port p1 or port p2. Selection of the port to be used is made by asserting fields **usb1** (bit 29) or **usb2** (bit 30) in general configuration register R_GEN_CONFIG. The act of port selection enables the USB interface, which is entirely inoperative prior to this.

All operations at the enabled port are handled by the root hub under the control of the **port_sel** (bits 7 and 6) and **port_cmd** (bits 5 and 4) fields in register R_USB_COMMAND_DEV. The status of the enabled port is read in register R_USB_RH_PORT_STATUS_1 if port p1 is in use, or R_USB_RH_PORT_STATUS_2 for port p2.

Root hub status register R_USB_RH_STATUS is not used in Device mode.

8.6.1 USB Controller Commands in Device Mode

Register R_USB_COMMAND_DEV contains fields for commanding the root hub and the USB controller. All these fields must be written in one operation.

Each time a write operation is performed to register R_USB_COMMAND_DEV, a command interpretation is triggered. This sets the **busy** field (bit 7) in register R_USB_STATUS. When the USB controller has executed the command, the **busy** field is cleared. The current state of the USB interface is read in main status register R_USB_STATUS.

The only commands that can be issued to the USB controller in Device mode are:

nop - (no operation)

The enabled port can be commanded without issuing any commands to the USB controller. The **nop** command is issued by setting field **ctrl_cmd** (bits 2 to 0) in register R_USB_COMMAND_DEV to the value 0x0.

deconfig

This is an emergency stop command that immediately deconfigures the entire USB interface, returning it to the condition that immediately succeeds a reset. The state of the USB controller changes to UNCONFIGURED. The **deconfig** command is issued by setting the **ctrl_cmd** field in R_USB_COMMAND_DEV to the value 0x2.

host_config

If the USB controller is in the UNCONFIGURED state, this command configures it as a host controller. The **host_config** command is issued by setting the **ctrl_cmd** field in register R_USB_COMMAND_DEV to the value 0x3.

dev_config

If the USB controller is in the UNCONFIGURED state, this command configures it as a device controller. The **dev_config** command is issued by setting the **ctrl_cmd** field in register R_USB_COMMAND_DEV to the value 0x4.

8.6.2 USB Port (Root Hub) Commands in Device Mode

In Device mode, commands to the root hub are given in the **port_cmd** field of register R_USB_COMMAND_DEV. They are issued to the port selected in the **port_sel** field (bits 7 to 6) of the command register as follows:

active

Enable the selected port. This must be done after the **dev_config** command has been issued to the USB controller. The **active** command is issued by setting the **port_cmd** field in register R_USB_COMMAND_DEV to the value 0x0.

passive

Disable the selected port. Traffic on the bus is ignored and none is sent from the device. The **passive** command is issued by setting the **port_cmd** field in register R_USB_COMMAND_DEV to the value 0x1.

wakeup

Drive a K-state of 3 ms in length on the bus to request remote wakeup. This command is valid only if the device is in a suspended mode. The **wakeup** command is issued by setting the **port_cmd** field in R_USB_COMMAND_DEV to the value 0x3.

8.6.3 USB Data Structures in Device Mode

In Device mode, the USB controller can handle four IN endpoints and 12 OUT endpoints. Note that the terms IN and OUT refer to the direction at the USB host; in other words, an IN Transfer is outgoing from the device, and an OUT transfer is incoming to the device. DMA channel 8 is used for outgoing traffic from the USB device to the host and is divided into four sub-channels, one for each IN endpoint.

The data structures for the sub-channels of DMA channel 8 are organized as four linked lists of linked lists, where the first list (the EP list), always points to itself. DMA sub-channel 8.0 is used as a control endpoint, while sub-channels 8.1 to 8.3 can be used for any traffic type.

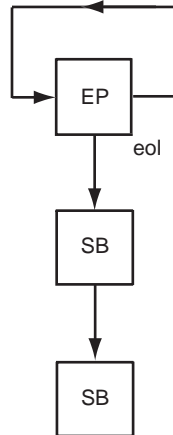


Figure 8-20 DMA List Structure for USB Device (IN Endpoint)

ETRAX 100LX supports 12 endpoints for incoming traffic from the USB host (OUT traffic). DMA channel 9 handles this traffic, all of which is placed in the FIFO buffer of DMA channel 9. This channel behaves in the same way as any other DMA channel in ETRAX 100LX, but with additional status flags.

DMA Descriptors for OUT Transfers in Device Mode

The format of a DMA descriptor for an OUT transfer (incoming traffic to the device) is shown below. This format is used by DMA channel 9, and it is recommended that all DMA descriptors for USB are 32-bit aligned.

