Freescale Semiconductor
Application Note

Document Number: AN3520
Rev. 0, 09/2007

Simplified EHCI Data Structures for the High-End ColdFire Family USB Modules

by: Melissa Hunter
Microcontroller Division

1 Introduction

Some of the high-end ColdFire products (such as, the MCF532*x*, MCF5253, and MCF5445*x* devices) contain an EHCI-compatible host or dual-role/OTG USB controller. The dual-role module can be used as a USB host, device, or an On-the-Go device. In host mode, both USB modules are EHCI compliant. The EHCI specification defines a register set and data structures that control USB data movement.

The EHCI specification was designed for the PC world and therefore allows for an extremely robust host implementation supporting many different types of devices on a single port. For an embedded system a simpler implementation of USB might be desired. The purpose of this application note is to discuss a simplified version of the EHCI data structures, where a USB host driver supporting a few USB devices is desired instead of a full EHCI stack. It explains how the different data structures are used together, and provides basic examples of actual usage.

Contents

freescale™ semiconductor

This document is intended as a guide for developing a simple driver for communicating with a single device that could be one of a few types. For instance, if you want to have support for a mouse, then you build a mouse driver that only works with a mouse. If a different USB device is plugged in an error is returned. It is assumed that transfer sizes larger than 4 KB are not required. This application note assumes the reader is familiar with the basics of USB operation.

The data structures as discussed in this document do not support all of the capability of USB or EHCI. To simplify the EHCI data structures, isochronous transfers are not discussed. This document also assumes that only one device is connected to a port at a time. Since split transactions are only used to communicate with a full speed (FS) or low speed (LS) device that is connected through a USB 2.0 hub, they are not covered. Not supporting isochronous and split transactions eliminates some of the EHCI data structures entirely and many fields of the remaining data structures are not used.

NOTE

Portions of this document relating to the EHCI specification are Copyright © Intel Corporation 1999-2001. The EHCI specification is provided “As Is” with no warranties whatsoever, including any warranty of merchantability, non-infringement, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification or sample. Intel disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in the EHCI specification. Intel may make changes to the EHCI specifications at any time, without notice.

2 USB Host Overview

This section discusses the data structures that are used for an EHCI-compatible host. Several of the data structure types in the EHCI specification are used exclusively for handling isochronous transfers.

Once the data structures for isochronous transfers are eliminated, there are only four EHCI data structures—the periodic schedule, the asynchronous schedule, queue heads (QHs), and queue element transfer descriptors (qTDs).

The host controller within the USB dual-role controller uses two different systems for scheduling USB transfers:

- The periodic schedule manages interrupt and isochronous transfers
- The asynchronous schedule manages control and bulk transfers

Both the asynchronous and periodic schedules use QH and qTD data structures to configure the transfers. Typically there is a QH defined for each endpoint the host accesses. The QH determines the USB address and endpoint number for the transfer along with other information about the endpoint. The QH contains a pointer to the current qTD, a qTD overlay area where the contents of the current qTD are stored, and a pointer to the next qTD. The qTDs define the actual transfer (number of bytes, location to read/write data, and status).

2.1 Queue Head (QH)

The primary purpose of the QH is to define the characteristics of a particular endpoint that is being addressed. This means that in most cases there is one QH for each device endpoint being addressed. QHs must be 32-byte aligned.

Figure 1 shows a simplified QH structure where the fields relating to split transactions, isochronous transfers, and large transfers (transfers that require more than one buffer pointer) are removed or set to static values. This is a simplified form of the QH structure defined in the EHCI specification.

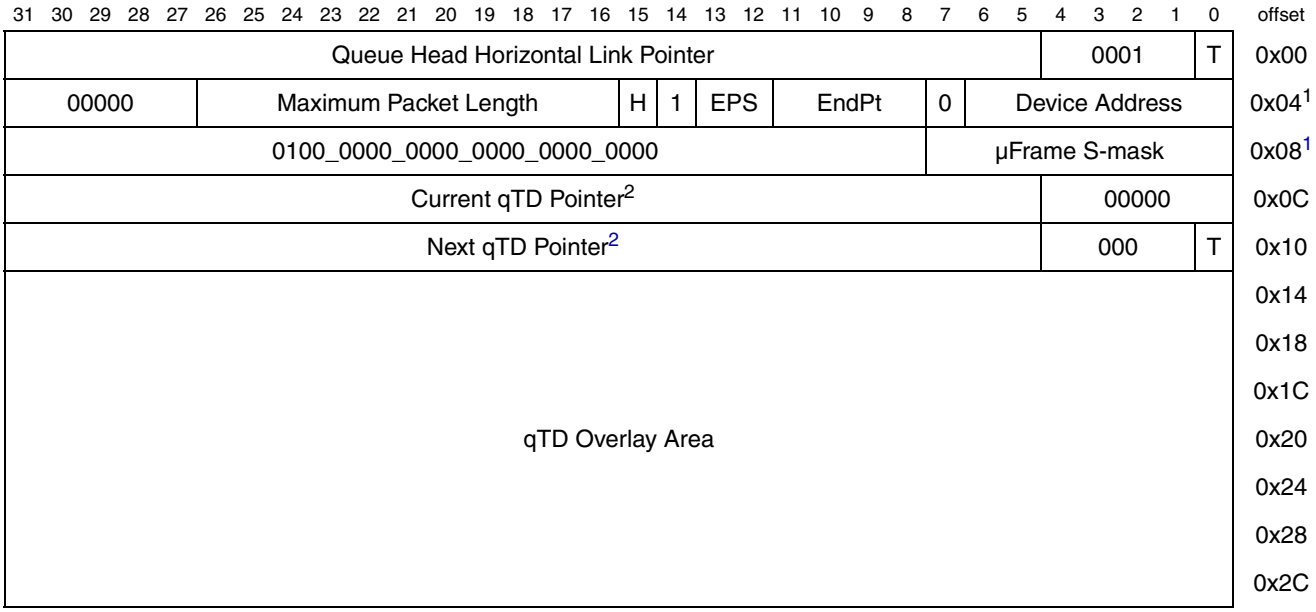

Figure 1. Simplified Queue Head (QH) Layout

1 Offsets 0x04–0x0B contain the static endpoint state.

2 Host controller read/write; all others read-only.

2.1.1 Queue Head Horizontal Link Pointer (Offset = 0x00)

The first longword of a QH contains a link pointer to the next data object to be processed after any required processing in this queue has been completed, as well as the control bits defined below. This pointer may reference a queue head or one of the isochronous transfer descriptors. In our case, it always is a pointer to the next QH.

Table 2-1. Queue Head Horizontal Link Pointer

Field	Description
31–5 QHLP	Queue head horizontal link pointer. This field contains the address of the next QH to be processed in the horizontal list and corresponds to memory address signals [31:5], respectively.
4–1	Write as 0001.
0 T	Terminate. 0 Pointer is valid 1 Last QH (pointer is invalid)

2.1.2 Endpoint Characteristics (Offset = 0x04)

The second longword of a QH specifies static information about the endpoint. This information does not change over the lifetime of the endpoint. These fields are written by the USB host software stack when the QH is setup and are never modified by the host controller hardware.

