



Bachelor's Thesis

Development of a UHCI driver for a x86-based Operating System

submitted by

Sandro Jose Leal Miudo

from Neuss

Department Operating Systems
Prof. Dr. Michael Schöttner
Heinrich Heine University Düsseldorf

22. April 2024

Primary Reviewer: Prof. Dr. Michael Schöttner
Secondary Reviewer: Prof. Dr. Stefan Conrad
Supervisor: Prof. Dr. Michael Schöttner

Contents

1	Introduction	1
2	Basics	2
2.1	Universal Serial Bus	2
2.1.1	Architecture	2
2.1.2	USB Communication Flow	4
2.1.3	USB Protocol	7
2.1.4	USB Transfers	11
2.2	Universal Serial Bus Device	13
2.2.1	Architecture	13
2.2.2	USB Device Request	14
2.2.3	Standard Requests	16
2.2.4	Standard Descriptors	19
2.3	Universal Host Controller Interface	27
2.3.1	PCI Enumeration	27
2.3.2	I/O Space	28
2.3.3	Internals	33
2.3.4	Stack	34
3	USB System Implementation	48
3.1	Design/Architecture	48
3.1.1	UHCI Custom QH Field	48
3.1.2	UHCI Schedule	49
3.1.3	System Overview	50
3.1.4	Core System Component Analysis	52
3.1.5	Driver System Component Analysis	59
3.1.6	Interaction between Core System Components	64
3.1.7	Functional Analysis of Core System Components	90
3.1.8	Functional Analysis of Driver System Components	108
3.2	Implementation	118
3.2.1	Utility	118
3.2.2	Core System Component Implementation	119
3.2.3	Driver System Component Implementation	154
3.2.4	Changes in hhuOS	167
4	Evaluation	173
4.1	Experimental Setup	173
4.2	Evaluation of Components	174

5 Conclusion	181
Bibliography	183
List of Figures	185
List of Tables	187

Chapter 1

Introduction

The primary objective of this thesis is to develop a specialized Universal Host Controller Interface (UHCI) driver intended for an x86 operating system. The selected experimental platform for this endeavor is hhuOS, 'a small operating system' [1] conceptualized by the Operating Systems Group at Heinrich Heine University. Notably, hhuOS features an array of functionalities including paging, networking, among others.

Rather than confining its focus solely to the UHCI driver, this thesis adopts a broader perspective by incorporating the entirety of the Universal Serial Bus (USB) system into hhuOS. By contextualizing the UHCI driver within the overarching framework of the USB system, this study aspires to provide a comprehensive understanding of USB and its intricate implementation nuances.

The thesis follows a systematic approach, initiating with a methodical exposition of fundamental domain-specific concepts pertinent to USB, USB devices, and the UHCI 2. Subsequently, the focus transitions to the USB System Implementation, where initial scrutiny is devoted to the design and architecture, succeeded by an exhaustive examination of the practical implementation within hhuOS 3.

The ensuing evaluation phase serves to expound upon the insights and discoveries derived from the integration process, furnishing a critical appraisal of the system's performance and operational functionality 4. Lastly, the conclusion synthesizes the pivotal findings, offering a comprehensive summation of the study's outcomes while also delineating improvements for the current implementation and proposing future avenues for exploration and development 5.

Chapter 2

Basics

This chapter will embark on a comprehensive exploration of the domain-specific intricacies inherent in the USB architecture. It will commence with an in-depth analysis of the USB's structural framework, encompassing its architecture, communication flow, protocol specifications, and data transfer mechanisms. Subsequently, attention will be directed towards the assembly and functionality of USB devices, delving into the intricacies of requesting specific device details and elucidating the standard requests mandated for device responsiveness.

Furthermore, this examination will extend to scrutinizing the descriptors returned by USB devices in response to these requests, shedding light on their structural composition and functional significance. Lastly, a meticulous examination of the UHCI will ensue. This exploration will encompass various facets, including its enumeration as a PCI device, delineation of its I/O space, elucidation of its internal operational mechanisms, and a comprehensive overview of the UHCI stack's architectural underpinnings.

2.1 Universal Serial Bus

2.1.1 Architecture

To commence, it is imperative to delineate an understanding of the fundamental architecture inherent in the entirety of the USB system. This comprehensive system is characterized by the presence of three primary components, wherein each assumes a pivotal role crucial to its operational efficacy.