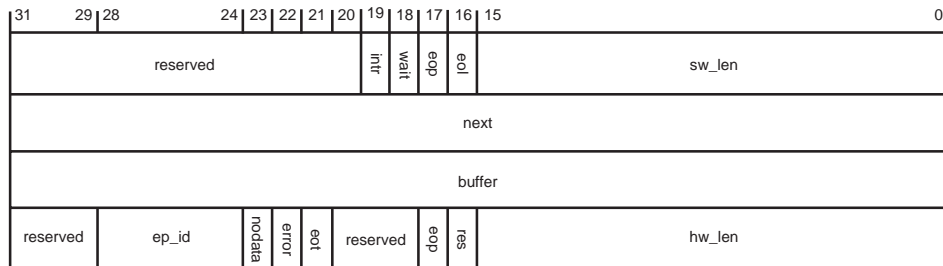


Figure 8-21 Format of DMA Descriptor

For a detailed description of the DMA descriptor refer to chapter 7.4.3 *DMA Descriptor Format for USB*.

EP Descriptors in Device Mode

Endpoint descriptors are mainly used to store the **ep_id** number but, in USB Device mode, the **ep_id** number is fixed as follows:

Sub-Channel	ep_id
8.0	0
8.1	1
8.2	2
8.3	3

The format of an EP descriptor, used by DMA channel 8, is shown below. There is only one EP descriptor per sub-channel. All EP descriptors must be 32-bit aligned.

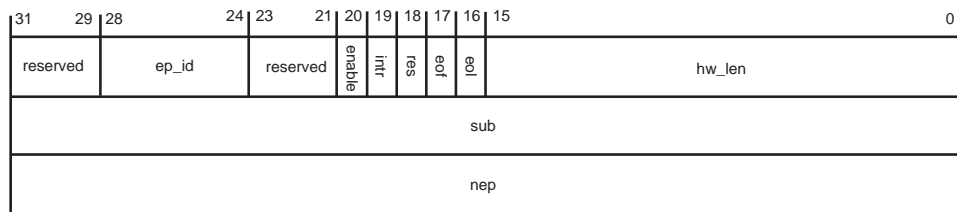


Figure 8-22 Format of an EP Descriptor

Note that the **eof** bit is not used in Device mode, and should be set to 0. For a detailed description of the EP descriptor refer to chapter 7.4.3 *DMA Descriptor Format for USB*.

SB Descriptors in Device Mode

In Device mode, the SB descriptors for the sub-channels in DMA channel 8 contain status and pointer fields to the data that will be transmitted from the device to the USB host. The format of an SP descriptor in a sub-channel is illustrated below.

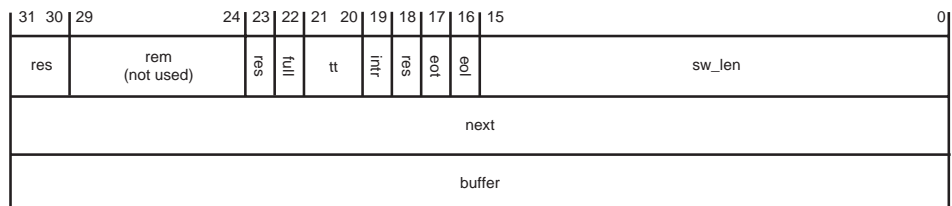


Figure 8-23 Format of SB Descriptor in Device Mode

Note that the **rem** field is not used in Device mode, and should be set to 0. In Device mode, the only transfer types allowed in the **tt** field are ZOUT and OUT. For a detailed description of the SB descriptor refer to chapter 7.4.3 *DMA Descriptor Format for USB*.

8.6.4 EP Table in Device Mode

The EP table is used in Device mode as well as Host mode, and is accessed through mode registers:

- **R_USB_EPT_INDEX** which is used to point to the desired entry in the EP table.
- **R_USB_EPT_DATA_DEV** which is used to read and write to the indexed entry in the EP table.

The EP table also contains status information for each valid **ep_id**. 8-2 below shows how the rows in the EP table are used:

To fill rows 0 - 11 and 16 - 19, the register macros for **R_USB_EPT_DATA_DEV** could be used.

The data toggle bit for an endpoint is only initialized during a setup transaction.

ep_id	Description
0 - 11	Configuration for USB out traffic - incoming traffic to the device. All data received will end up in DMA channel 9, and there are 12 endpoints available. ep_ids 20 - 31 are used to know the transfer size of USB out traffic: ep_id 20 is used by ep_id 0 ep_id 21 is used by ep_id 1 ep_id 22 is used by ep_id 2 ep_id 23 is used by ep_id 3 ep_id 24 is used by ep_id 4 ep_id 25 is used by ep_id 5 ep_id 26 is used by ep_id 6 ep_id 27 is used by ep_id 7 ep_id 28 is used by ep_id 8 ep_id 29 is used by ep_id 9 ep_id 30 is used by ep_id 10 ep_id 31 is used by ep_id 11
12 - 15	Reserved
16 - 19	Configuration for USB in traffic - outgoing traffic from the device. There are four endpoints are available, one for each DMA channel 8 subchannel.
20 - 31	Counters for ep_ids 0 - 11. Bit [31:16] is the transfer size Bit [15:0] is the number of bytes remaining in the ongoing transfer. Note that for a control endpoint, the counter is loaded by hardware with the transfer size received within the setup packet.