Table 2-2. Endpoint Characteristics

Field	Description
31–27	Write as 00000.
26–16 Maximum Packet Length	Set to the maximum packet size of the associated endpoint. The descriptors for the device specify the maximum packet length. The maximum value this field may contain is 0x400 (1024).
15 H	Head of reclamation list flag. This bit is set by system software to mark a queue head as the head of the asynchronous schedule.
14	Write as 1.
13–12 EPS	Endpoint speed. 00 Full speed (12Mbs) 01 Low speed (1.5Mbs) 10 High speed (480 Mb/s) 11 Reserved
11–8 EndPt	Endpoint number. This 4-bit field selects the particular endpoint number on the device serving as the data source or sink.
7	Write as 0.
6–0 Device Address	This field selects the specific device serving as the data source or sink.

2.1.3 Endpoint Capabilities (Offset = 0x08)

The third longword of a QH specifies a number of parameters that are associated with split transactions, so most of this longword is always set to the same value.

Table 2-3. Endpoint Capabilities

Field	Description
31–8	Write as 0x400_0000.
7–0 µFrame S-mask	Interrupt schedule mask. This field is only used for interrupt endpoints. For control and bulk endpoints this field must be written as 0x00. The mask corresponds to each microframe in a frame. If the current microframe number matches the S-mask value, then the QH is processed for that microframe. Since the minimum poll rate for a FS interrupt endpoint is 1 ms, this field should always be set to 1 for FS/LS interrupt endpoints. For high speed (HS) endpoints the S-mask value can be used for interrupts that have a poll rate less than 1 ms. For example, a value of 0xFF means the interrupt is processed on every microframe (every 125 µs). With a value of 0x11 the interrupt is processed every fourth microframe (every 500 µs). **Note:** An S-mask value of 0x00 for an interrupt endpoint causes undefined operation.

2.1.4 Current qTD Pointer (Offset = 0x0C)

This longword is the address of the qTD that is currently being processed. This field is written by the host controller when it reads in the qTD. Software does not need to initialize this longword when creating a new QH.

2.1.5 qTD Overlay Area (Offset = 0x10–0x2C)

The eight longwords in this area are a working copy of the qTD that is currently being processed or was last processed. While a transfer is in progress, the controller writes incremental status information to the qTD overlay area. When the transfer is complete, the results are written back to the original qTD location (the address pointed to by the current qTD pointer).

These values are initialized by the host controller when it copies in the current qTD. Therefore, software does not need to initialize these fields. The one exception is the next qTD pointer. This value should be initialized by software when creating a new QH. The next qTD pointer should be set to the address of the first qTD to be processed for the endpoint (with the T bit cleared to indicate a valid pointer). The controller uses the next qTD pointer to access the beginning of the linked list of qTDs for the endpoint.

2.2 Queue Element Transfer Descriptor (qTD)

A qTD defines an actual data movement for control, bulk, or interrupt transfers. The qTDs are processed as a singly-linked list. The next qTD pointer in the QH should be initialized to point to the first qTD in the linked list. After the first qTD is processed, the controller uses the next qTD pointer in the first qTD to find the second qTD. This process repeats until a qTD with an invalid next qTD pointer is reached. qTDs must be aligned on 32-byte boundaries.

Figure 2 shows a simplified version of the qTD defined in the EHCI spec that can transfer up to 4 KB of data.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	offset
Next qTD Pointer	0000 T	0x00
000_0000_0000_0000_0000_0000_0000_0000	1	0x04
dt[1] / Total Bytes to Transfer / ioc / 000[1] / Cerr[1] / PID Code / Status[1]		0x08
Buffer Pointer[1]		0x0C
0000_0000_0000_0000_0000_0000_0000_0000		0x10
0000_0000_0000_0000_0000_0000_0000_0000		0x14
0000_0000_0000_0000_0000_0000_0000_0000		0x18
0000_0000_0000_0000_0000_0000_0000_0000		0x1C

Figure 2. Simplified Queue Element Transfer Descriptor (qTD) Layout

[1] Host controller read/write; all others read-only.

2.2.1 Next qTD Pointer (Offset = 0x00)

The first longword of a qTD is a pointer to another qTD. This pointer is used to create a singly-linked list of qTDs.

Table 2-4. qTD Next Element Transfer Pointer (longword 0)

Field	Description
31–5 Next qTD Pointer	This field contains the physical memory address of the next qTD to be processed. The field corresponds to memory address signals[31:5], respectively.
4–1	Reserved. The value of these bits has no effect on operation.
0 T	Terminate. This bit indicates to the host controller that there are no more valid entries in the queue. 0 Pointer is valid (points to a valid qTD) 1 Pointer is invalid

2.2.2 qTD Token (Offset = 0x08)

The third longword of a queue element transfer descriptor contains most of the information the host controller requires to execute a USB transaction (the remaining endpoint and addressing information is specified in the QH).

Table 2-5. qTD Token (longword 2)

Field	Description
31 dt	Data toggle. This bit controls the data toggle sequence. This bit should be set for IN and OUT transactions and cleared for SETUP packets.
30–16 Total Bytes to Transfer	This field specifies the total number of bytes to be moved with this transfer descriptor. This field is decremented by the number of bytes actually moved during the transaction only on the successful completion of the transaction. If the value of this field is zero when the host controller fetches this transfer descriptor (and the active bit is set), the host controller executes a zero-length transaction and retires the transfer descriptor. **Note:** The maximum value software may store in this field is 4 K (0x1000). This is the maximum number of bytes a single page pointer can access. The host controller can accommodate larger transfers using multiple page pointers. But, for the purposes of this application note the transfer size is limited to 4 KB to simplify the data structures.
15 ioc	Interrupt on complete. If this bit is set, when this qTD is completed, the host controller should issue an interrupt at the next interrupt threshold.
14–10	Write as 000.
11–10 Cerr	Error counter. This field is a 2-bit down counter that tracks the number of consecutive errors detected while executing this qTD. The host controller decrements the count for each consecutive error and writes it back to the qTD if the transaction fails. Write this field as 0x3, to allow up to three retries for the transfer. If the counter counts from one to zero, the host controller marks the qTD inactive, sets the halted bit and error status bit for the error that caused Cerr to decrement to zero. An interrupt is generated if the USB error interrupt enable bit in the USBINTR register is set. Write-backs of intermediate execution state are to the QH's overlay area, not the qTD.