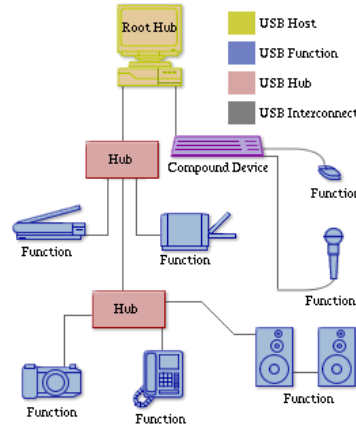


Figure 2.1: USB System Illustration [2]

In the hierarchy of the USB system, the foundational components consist of devices categorized into two primary types: hubs and functions. While the USB specification delineates a plethora of device classes, this scholarly inquiry centers its attention predominantly on two distinct classifications. These classifications encompass low speed functions, which operate at speeds of up to 1.5 Mb/s, and full speed functions, distinguished by their capacity for data transmission at rates of up to 12 Mb/s. It is noteworthy to mention that within the framework of UHCI implementation, support is not extended to two additional device types: High speed functions, boasting speeds of up to 480 Mb/s, and super speed functions, capable of achieving data rates of up to 5 Gb/s. [2], [3]

In contrast, hubs exhibit a distinct operational paradigm in comparison to functions. Each hub is endowed with multiple downstream ports, serving as conduits for the attachment of additional functions to the system. This hierarchical configuration affords scalability to the USB network, thereby accommodating the incorporation of a greater number of devices. However, it is imperative to underscore that while the stacking of hubs is feasible, it is constrained by a specification limitation of up to seven tiers. [4, sec. 4.1.1], [2]

Following the exhaustive examination of USB devices and hubs, the subsequent essential component integral to the USB system is the USB host. Serving as the linchpin for connectivity with the attached USB devices, the Universal Host Controller (UHC) assumes a central role in this context. Within the scope of this thesis, particular emphasis is placed on the UHCI, a component slated for comprehensive exploration in subsequent sections. Furthermore, completing the architectural framework is the USB interconnect, which functions as a vital conduit essential for facilitating communication between USB devices and the USB host. [2]

This intricate interconnection of devices constitutes the USB topology, delineating the hierarchical arrangement and data transmission within a USB system. To gain a deeper understanding of the construction of this topology, refer to the illustration provided below.

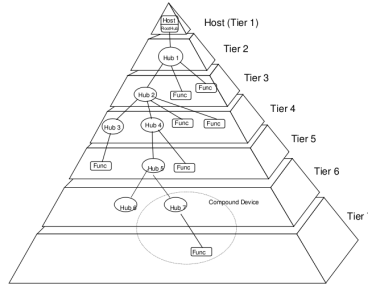


Figure 2.2: USB Topology [4, sec. 4.1.1]

In the depicted schematic, the initial tier of the USB network comprises the downstream ports seamlessly integrated into the UHC. Subsequent expansion of the USB network entails either appending functions, thereby halting tier growth, or interfacing hubs to extend the network by an additional tier. As previously elucidated, the hierarchical structure of this network is constrained to seven tiers, signifying that downstream devices attached to a hub at tier 7 will not function. [4, sec. 4.1.1]

2.1.2 USB Communication Flow

Subsequently, it is imperative to comprehend the underlying concept governing the communication among these components. The communication flow in USB architecture is facilitated through a fundamental component known as a 'pipe'. These pipes serve as data channels, enabling the exchange of information between the host and the device. On the host side, a pipe is represented by a consecutive block of memory, serving as one end of the data channel. The counterpart on the device side is the endpoint, positioned at the termination point of the data channel. While a detailed examination of endpoints will be conducted later, for present purposes, envision an endpoint as a component facilitating communication with the device. [4, sec. 5.3]

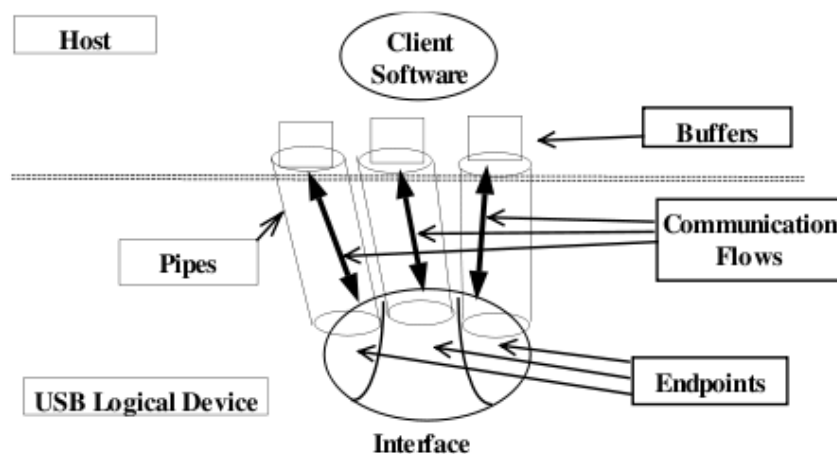


Figure 2.3: USB Communication Flow [4, sec. 5.3]

The concept of interfaces, as depicted in the aforementioned figure, can be grasped as containers encapsulating one or multiple endpoints, thereby enabling access by a single driver. It is imperative to recognize that each interface manages precisely one type of

logical connection. However, for comprehension at this stage, the elucidation provided suffices to comprehend the interconnection of these components.