Table 8-2 EP table usage

For a full definition of the fields of the EP table in Device mode, please refer to chapter 18.16.19 *R_USB_RH_PORT_STATUS_1*.

8.6.5 Device Mode Interrupts

The USB interrupts in Device mode are:

out_eot

This interrupt is triggered by an end-of-transfer flag on any of the twelve OUT endpoints.

ep3_in_eot, ep2_in_eot, ep1_in_eot

These interrupts are triggered by an end-of-transfer flag on IN endpoints numbered 1 to 3 respectively.

ep0_in_eot

This interrupt is triggered by an end-of-transfer flag on IN endpoint number 0 (control).

epid_attn

In Device mode, this interrupt is triggered in response to an endpoint overrun or underrun condition. The interrupting endpoint is identified in register *R_USB_EPID_ATTEN* and the error condition can be read in register *R_USB_STATUS*.

sof

In Device mode, this interrupt is triggered each time the frame timer reaches 0. The frame timer interval is set in *R_USB_FM_INTERVAL*, and can be synchronized with the frame interval of the host by using the information in *R_USB_FM_NUMBER_DEV*.

port_status

In Device mode, this interrupt signals any change in the status of the USB port.

ctl_status

In Device mode, this interrupt indicates a change in configuration.

8.7 Physical Interface

The physical interface of the USB is divided into two main parts: data transmission and power management. These differ slightly in Host mode and Device mode.

8.7.1 Data Transmission

The data transmission interface is compatible with Philips USB transceiver type PDIUSBP11A. It is almost identical for Host and Device mode, with minor differences in the protocol handling. There is also a difference on the outside of the transceiver, concerning the pull-up and pull-down for device speed indication.

In host mode, D+ and D- are individually pulled down with 15K Ohm.

In Device mode, a 1.5K Ohm resistor is used for device speed selection. If a full-speed device is built, the resistor should be a pull-up on the D+ line. For a low-speed device, the resistor should be a pull-up on the D- line.

8.7.2 Power Management

USB power management is not implemented in the ETRAX 100LX USB interface. Since power management is slow and rare, it can be implemented in software using general port PA for overcurrent sensing, and any generic I/O pin for power on/off control. The fact that general port PA can be configured to generate interrupts is a useful feature in this regard.

8.7.3 Hardware Reset

Whenever the USB interface is not configured in register R_GEN_CONFIG, then the USB interface reset is active and the interface is entirely frozen. Writing to the USB registers has no effect, and reading from the registers may produce unpredictable results. Immediately after reset the USB interface is in a passive state. The USB ports are off, in the sense that their output enable signals are inactive.

Note 8: One way to perform a hardware reset of the USB interface is to deconfigure the interface in register R_GEN_CONFIG and then reconfigure the interface.

It is the responsibility of the software to manage USB power during startup of the USB interface. This is done outside the USB interface as noted in 8.7.2. above.

8.8 Procedures

8.8.1 Configuring the USB Interface for Host Mode

Setting the General Configuration

Set the **usb1** (bit 29) and/or **usb2** (bit 30) fields in register **R_GEN_CONFIG** to configure either or both of the two available USB ports. This initializes the USB interface and causes a hardware reset of the USB interface to be performed.

Configuring the Interface

Set the **ctrl_cmd** field (bits 2 to 0) in register **R_USB_COMMAND** to the value **host_config** (0x3). This issues the **host_config** command to the USB interface.

8.8.2 Starting and Stopping the Host Mode

When configured, the USB interface eventually reaches the **HOST_MODE** state, where it remains until at least one port is enabled. When a port is enabled, the **HOST_STARTED** state is attained and traffic processing can be started by issuing the **host_run** command to the USB controller. This sets the USB interface to the **HOST_RUNNING** state.

Starting USB Traffic Processing

Set the **ctrl_cmd** field in register **R_USB_COMMAND** to the value **host_run** (0x6). This issues the **host_run** command to the USB controller.

Stopping USB Traffic Processing

Set the **ctrl_cmd** field in register **R_USB_COMMAND** to the value **host_stop** (0x7). This issues the **host_stop** command to the USB controller.

All traffic processing is suspended but frame generation continues. The traffic for the current frame is finished before the state changes to **HOST_STARTED**. The traffic can be started again without any special actions as long as the data structures are intact. However it should be noted that the devices may have become unstable during the stoppage time.

8.8.3 Starting and Stopping Traffic in Host Mode

This is related to the start/stop of traffic processing. In principle, traffic can be forcibly stopped by commanding the USB controller from the **HOST_RUNNING** state to the **HOST_STARTED** state. However a clean shutdown may leave devices in a safer condition than a forced shutdown.

Stopping Each Traffic Type with a Clean Procedure

- 1 Withdraw any pending transfers. This may require the temporary disabling of the endpoint in order to inspect and manipulate the SB descriptor lists.
- 2 Wait for the DMA sublist to stop. When the endpoints exhaust one by one, they are automatically disabled by the USB interface. When all endpoints are disabled, the DMA sub-channel will stop.