Table 2-5. qTD Token (longword 2) (continued)

<table>
<tr><th>Field</th><th colspan="2">Description</th></tr>
<tr><td>9–8
PID Code</td><td colspan="2">This field is an encoding of the token used for transactions associated with this transfer descriptor.
00 OUT token (generates token 0xE1)
01 IN token (generates token 0x69)
10 SETUP token (generates token 0x2D) (undefined if endpoint is an interrupt transfer type, for example. µFrame S-mask field in the queue head is non-zero.)
11 Reserved</td></tr>
<tr><td rowspan="9">7–0
Status</td><td colspan="2">This field is used by the host controller to communicate individual command execution states back to the host controller driver (HCD) software. This field contains the status of the last transaction performed on this qTD.</td></tr>
<tr><td>Bit</td><td>Status Field Description</td></tr>
<tr><td>7</td><td>Active. Set by software to indicate that the qTD has been initialized and is ready to use. Enables the execution of transactions by the host controller.</td></tr>
<tr><td>6</td><td>Halted. Set by the host controller during status updates to indicate that a serious error has occurred at the device/endpoint addressed by this qTD. This can be caused by one of the following:
• Babble
• The error counter reaching zero
• Reception of the STALL handshake from the device during a transaction
Any time a transaction results in setting the halted bit, the active bit is also cleared.</td></tr>
<tr><td>5</td><td>Data buffer error. Set by the host controller during status update to indicate that the host controller is either:
• Unable to keep up with the reception of incoming data (overrun)
• Unable to supply data fast enough during transmission (underrun)
If an overrun condition occurs, the host controller forces a time-out condition on the USB, invalidating the transaction at the source.</td></tr>
<tr><td>4</td><td>Babble detected. Set by the host controller during status update when babble is detected during the transaction. In addition to setting this bit, the host controller also sets the halted bit. Since babble is considered a fatal error for the transfer, setting the halted bit ensures that no more transactions occur because of this descriptor.</td></tr>
<tr><td>3</td><td>Transaction error. Set by the host controller during status update when the host did not receive a valid response from the device (time-out, CRC, or bad PID).</td></tr>
<tr><td>2</td><td>Missed microframe. This bit is ignored unless QH[EPS] indicates a full- or low-speed endpoint and the queue head is in the periodic list. This bit is set when the host controller detected a host-induced hold-off caused the host controller to miss a required complete-split transaction.</td></tr>
<tr><td>1</td><td>Split transaction state. Write as 0.</td></tr>
<tr><td>0</td><td>Ping state (P)/ERR. If the QH[EPS] field indicates a high-speed device and the PID code indicates an OUT endpoint, then this is the state bit for the ping protocol.
0 Do OUT. This value directs the host controller to issue an OUT PID to the endpoint.
1 Do Ping. This value directs the host controller to issue a PING PID to the endpoint.</td></tr>
</table>

2.2.3 qTD Buffer Page Pointer (Offset = 0x0C)

The qTD buffer page pointer is used to specify the memory address of the data buffer for the transfer.

Table 2-6. qTD Buffer Pointer

Field	Description
31–0 Buffer Pointer	Buffer pointer. Indicates the memory address for the data buffer used by the qTD. The host controller uses the first part of the address (bits 31–12) as a pointer to a 4-KB page and the lower part of the address (bits 11–0) as an index into the page. The host controller increments the index internally, but does not increment the page address. This is what determines the 4-KB transfer size limitation used for this application note. This means that the data buffer cannot span the 4KB page boundary. For applications where most of the transfers are small, the run-time buffer alignment can be avoided entirely by careful placement of the heap. If the heap space used to allocate memory for QHs, qTDs, and data buffers starts at the beginning of a 4-KB page and the application does not require more than 4 KB of data structures and buffers at a time, then the data buffer alignment is not a concern. If more than 4-KB is needed for data structures and data buffers, then there are a couple of ways to avoid a data buffer crossing a 4-KB page boundary: • Force the data buffers to a 4-KB alignment. However, this does not make very efficient use of memory unless most transfers are close to 4 KB. • Align each data buffer to its own size. For example, a 16-byte transfer is aligned to a 16-byte line address. This is a much more efficient use of memory for most applications. However, this adds a small amount of code overhead to handle the alignment.

2.3 Periodic Schedule

Figure 3 shows the organization of the periodic schedule. This schedule is used for all interrupt transfers.

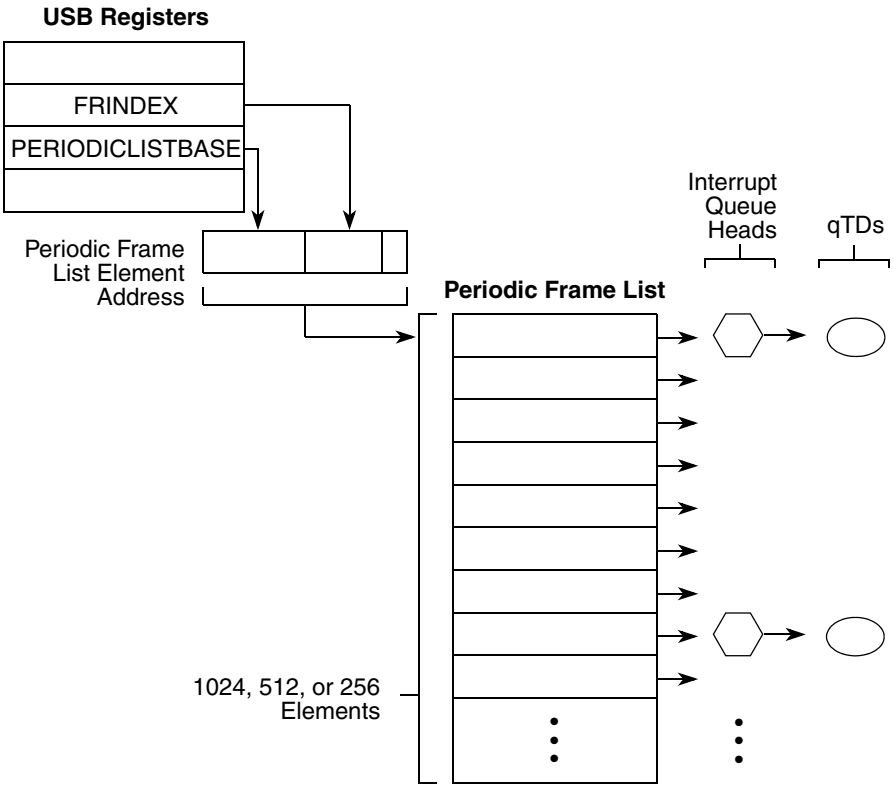

Figure 3. Periodic Schedule Organization

The USB module's PERIODICLISTBASE register and FRINDEX (bits 13–3) are combined to create a pointer into an array of pointers called the periodic frame list. The pointer into the periodic frame list is incremented every frame (1 ms).

The periodic frame list is a 4-KB page-aligned array of frame list link pointers. The length of the frame list is programmable using the USBCMD[FS] field. The EHCI specification supports a periodic frame list of 1024, 512, or 256 elements. The USB module also supports periodic frame list sizes of 128, 64, 32, 16, and 8 elements. In an embedded application where memory is limited, the smaller, non-EHCI compliant frame list sizes help to reduce the memory footprint needed for USB software.

2.3.1 Frame List Link Pointers

Frame list link pointers direct the host controller to the first work item in the periodic schedule for the current frame. The link pointers are aligned on longword boundaries within the periodic frame list. Figure 4 shows the format for the frame list link pointer.

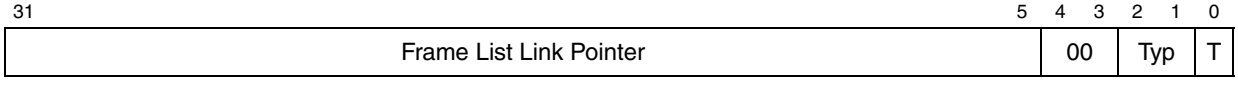

Figure 4. Frame List Link Pointer Format

Frame list link pointers always reference memory objects that are 32-byte aligned. The least significant bits in a frame list pointer key the host controller as to the type of object the pointer is referencing. For interrupt transfers the frame list link pointer always points to a QH (TYP = 0b01).