Lastly, it is essential to acknowledge the significance of the 'default control pipe', which utilizes endpoint 0 to establish the data channel between the host and the device. This endpoint's implementation is mandatory for all devices, as it is instrumental in initiating and managing requests to the device. Access to this default control pipe is always available once the device is powered on and enters its default state. [4, sec. 5.3.1]

Before proceeding to the subsequent subsection, it is imperative to underscore a fundamental characteristic of the USB system: its inherent nature as a polled bus. This signifies that data transfer operations are invariably initiated by the UHC, exemplifying a central control mechanism governing communication across the USB network. [2]

An integral aspect of comprehending the communication protocol of USB hosts with connected devices involves an understanding of 'transfers'. This section will delve into the various types of transfers, their limitations, and how they are constructed.

USB transfers can be conceptualized as containers encapsulating internal components known as USB transactions, which form the underlying protocol for device communication. Each transaction comprises three distinct packets, each serving a unique function. [2]

Firstly, the token packet encompasses all pertinent information regarding the target device, endpoint, directionality, and other relevant parameters. Subsequently, the data packet carries the actual data to be transmitted across the designated endpoint. Lastly, the handshake packet serves to indicate the success or failure of the transfer. [2]

In elucidating the intricacies of transfer types, USB architecture supports four primary transfer types: **Bulk**, **Control**, **Interrupt**, and **Isochronous** transfers. [2], [4, sec. 5.4] For the purpose of this thesis, emphasis will be placed on the former three, and thus, discussion regarding the internals of isochronous transfers will be limited. Nonetheless, an abstract overview and examination of the limitations of isochronous data transfer will be provided.

For readers seeking a deeper understanding of isochronous transfers, supplementary resources are recommended for further exploration [2], [4, sec. 5.6], [4, sec. 8.5.4]. However, the subsequent discussion will provide a comprehensive understanding of transfer types and their implications within the context of USB communication protocols.

Control-Transfer

The control transfer within USB communication serves as a pivotal mechanism for device control, encompassing configuration, command issuance, status retrieval, and related functionalities. Central to this transfer paradigm is the default pipe, constituting a control pipe present in every USB device. Its primary role involves the initialization and configuration of the device, encompassing tasks such as reading internal information and setting parameters crucial for proper device operation. [4, sec. 5.5], [2]

In contrast to subsequent transfer types, the control transfer typically involves multiple stages. The initial stage, known as the **setup stage**, comprises a setup transaction wherein the host communicates specific parameters to the device. These parameters in-

clude the transfer direction, data length, and other essential configurations to be further elucidated. Subsequently, the **data stage** follows, comprising one or more data transactions aimed at transmitting the payload. Finally, a **status stage** concludes the transfer, serving to verify the success or failure of the operation. Notably, this status transaction invariably transmits a zero-length data packet. [4, sec. 8.5.2], [2]

During data transmission, it is imperative to fill the data to be transmitted up to the payload capacity supported by the endpoint. If the transmitted data falls short of the endpoint's maximum payload capacity, padding the remaining space with zeros is unnecessary. Instead, sending less than the maximum payload signifies the termination of data transmission. For instance, if 50 bytes of data are to be transferred, and the endpoint's maximum data payload is 8 bytes, six transactions containing exactly 8 bytes each would precede the transmission of the final 2 bytes, signaling the end of data transmission. Attempts to transmit data exceeding the maximum payload capacity result in an immediate return and termination of the transfer. [4, sec. 5.5.3], [2]

The maximum data payload permissible for a control endpoint is contingent upon the device's speed. Low speed devices can transmit or receive a maximum of 8 bytes, while full speed devices can accommodate payloads of 8, 16, 32, or 64 bytes. Additionally, it is noteworthy that the setup transaction's data payload is consistently fixed at 8 bytes, aligning with the size of the 'DeviceRequest', a standard requirement for communication over the control pipe. Thus, the lower threshold for low speed devices is set at 8 bytes to accommodate this essential protocol constraint. [4, sec. 5.5.3], [2]

Bulk-Transfer

The bulk transfer protocol in USB communication enables the exchange of data between devices without the assurance of allocated bandwidth. Typically utilized by devices such as storage devices, exemplified by USB sticks, bulk endpoints facilitate the transmission and reception of large volumes of data. [4, sec. 5.8], [2]

One noteworthy characteristic of bulk transfers is the absence of guaranteed bandwidth, signifying a lack of predetermined data rate. This low bandwidth arises from the UHC's traversal structure, which interfaces with the bulk data structure relatively late in the transfer process. A more in-depth understanding of this dynamic is elucidated in the examination of the UHCI implementation.