- 3 When all sub lists have stopped, issue the **host_stop** command to the USB controller by setting the **ctrl_cmd** field in register R_USB_COMMAND to the value **host_stop** (0x7).

Restarting Traffic After a Stop

- 1 Set the **ctrl_cmd** field in register R_USB_COMMAND to the value **host_run** (0x6). This issues the **host_run** command to the USB controller.
- 2 Set up the new transfers to be processed. All traffic was withdrawn before the traffic was stopped and therefore all SB lists are empty.

8.8.4 Managing EP Descriptor Lists in Host Mode

All EP descriptor lists must have a disabled dummy EP descriptor with the **eol** flag set. To create a new EP descriptor list it is necessary to create the first descriptor, link the first descriptor to the dummy EP descriptor, and set the **eol** flag. This must be done before the DMA sub-channel is started. The list will resemble the following diagram:

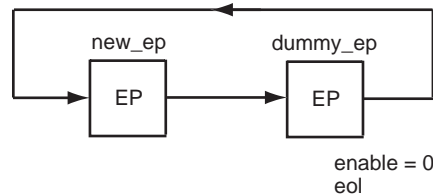


Figure 8-24 New EP Descriptor List in Host Mode

Inserting an EP Descriptor into an Existing List

- 1 Create a new EP descriptor, here named **new_ep**. It is recommended that all values in the new descriptor should be set to zero. Locate the point in the list where the **new_ep** is to be inserted. The descriptor that precedes the insertion point is here named **before_ep** and the descriptor that succeeds the insertion point is named **after_ep**.

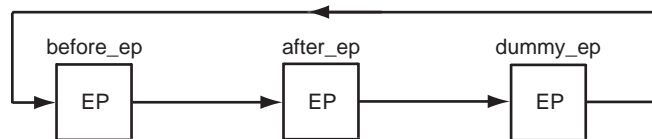


Figure 8-25 Existing Descriptor List

- 2 Point the new endpoint at the **after_ep** descriptor. This must be done before the next step.

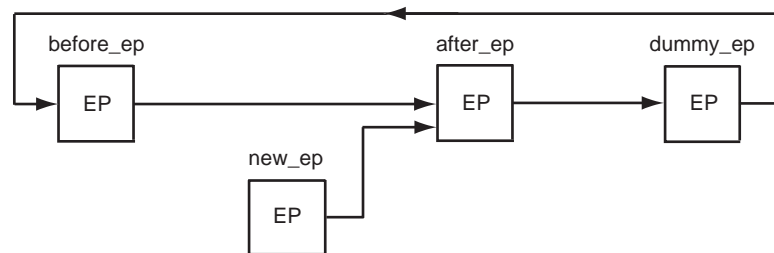


Figure 8-26 Pointing the New Endpoint at the Succeeding Descriptor

- 3 Point the **before_ep** at the **new_ep** descriptor. To ensure that the hardware does not detect a partially-updated new endpoint, this must be done in one single 32-bit write operation.

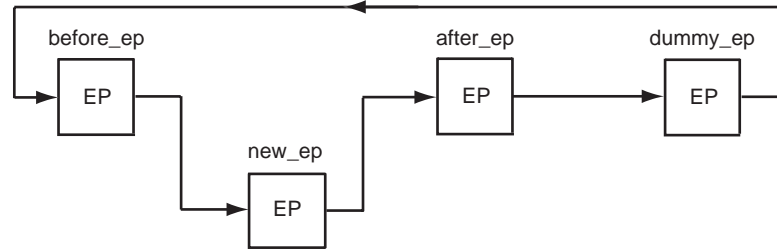


Figure 8-27 Pointing the Preceding Descriptor at the New Endpoint

Removing an EP Descriptor from the List

- 1 Bypass the descriptor by changing **before_ep.next** to **old_ep.next**. This must be done in one single 32-bit write operation.
- 2 Check that the value in `R_DMA_CH8_SUBx_EP` is not equal to **old_ep**.
- 3 If the values are not the same it is safe to remove the **old_ep**.
- 4 If these two values are equal, set the interrupt bit **intr** in the *dummy* descriptor and wait for the `dma8_subx_desc` interrupt.
- 5 At the `dma8_subx_desc` interrupt, remove **old_ep**, and clear the interrupt bit in the dummy descriptor.
- 6 Acknowledge the interrupt by writing to `R_DMA_CH8_SUBx_CLR_INTR`.