The least significant bit is the terminate bit, which is used to let the controller know when the contents of the pointer are valid. When set, the host controller ignores the entry in the frame list. When cleared, the frame list link pointer is used to access the referenced object, in this case a QH.

2.3.2 Periodic Schedule Traversal

The periodic schedule is enabled or disabled by the periodic schedule enable bit (USBCMD[PSE]). Modifications to the PSE bit are not necessarily immediate. The USBSTS[PS] bit reflects the current status of the periodic schedule. If PS is cleared, then the host controller does not attempt to traverse the periodic schedule. Likewise, if PS is set, at the start of each microframe (every 125 μs) the USB controller begins scheduling USB traffic by looking at the periodic schedule. The periodic traffic is guaranteed bus bandwidth, so the periodic schedule has priority over the asynchronous schedule. The controller uses the pointer into the periodic frame list to access the current frame list link pointer. If the T bit is cleared (indicating a valid pointer) the controller accesses the QH pointed to by the frame list link pointer.

The host controller schedules periodic traffic at the beginning of each microframe, but the pointer into the periodic frame list only increments for a full frame. Each item in the periodic frame list is accessed eight times per frame. The QH[uFrame S-mask] field value determines if traffic is scheduled for a given QH for each microframe.

If a QH is found that is valid for the current microframe, the host controller processes any qTDs in the list for the QH. When the qTDs for the first QH are complete, the controller checks to see if the first QH points to another QH. If so, it moves onto the second QH and process its qTDs. This continues until the last QH (a QH that doesn’t point to another QH) is reached. At this point the controller switches to the asynchronous schedule. Any time remaining in the microframe is used to process asynchronous schedule transfers.

If the periodic schedule is disabled or the current frame list link pointer has the T bit set, the entire frame can be used for asynchronous schedule traffic.

2.3.3 Adding Interrupt Queue Heads to the Periodic Schedule

When an interrupt device endpoint is being activated, the host software should create a QH for that endpoint and then link it to the periodic schedule. QHs are linked into the periodic schedule so they are polled at the appropriate rate. Each periodic frame list entry is used for 1 ms. So, for each periodic frame list entry that doesn’t point to a given QH there is 1 ms of delay.

For example, to get an 8ms poll rate for a FS/LS interrupt, every eighth entry in the periodic frame list is pointed to the QH (QH[uFrame S-mask] should be set).

Figure 5 shows an example of how the periodic frame list would be setup for a single interrupt that is polled every 4 ms. Every fourth entry in the periodic frame list is pointed to the same QH to create the desired poll rate.

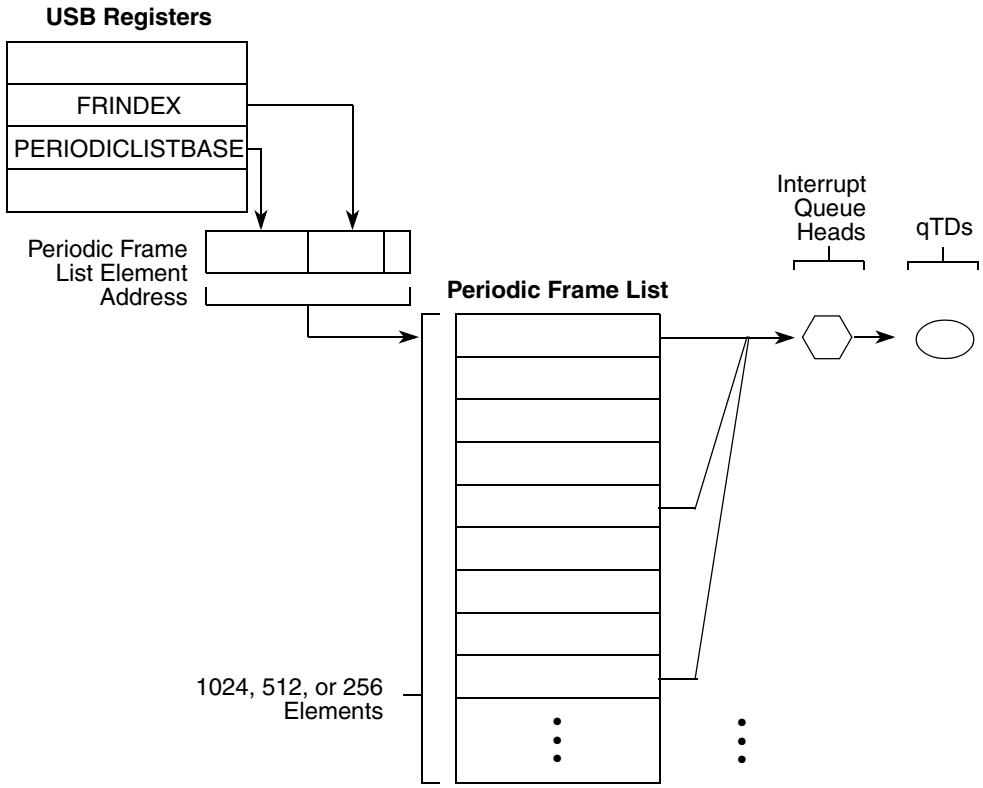

Figure 5. Periodic Frame List Example for an Interrupt Polled Every 4ms

If a HS interrupt with a poll rate that is less than 1 ms is needed, the QH[uFrame S-mask] value can create the desired poll rate. The QH is linked to every entry in the periodic frame list and then the spacing between the set bits in the QH[uFrame S-mask] field determine how frequently the interrupt occurs.

For example, a QH[uFrame S-mask] set to 0b01010101 results in an interrupt on every other microframe (every 250us). A value of 0b00010001 gives you an interrupt on every fourth microframe (every 500us).

2.4 Asynchronous Schedule

Figure 6 shows the organization for the asynchronous schedule. This schedule is used for all control and bulk transfers. Because control and bulk transfers do not get guaranteed USB bus bandwidth, the controller only uses this list when either:

- It reaches the end of the periodic list
- The periodic list is disabled
- The periodic list is empty

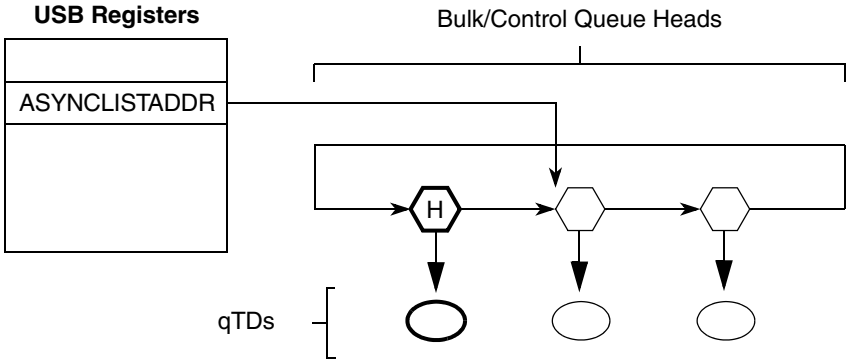

Figure 6. Asynchronous Schedule Organization

The asynchronous list is a simple circular list of queue heads, pointed to by the ASYNCLISTADDR register. Software should initialize the ASYNCLISTADDR to point to the first QH. The controller sets the ASYNCLISTADDR to point to the next QH as it processes the list. This way the controller returns to processing the asynchronous list (after periodic list processing) at the point it left off instead of returning to the beginning of the list each time. This implements a pure round-robin service for all QHs linked into the asynchronous list.

2.4.1 Asynchronous Schedule Traversal

The asynchronous schedule traversal is enabled or disabled by the asynchronous schedule enable bit (USBCMD[ASE]). Modifications to the ASE bit are not necessarily immediate. The USBSTS[AS] bit reflects the current status of the asynchronous schedule. If the AS bit is cleared, then the host controller simply does not try to access the asynchronous schedule. If the AS bit is set, the host controller uses the ASYNCLISTADDR register to traverse the asynchronous schedule.

When the host controller begins traversing the asynchronous schedule, it starts by using the value of the ASYNCLISTADDR register. It reads the first referenced QH and begins executing transactions and traversing the linked list as appropriate. When the host controller completes processing the asynchronous schedule, it retains the value of the last accessed QH's horizontal pointer in the ASYNCLISTADDR register. The next time the asynchronous schedule is accessed this is the first QH that is serviced. This provides round-robin fairness for processing the asynchronous schedule.

A host controller completes processing the asynchronous schedule when one of the following events occur:

- The end of a microframe is reached
- The host controller detects an empty list condition
- The asynchronous schedule is disabled (USBCMD[ASE] is cleared)

2.4.2 Adding Control or Bulk Queue Heads to the Asynchronous Schedule

One QH is added to the list every time a new device endpoint becomes active. Any traffic for that endpoint is then setup in a qTD and is linked to the appropriate QH. The host controller cycles through the QHs in a loop checking for active qTDs.

Because the asynchronous schedule processes QHs in a loop, the loop should not be broken while the schedule is active. This means that care should be taken when adding or removing a QH to or from the asynchronous schedule.

3 USB Host Example

Now let's look at working USB host software to get some real world examples of how the data structures are used. This section discusses the "m5329evb_usb_host_mouse_test" demo code in the MCF532XSC.zip file available on the Freescale's ColdFire website (http://www.freescale.com/coldfire). This example code:

1. Configures the USB module for host mode
2. Enumerates a USB mouse
3. Reads offset and button click information from the mouse

This application note uses the MCF532XSC.zip file as a specific example, but the example code can easily be ported to any other ColdFire device that includes an EHCI-compatible host or dual-role USB controller.

3.1 Control Queue Head

Once the host controller detects a new device attached to the USB, one of the first things to do is to setup a QH to handle the control traffic to enumerate the device. Figure 7 shows the simplified QH layout and the actual values used in the example software to initialize a QH for endpoint 0 immediately below each line. The qTD overlay area is removed for all of the examples, since they do not need to be initialized by software when creating the QH. The example software always clears the overlay area to make reading QH values easier. After the QH is initialized, it is the first and only QH in the asynchronous list, so the ASYNCLISTADDR register is written with the address of the QH.

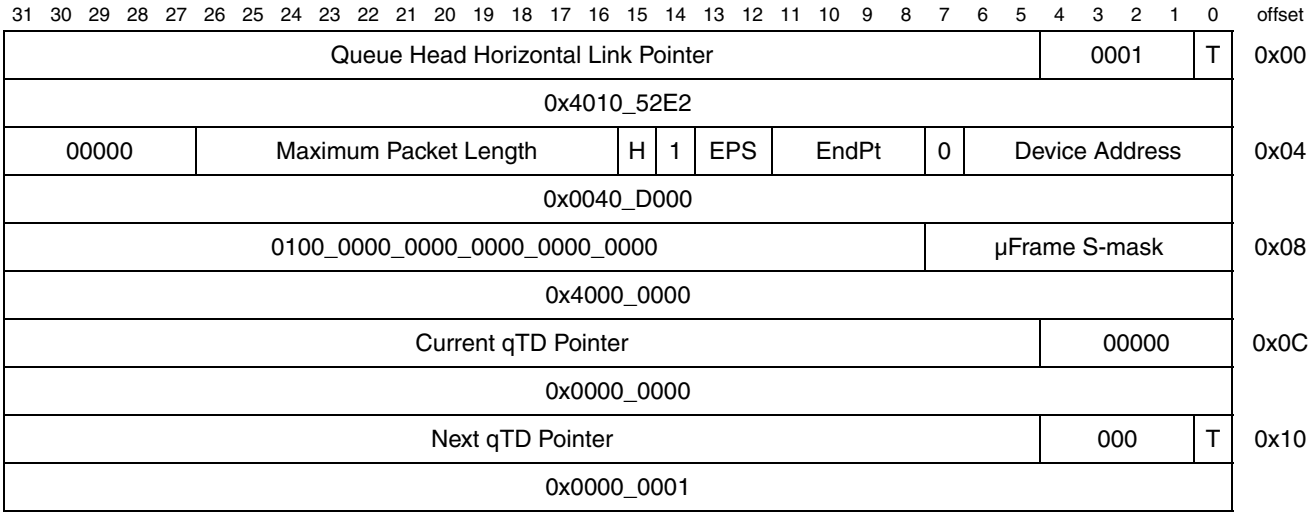

Figure 7. Endpoint 0 Control Queue Head Example

3.1.1 Example Queue Head Horizontal Link Pointer (Offset = 0x00)

Table 3-7. Queue Head Horizontal Link Pointer

Field	Description
31–5 QHLP	This value corresponds to the address of the QH. The QH itself resides at address 0x4010_52E0. So at this point there is a single QH that points back to itself.
4–1	0b0001
0 T	This bit is cleared indicating that the horizontal link pointer value is valid.

3.1.2 Endpoint Characteristics (Offset = 0x04)

Table 3-8. Endpoint Characteristics

Field	Description
31–27	0b00000
26–16 Maximum Packet Length	The max packet length is set with an initial value of 0x40. This value should be modified to reflect the actual max packet length of the device once the device descriptor is read.
15 H	This bit is set to mark the QH as the head of the asynchronous schedule.
14	0b1
13–12 EPS	0b01 indicates a low-speed endpoint, since the mouse is a low-speed device.
11–8 EndPt	The endpoint number is 0, since this is the default control endpoint for enumeration.

Table 3-8. Endpoint Characteristics (continued)

Field	Description
7	Write as 0.
6–0 Device Address	The device address is set to 0 initially. This is the default address used by a device before it has been assigned an address by the host. Once the set address command is sent to the device, this field should be updated to match the device's new address.

3.1.3 Endpoint Capabilities (Offset = 0x8)

Table 3-9. Endpoint Capabilities

Bit	Description
31–8	Write as 0x400_0000.
7–0 µFrame S-mask	Since this is a control endpoint the S-mask is cleared.