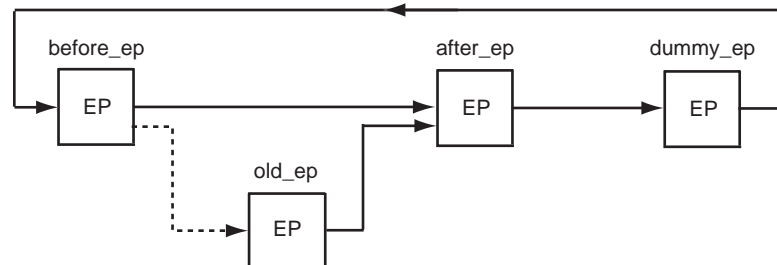


Figure 8-28 Removing the EP Descriptor

8.8.5 Managing SB Descriptor Lists in Host Mode

It is possible to insert descriptors in the middle of an SB descriptor list, but advanced disable/enable EP descriptor manipulation with list traversal is necessary to find the intersections between transfers. New traffic must not be inserted in the middle of a transfer because it would confuse the USB hardware and produce unpredictable results.

The preferred method of inserting descriptors is to append new traffic at the end of the list, as described below. It is recommended an entire transfer should be inserted in one operation because this simplifies other aspects of the operation.

- 1 Find the last SB descriptor in the list.
- 2 Append the new descriptor to the list.
- 3 Clear the **eof** flag in the last descriptor.

- 4 Check whether the EP descriptor at the head of the list is enabled or disabled. If it is enabled, then the procedure is concluded. Otherwise go to step 5.
- 5 Check whether an error disabled the EP descriptor, taking care not to lose any information from the interrupt system. If there was an error condition, proceed with step 6. Otherwise go to step 7.
- 6 Correct the error condition, which may necessitate a permanent stop of the endpoint. This must be handled in software, which must correctly withdraw all transfers in the SB descriptor list, including the newly-inserted transfer. When this has been done, restart the endpoint if possible.
- 7 Check if the **sub** field in the EP descriptor is pointing to the new SB descriptor. If so, the procedure is completed; otherwise, go to step 8.
- 8 Update the EP descriptor. There was no error and the EP descriptor was disabled, therefore the SB descriptor list was exhausted. The sub field in the EP descriptor must be pointed at the new SB descriptor.
- 9 Enable the EP descriptor to conclude the procedure.

When removing descriptors from the SB descriptor list it is very important not to confuse DMA, which could lead to unpredictable events in the USB interface. For more information, please refer to the above section 8.8.4 *Managing EP Descriptor Lists in Host Mode*.

Removing SB Descriptors from the List

- 1 Disable the endpoint, see 8.8.4 *Managing EP Descriptor Lists in Host Mode*.
- 2 Find the start and stop of the transfer to retire. If the endpoint was disabled due to an error, the error must also be corrected.
- 3 Remove the transfer to retire.
- 4 If the list is not empty, re-enable the endpoint.

8.8.6 Managing the EP Table in Host Mode

Connecting an EP Descriptor to an EP Table Entry

- 1 Allocate an EP table entry. The entry number is the **ep_id**.
- 2 Write the **ep_id** into the **value** field of register R_USB_EPT_INDEX and set the **ep_id** field in the EP descriptor.
- 3 Clear the EP table entry in register R_USB_EPT_DATA.
- 4 Set the device and endpoint parameters in register R_USB_EPT_DATA.
- 5 Set the **valid** field (bit 31) in the EP table entry.
- 6 Insert the EP descriptor into an EP list.
- 7 Enable the EP descriptor.

Disabling an Endpoint and Accessing the EP Table Entry

Follow the procedure set forth in section 8.8.4 *Managing EP Descriptor Lists in Host Mode*. When the EP is safely disabled, the entry in the EP table could also be modified.

Re-enabling the Endpoint

- 1 Set the enable bit in the EP descriptor.
- 2 Check that the DMA is still running. If it has stopped for some reason (e.g. no more data to process), then write a start command to the appropriate DMA command register.

8.8.7 Managing the DMA Channel 9 Descriptor List

This is largely the same as managing any DMA list for incoming traffic. A typical USB DMA descriptor list structure resembles Figure 8-29 below.

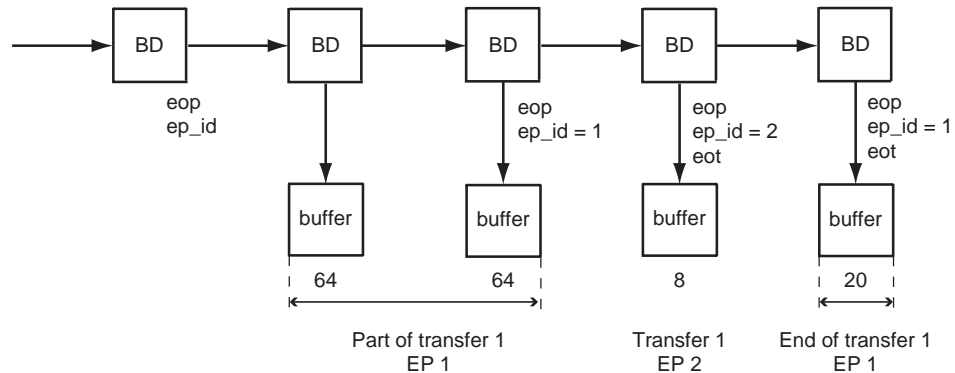


Figure 8-29 Typical USB DMA Descriptor List

After reception of a transfer (one packet), the **ep_id** and the **eop** field are written to the DMA descriptor in DMA channel 9. If the transfer is an empty packet, a dummy byte is written and the **nodata** bit is set.