3.1.4 Current qTD Pointer (Offset = 0xC)

The current qTD pointer does not need to be initialized by software, but in this case the current qTD pointer has been written to 0. For debugging purposes this can make it a bit easier to read the QH in memory.

3.1.5 Next qTD Pointer (Offset = 0x10)

The next qTD pointer is written as 0x0000_0001, which indicates that the QH does not currently point to a valid qTD. Once qTDs for this endpoint are initialized this value should be updated to point to the first qTD in the linked list.

3.2 Get Device Descriptor qTDs

At this point the asynchronous list is setup and enabled, but the host controller does not request any bus cycles yet. To request USB traffic we need to create some qTDs. The first step of the USB enumeration process is to read in the device descriptor from the attached device, so we use this as an example for initializing qTDs. The device descriptor provides some basic information about the attached device including the maximum packet size the device supports for endpoint zero. After the max packet size is read in, the QH[Maximum Packet Length] field should be written to match the device's capability.

To read in the device descriptor, three different transfers are required:

1. A setup packet is sent with the get device descriptor command.
2. The host sends an IN packet to allow the device to send the descriptor.
3. The host issues a zero-length OUT packet to acknowledge reception of the descriptor.

3.2.1 Get Descriptor Setup Packet

Figure 8 shows the simplified qTD layout, along with the actual values used to send the get descriptor command in the software example. Since they are not used, the last four longwords of the qTD are removed for all of the examples. The example software always clears the last four longwords of a qTD to make reading qTD values easier.

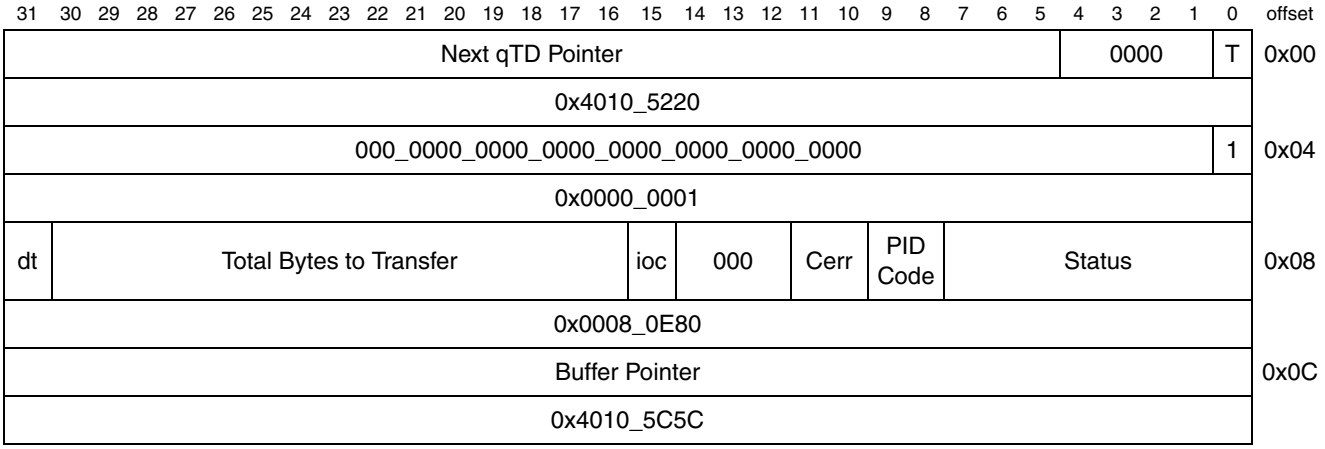

Figure 8. Get Descriptor SETUP Packet qTD Example

3.2.1.1 Next qTD Pointer (Offset = 0x00)

Table 3-10. qTD Next Element Transfer Pointer (longword 0)

Field	Description
31–5 Next qTD Pointer	This points to the IN packet qTD. In this case it is located at address 0x4010_5220. See Section 3.2.2, “IN Packet” for more details on the IN packet.
4–1	0b0000
0 T	This bit is cleared indicating that the next qTD pointer value is valid.

3.2.1.2 qTD Token (Offset = 0x08)

Table 3-11. qTD Token (longword 2)

Field	Description
31 dt	This bit is cleared since we are sending a SETUP packet.
30–16 Total Bytes to Transfer	Set to 0x8 since setup packets are always 8 bytes.
15 ioc	This bit is cleared. At the end of the OUT packet, we will request an interrupt to indicate the completion of the full get descriptor transaction.
14–10	0b000

Table 3-11. qTD Token (longword 2) (continued)

Field	Description
11–10 Cerr	Set to 0b11 to allow for up to three consecutive retries.
9–8 PID Code	0b10 indicates a SETUP PID.
7–0 Status	0x80 marks the qTD as active and ready for the host controller hardware to process.

3.2.1.3 qTD Buffer Page Pointer (Offset = 0x0C)

Table 3-12. qTD Buffer Pointer

Field	Description
31–0 Buffer Pointer	0x4010_5E2C is the location of the buffer that contains the data to transmit. 0x8006_0001 and 0x0000_4000. This translates to a GET_DESCRIPTOR command, where the type of descriptor is DEVICE and the length is 64 bytes.

3.2.2 IN Packet

Figure 9 shows the simplified qTD layout along with the actual values used in the software example to send the IN command to read the device descriptor.

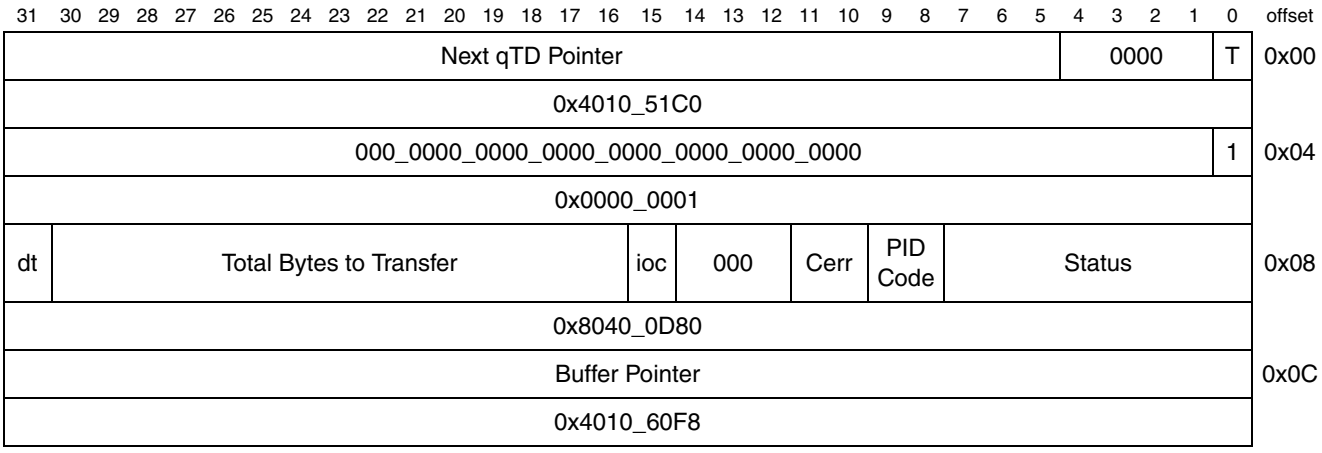

Figure 9. Get Descriptor IN Packet qTD Example

3.2.2.1 Next qTD Pointer (Offset = 0x00)

Table 3-13. qTD Next Element Transfer Pointer (longword 0)

Field	Description
31–5 Next qTD Pointer	This points to the OUT packet qTD. In this case it is located at address 0x4010_51C0. See Section 3.2.3, “OUT Packet” for more details on the OUT packet.
4–1	0b0000
0 T	This bit is cleared indicating that the next qTD pointer value is valid.

3.2.2.2 qTD Token (Offset = 0x08)

Table 3-14. qTD Token (longword 2)

Field	Description
31 dt	This bit is set since we are requesting an IN packet.
30–16 Total Bytes to Transfer	Set to 0x40. The mouse has a maximum packet length of eight. So, it sends the first eight bytes of the device descriptor. Since this is shorter than the requested length, the host controller interprets this as the end of the packet and does not request more data. The device descriptor is actually 18 bytes long, so the eight bytes we have read so far are not the full descriptor. However, this is enough of the descriptor to determine the maximum packet length the device can support on endpoint zero (the eighth byte of the device descriptor is the max packet length). After this value has been programmed into the QH[Maximum Packet Length] field the host controller recognizes eight bytes as a full packet from the device and responds accordingly. For example, the device descriptor is read a second time in the enumeration process. This time the host requests three different IN packets. The first two are the full 8 bytes. Then on the final IN the device just sends two bytes of data. Since this is less than the max packet length, the host recognizes this as the end of the packet.
15 ioc	This bit is cleared. At the end of the OUT packet, we will request an interrupt to indicate the completion of the full get descriptor transaction.
14–10	0b000
11–10 Cerr	Set to 0b11 to allow for up to three consecutive retries.
9–8 PID Code	0b01 indicates an IN PID.
7–0 Status	0x80 marks the qTD as active and ready for the host controller hardware to process.

3.2.2.3 qTD Buffer Page Pointer (Offset = 0x0C)

Table 3-15. qTD Buffer Pointer

Field	Description
31–0 Buffer Pointer	0x4010_60F8 is the location of the buffer that the host controller writes the data to as it receives it from the device.

3.2.3 OUT Packet

Figure 10 shows the simplified qTD layout along with the actual values used in the software example to send the OUT to acknowledge reception of the device descriptor.

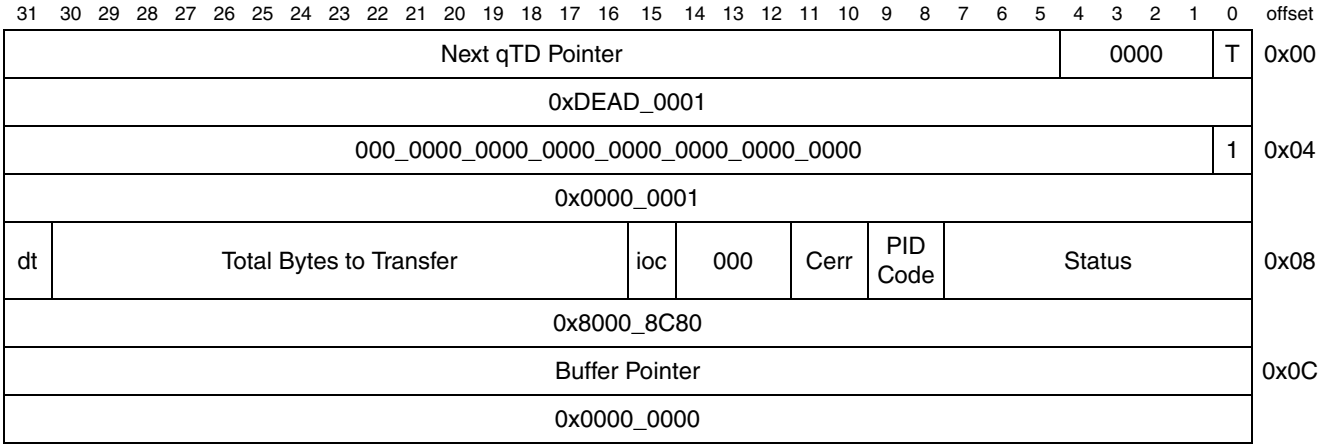

Figure 10. Get Descriptor OUT Packet qTD Example

3.2.3.1 Next qTD Pointer (Offset = 0x00)

Table 3-16. qTD Next Element Transfer Pointer (longword 0)

Field	Description
31–5 Next qTD Pointer	The OUT packet is the last qTD needed to complete the get descriptor transaction. There is no other traffic to request at this time. So, the next qTD pointer is an invalid value. The example code uses a value of 0xDEAD_0001, so that it is easy to recognize the end of a qTD chain.
4–1	0b0000
0 T	This bit is set indicating that the next qTD pointer value is invalid.

3.2.3.2 qTD Token (Offset = 0x08)

Table 3-17. qTD Token (longword 2)

Field	Description
31 dt	This bit is set since we are sending an OUT packet.
30–16 Total Bytes to Transfer	Set to 0x0. The OUT is a zero length transaction that is only used as an acknowledge. So, no actual data is sent from the host.
15 ioc	This bit is set to request an interrupt when the full get descriptor transaction is complete.
14–10	0b000
11–10 Cerr	Set to 0b11 to allow for up to three consecutive retries.
9–8 PID Code	0b00 indicates an OUT PID.
7–0 Status	0x80 marks the qTD as active and ready for the host controller hardware to process.

3.2.3.3 qTD Buffer Page Pointer (Offset = 0x0C)

Table 3-18. qTD Buffer Pointer

Field	Description
31–0 Buffer Pointer	The buffer pointer is cleared. Since no data is being sent a data buffer is not needed.

3.3 Interrupt Queue Head

After the descriptors have been read from the mouse, the example software configures a QH to communicate with an interrupt endpoint. The mouse uses the interrupt endpoint to return x- and y- offset information along with button click data and scroll wheel information (the format of the data returned can vary from mouse to mouse).

Figure 11 shows the simplified QH layout, along with the values used in the example software to initialize an interrupt QH for endpoint one.

31–27	26–16	15	14	13–12	11–8	7	6–0	offset
Queue Head Horizontal Link Pointer							0001 T	0x00
0x4010_4243								
00000	Maximum Packet Length	H	1	EPS	EndPt	0	Device Address	0x04
0x0008_5102								
0100_0000_0000_0000_0000_0000							μFrame S-mask	0x08
0x4000_0001								
Current qTD Pointer							00000	0x0C
0x0000_0000								
Next qTD Pointer							000 T	0x10
0x0000_0001								

Figure 11. Interrupt Queue Head Example

3.3.1 Example Queue Head Horizontal Link Pointer (Offset = 0x00)

Table 3-19. Queue Head Horizontal Link Pointer

Field	Description
31–5 QHLP	This value corresponds to the address of the QH. The QH itself resides at address 0x4010_4240.
4–1	0b0001
0 T	This bit is set indicating that the horizontal link pointer value is invalid. Since this QH goes into the periodic schedule, it is not used in a circular linked list like QHs for the asynchronous schedule.