Observe that a packet ends in the buffer that is flagged **eop**, and the next packet begins in the immediately succeeding buffer.

The software must manage the demultiplexing of packets to the correct pipe, and the cleanup after an unrecoverable transaction error.

8.8.8 Managing the Root Hub

As previously noted, the root hub manages the low level details of the USB connect/disconnect, speed detection, reset and suspend/resume signalling. The root hub also detects faulty signalling and babbling devices.

Section 8.8.3 *Starting and Stopping Traffic in Host Mode*, describes the command sequences to the USB controller. The command sequences to reset and enable a USB port are discussed here.

The root hub has dedicated registers to disable the USB ports. These registers are `R_USB_PORT1_DISABLE` and `R_USB_PORT2_DISABLE`. They must be configured before a USB port can be used.

Writing to the Disable Registers

Set the **disable** field (bit 0) in register `R_USB_PORT1_DISABLE` to the value **no** (0x1). This ensures that USB port p1 can be enabled. Set the **disable** field (bit 0) in register `R_USB_PORT2_DISABLE` to the value **no** (0x1). This ensures that USB port p2 can be enabled.

Detecting a Port Event

- 1 Wait for the USB interrupt. If register R_USB_IRQ_READ indicates a **port_status** interrupt, go to step 2. Otherwise wait until an interrupt occurs.
- 2 Handle the event. Ensure that the software can remember the values in these registers in order to detect the condition that has changed.

Some events also trigger a **ctl_status** interrupt. For instance a command changing a port from **reset** to **enabled** condition may set the USB controller from the **HOST_MODE** state to the **HOST_STARTED** state. Also note that the last port to change from **enabled** to **disconnected** or **reset** condition will return the controller to the **HOST_MODE** state.

If a port event indicates that a port became connected, it is appropriate to reset the port as follows:

- 1 Issue a reset command to the port by writing to the R_USB_COMMAND register.
- 2 Wait for the reset to complete. Wait for the port enabled event.
- 3 Notify the upper software that an enabled event occurred at the port.

The sequence above shows a command to one port (i.e. the root hub). This is the best way to command the root hub. It is possible to issue commands to the USB controller and the root hub, but there is a potential for error.

Many operations on the root hub follow the same pattern:

- 1 Perform an event.
- 2 Wait for its completion.
- 3 Report the change in status.

Disabling a Port

- 1 Set the **disable** fields (bit 0) in register R_USB_PORT1_DISABLE or R_USB_PORT2_DISABLE to the value **yes** (0x0). This forces the respective port to recognize a disconnect event and change the port status accordingly.
- 2 Wait for the port status to change to disconnected.

8.8.9 Managing USB IN Traffic in Device Mode

As previously noted, in Device mode the USB controller has 4 IN endpoints. The traffic for those endpoints are scheduled in the sub-channels of DMA channel 8. Because of the endpoint numbers (0, 1, 2 and 3), it is necessary to use endpoint 0 as the control channel. Any of the other endpoint (1, 2 and 3) can carry bulk, interrupt or isochronous traffic.

Creating an IN Endpoint

- 1 Select an endpoint (0, 1, 2 or 3). In this example, endpoint 1 is used.
- 2 Create an EP list for the corresponding DMA sub-channel. The list comprises one EP descriptor pointing at itself.

- 3 Create SB descriptors describing the data that will be sent. In this example 321 bytes will be sent, with a buffer size of 256 bytes.

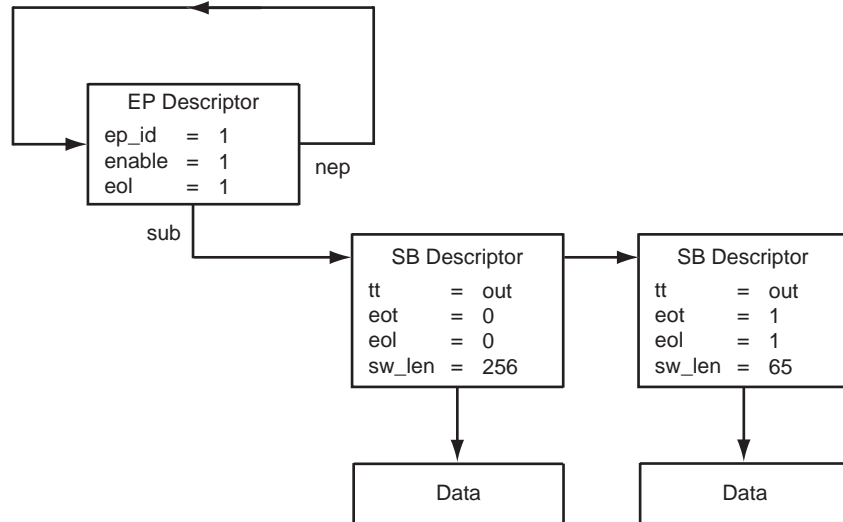


Figure 8-30 Creating an IN Endpoint

- 4 Fill information into the EP table. Write to the corresponding row in the EP table (row 17 since endpoint 1 has been chosen). This enables the endpoint and sets its parameters.
- 5 Start the DMA sub-channel by setting the **cmd** field (bit 0) in register `R_DMA_CH8_SUB1_CMD` to the value **start** (0x1).