3.3.2 Endpoint Characteristics (Offset = 0x04)

Table 3-20. Endpoint Characteristics

Field	Description
31–27	0b00000
26–16 Maximum Packet Length	The max packet length is set to 0x08. This corresponds to the maximum packet size in the device's endpoint descriptor. A USB mouse always has a maximum packet size of 0x8.
15 H	This bit is cleared since this bit is not used for the periodic schedule.
14	0b1
13–12 EPS	0b01 indicates a low-speed endpoint, since the mouse is a low-speed device.
11–8 EndPt	The endpoint number is 1, since this is the endpoint used by the mouse for interrupt traffic.

Table 3-20. Endpoint Characteristics (continued)

Field	Description
7	Write as 0.
6–0 Device Address	The device address is set to 2. This is the address that the example software assigns to the device during the enumeration process.

3.3.3 Endpoint Capabilities (Offset = 0x8)

Table 3-21. Endpoint Capabilities

Field	Description
31–8	Write as 0x400_0000.
7–0 μFrame S-mask	Since this QH is being used for a LS interrupt endpoint, the S-mask value is set to one.

3.3.4 Current qTD Pointer (Offset = 0xC)

The current qTD pointer does not need to be initialized by software, but in this case the current qTD pointer is written to 0. For debugging purposes this can make it a bit easier to read the QH in memory.

3.3.5 Next qTD Pointer (Offset = 0x10)

The next qTD pointer is written as 0x0000_0001. This indicates that the QH does not point to a valid qTD. After a qTD for this endpoint is created, this field should be written to point to the first qTD.

3.4 Periodic Frame List Initialization

At this point the example software configures the periodic frame list. The periodic_schedule_init function performs the following:

- Initializes the USBCMD[FS] field, which defines the size of the periodic frame list
- Allocates memory for the periodic frame list
- Fills the frame list with longwords of 0x0000_0001, indicating that the frame list pointers are currently invalid
- The PERIODICLISTBASE register is set to point to the frame list
- The periodic schedule is enabled

Now, the interrupt QH needs to be linked to the periodic frame list to create the desired polling rate. By default the example software sets the periodic frame list size to 32. The first entry in the frame list is set to 0x4010_4242 to point it to the interrupt QH.

This means that the device is polled every 32 ms (1 ms per frame list pointer × 32 frame list pointers). The poll rate could be increased by pointing more of the frame list pointers to the interrupt QH. The frame list size could be decreased to increase the poll rate as well.

3.5 Interrupt qTD

The periodic schedule is running at this point, but a qTD is needed to move data. The example code uses a single qTD for interrupt traffic. The qTD is setup to accommodate 20 packets from the USB mouse. Once the 20 packets have been received, the total bytes to transfer field and the buffer pointer are re-initialized to their original values, so that the qTD can be used again in a continuous loop.

Figure 12 shows the actual interrupt qTD values used by the example code.

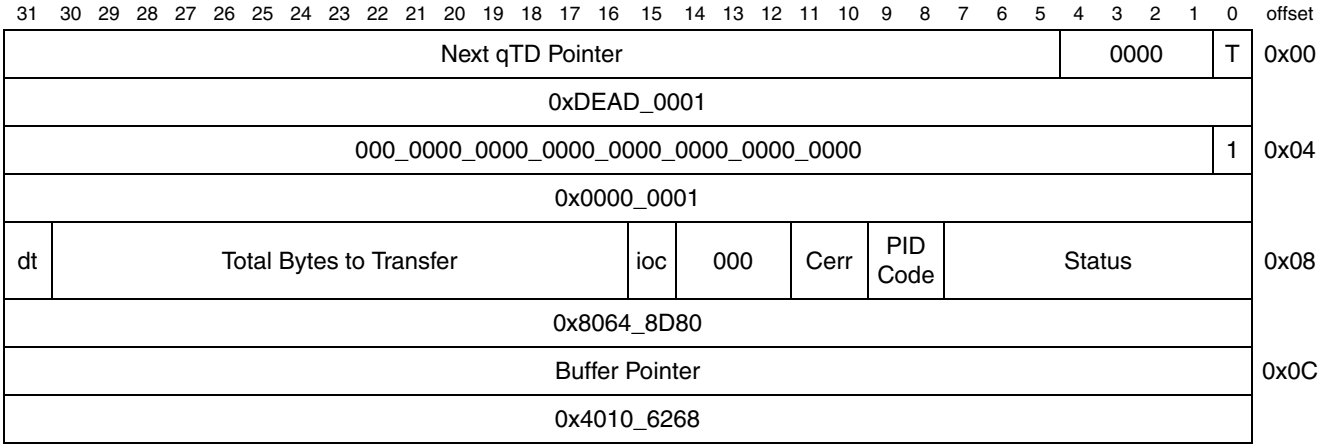

Figure 12. Interrupt qTD Example

3.5.1 Next qTD Pointer (Offset = 0x00)

Table 3-22. qTD Next Element Transfer Pointer (longword 0)

Field	Description =
31–5 Next qTD Pointer	This qTD is the only one used for this QH. Since a linked list of qTDs is not needed, the next qTD pointer value is invalid.
4–1	0b0000
0 T	This bit is set indicating that the next qTD pointer value is invalid.

3.5.2 qTD Token (Offset = 0x08)

Table 3-23. qTD Token (longword 2)

Field	Description
31 dt	This bit is set since we are sending an IN packet.
30–16 Total Bytes to Transfer	The total bytes to transfer is equal to the size of 20 of packets from the mouse. The packet size is defined by byte 4 of the device's endpoint descriptor. Since the packet size can vary depending on the mouse, the total bytes to transfer can vary as well. In this case, the mouse connected to the M5329EVB returns five bytes of data for each IN packet. So, the total bytes to transfer is 100 (0x64).

Table 3-23. qTD Token (longword 2) (continued)

Field	Description
15 ioc	This bit is set to request an interrupt when the transaction is complete.
14–10	0b000
11–10 Cerr	Set to 0b11 to allow up to three consecutive retries.
9–8 PID Code	0b01 indicates an IN PID.
7–0 Status	0x80 marks the qTD as active and ready for the host controller hardware to process.

3.5.3 qTD Buffer Page Pointer (Offset = 0x0C)

Table 3-24. qTD Buffer Pointer

Field	Description
31–0 Buffer Pointer	0x4010_6268 is the location of the buffer where the data is written as it is read from the device.

4 Additional Information

Table 25 lists some additional resources that can be used to find more information on the USB and EHCI.

Table 25. Additional Resources

Document	Website	Description
Universal Serial Bus Specification	http://www.usb.org/developers/docs	Official USB specification
Enhanced Host Controller Interface Specification	http://www.intel.com/technology/usb/spec.htm	Official EHCI specification.

THIS PAGE IS INTENTIONALLY BLANK

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3520
Rev. 0
09/2007

RoHS-compliant and/or Pb-free versions of Freescale products have the functionality and electrical characteristics as their non-RoHS-compliant and/or non-Pb-free counterparts. For further information, see http://www.freescale.com or contact your Freescale sales representative.

For information on Freescale’s Environmental Products program, go to http://www.freescale.com/epp.

freescale™
semiconductor