If the USB host submits a request to this endpoint, the device will respond with data. When all data are sent, the sub-channel becomes disabled.

If the USB device controller receives more requests to this endpoint, it responds with NAK until the software places more data into the sub-channel and restarts it.

Appending Traffic to an IN Endpoint

- 1 Create an SB descriptor describing the data packet to be sent. In this example a USB transfer of 100 bytes in length will be appended. Set the **eot** and **eol** flags in this descriptor.

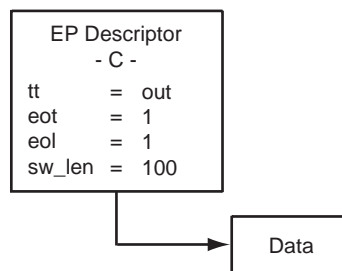


Figure 8-31 Creating an SB Descriptor for Appending Traffic to an IN Endpoint

- 2 Move the **next** pointer in the last SB descriptor (B) to point at (C).

- 3 Clear the **eol** flag in descriptor B. It is important that the **next** pointer is updated before the **eol** flag is cleared.

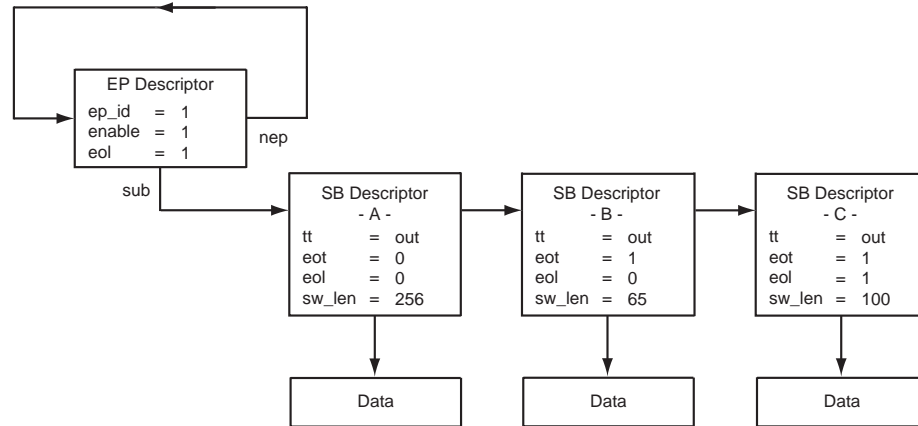


Figure 8-32 Clearing the **eol** Flag in Descriptor B

- 4 Check whether the sub-channel is stopped. The hardware stops a sub-channel only when the **eol** flag is reached. If the sub-channel is running, then the procedure is concluded. Otherwise go to step 5.
- 5 Check the sub-pointer in the EP descriptor. If this pointer is not directed at the new SB descriptor (C), then point it there and start the sub-channel. If the pointer in the EP is already pointing at the new SB, then that traffic has been sent and the sub-channel need not be started.

Stalling Traffic on an IN Endpoint

Set the stall bit in the corresponding row in the EP table. This forces the USB hardware to respond with a stall on all requests from the host to that endpoint. The software must disable stall mode to make this endpoint operate again. This should occur after a configuration event on the control channel (channel 0). It is not recommended that endpoint 0 should be stalled.

Removing Traffic from an Endpoint

- 1 Stop the DMA sub-channel by setting the **cmd** field (bit 0) in the appropriate register **R_DMA_CH8_SUBn_CMD** to the value **stop** (0x0).

The hardware will transmit the ongoing transaction. If the transmission of a packet has started, the hardware will attempt to finish the transmission. However the relatively abrupt stop of the DMA may cause an underrun that forces the hardware to abort the packet. This is signalled in the same way as an ordinary underrun condition.

- 2 Wait for the transaction to end. This cannot be observed by software, but the transaction cannot be longer than one USB frame. It is, therefore, possible to wait for one frame to have passed. This could be done by using the **sof** interrupt which is given once every frame.
- 3 Modify the list if necessary. Then start the sub-channel.

Error Conditions When Managing USB IN Traffic

All USB errors are handled by hardware, and the only error that can be reported is underrun. The underrun error is indicated in register R_USB_STATUS, and the number of the endpoint that caused the error can be read in register R_EPID_ATTEN.

If an isochronous endpoint encountered the underrun error, the hardware will abort the transaction and then disable the endpoint by clearing the valid bit at the corresponding row in the EP table. If a control, bulk or interrupt endpoint encountered the underrun error, the hardware only aborts the transaction, leading to a new request from the USB host.

8.8.10 USB OUT Traffic in Device Mode

As previously mentioned, all incoming traffic from the 12 OUT endpoints is handled in DMA channel 9. Endpoint 0 is always a control endpoint and the other endpoints can be bulk, interrupt or isochronous. All OUT endpoints require two entries in the EP table, one describing the endpoint and one containing the number of bytes in the transfer.

When constructing software for the device it is advisable to ensure that there are always free descriptors/buffers in DMA channel 9. This can be achieved by creating a list with interrupts enabled at certain points in the list - helping the software to free up descriptors. DMA channel 9 will stop if no descriptors remain, resulting in an overrun in the USB hardware.

USB Bulk and Interrupt OUT Traffic in Device Mode

As for all USB OUT transfers, the USB hardware must be informed of the expected transfer length. This is the maximum length for the transfer. The transfer could be shorter, but this must be signalled from the host by sending a short packet (not **max_packet_size**) or an empty packet.

If the transfer length is not known, the counter must be loaded with a large number (i.e 0xffff) and then reloaded with this value before it reaches zero. For bulk and interrupt traffic, the **max_packet_size** is 64 =>. The counter must be reloaded after ~1000 packets and, at the most, there can be 19 of these packets per frame (per 1 ms).

Another approach to an unknown transfer length is to load the counter with a multiple of the **max_packet_size** (i.e 0xffc0). The hardware will thus report *end-of-transfer* after 1023 packets but, if the actual end-of-transfer has not occurred, the host will continue to send data and the device hardware will receive the data.

After reception of a transaction (one packet), the **ep_id** and the **eop** field are written to the DMA descriptor in DMA channel 9. If the transaction is an empty packet, a dummy byte is written and the **nodata** bit is set.

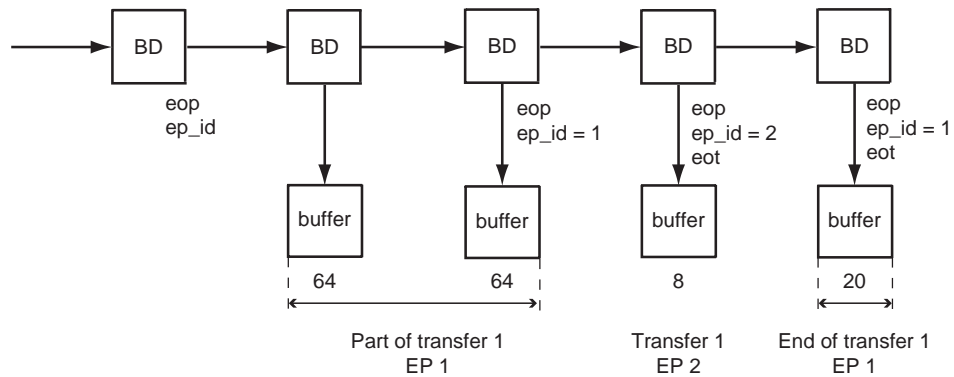


Figure 8-33 Typical USB DMA descriptor List

As mentioned before, the end-of-transfer can be signified by the receipt of:

- Exactly the number of previously agreed bytes
- A short packet
- An empty packet

At the end-of transfer, an interrupt is signalled (note that this interrupt is the same for all 12 OUT endpoints), and the `eot` bit is written in the DMA descriptor that contains the last data for that transfer.

USB Isochronous OUT Traffic in Device Mode

In Device mode, isochronous OUT traffic operates in almost the same way as bulk and interrupt traffic. The difference is that each transaction on an isochronous channel is a transfer. Consequently the transfer length required by hardware (from the EP table), must be set to the same value as `max_packet_size`.

Error Conditions for USB OUT Traffic in Device Mode

The only error not handled by the USB hardware is overrun. The overrun error is reported in register `R_USB_STATUS`, and the number of the endpoint that caused the error can be read in register `R_EPID_ATTEN`. If an endpoint overruns it does not acknowledge that transaction.

8.8.11 USB Control Traffic in Device Mode

Control traffic uses one OUT endpoint and one IN endpoint. With three exceptions, they act in the same way as other endpoints. These exceptions are:

- The transfer length of the OUT endpoint (traffic to device), is handled by hardware
- IN transfers are aborted if the host issues a new setup packet
- IN transfers are considered done when the status packet arrives

The transfer length of the OUT endpoint (traffic to device), is handled by hardware

For a setup packet, the hardware always loads the EP table with the value 8 and, after reception of the first DATA0 packet, the EP table is loaded with the expected transfer length. This is the two last bytes in that packet.

IN transfers are aborted if the host issues a new setup packet

If the host first issues a Control read command and then, before all data are read by the host, it issues another Control read command, then the hardware will skip the first transfer in the IN endpoint.

IN transfers are considered done when the status packet arrives

To support the requirements of section 8.5.2.3 of the revision 1.1 of the USB specification, the IN transfer is skipped if the status packet arrives before the IN transfer is complete. Consequently, there will be no eot interrupt on that endpoint.