The maximum data payload size supported by bulk endpoints varies among devices. Full speed device endpoints can accommodate payloads ranging from 8 to 64 bytes. However, unlike control endpoints, bulk endpoints remain inaccessible to low speed devices. Consequently, for communication with the USB host, all devices must operate at least at full speed. [4, sec. 5.8.3], [2]

Similar to control transfers, bulk transfers are susceptible to failure if the transmitted data exceeds the specified maximum data payload of the bulk endpoint. This underscores the significance of adhering to endpoint specifications to ensure the successful transmission of data. [2]

Interrupt-Transfer

Subsequently, interruption transfers will be explored, a critical aspect within USB communication protocols. These transfers are primarily employed for transmitting data at predetermined intervals, commonly utilized in Human Interface Devices (HID) such as mice and keyboards. Distinguished from control and bulk transfers, which serve varied purposes, interruption transfers fulfill a specialized function, facilitating periodic data exchange. [4, sec. 5.7], [2]

One significant aspect of interrupt transfers is their maximum data payload size, which varies based on the speed of the device. For full speed devices, the maximum data payload can range from 0 to 64 bytes, while for low speed devices, it ranges from 0 to 8 bytes. Unlike control and bulk transfers, the data payload size for interrupt transfers is not fixed but remains constant throughout the device configuration. [4, sec. 5.7.3], [2]

It's crucial to note that interrupt transfers adhere to the same termination criteria as other transfer types. If more data is sent to the endpoint than the maximum payload supported by the endpoint, the transfer will be terminated. [2]

Isochronous-Transfer

The final transfer type, presented here for the sake of comprehensive understanding, is the isochronous transfer, which, although not essential for further discourse, may pique the interest of inquisitive readers. Isochronous transfers are designed for transmitting data streams at consistent rates, albeit without guaranteeing data integrity. Unlike other transfer types, isochronous transfers do not involve handshaking protocols and consequently cannot experience stalling. These transfers are primarily utilized by devices necessitating real time transmission of data streams, such as audio devices like microphones. [4, sec. 5.6], [2]

In contrast to other transfer modes, isochronous transfers for full speed devices enable the transmission of data payloads up to the maximum capacity during any transaction, which, in such cases, can reach up to 1023 bytes. This capability is exclusive to full speed devices or higher, as low speed devices lack the capacity to utilize isochronous endpoints. [4, sec. 5.6.3], [2]

2.1.3 USB Protocol

Having acquired a thorough comprehension of each transfer type and its abstract functionalities, the exploration now proceeds into the foundational elements of USB communication, namely packets. It is noteworthy that each transaction within any USB transfer comprises a total of three packets. [2]

The first packet, known as the token packet, plays a pivotal role by encapsulating essential control information required for the transfer process. This control information encompasses various elements, each serving a distinct purpose.

Address Field

The address field, comprising 7 bits, serves as a crucial identifier for the UHC to determine the target device for communication. In the default state, the address field is invariably set to 0, serving as the reserved default address. This ensures that no device can be assigned the address of 0 during configuration. With 7 bits, the address field can accommodate up to $2^7 - 1$ devices, totaling 127 devices. [4, sec. 8.3.2.1], [2]

Addr0	Addr1	Addr2	Addr3	Addr4	Addr5	Addr6
-------	-------	-------	-------	-------	-------	-------

Table 2.1: Address Field [4, sec. 8.3.2.1]

Endpoint Field

The endpoint field, spanning 4 bits, specifies which endpoint the device should utilize. Endpoint 0 is mandatory for all devices, and setting the endpoint field to 0 should always succeed. Low speed devices can support only two additional endpoints, while full speed devices can accommodate up to 15 additional endpoints, constrained by the size of the endpoint field $2^4 - 1$. [4, sec. 8.3.2.1], [2]

Endp0	Endp1	Endp2	Endp3
-------	-------	-------	-------

Table 2.2: Endpoint Field [4, sec. 8.3.2.2]

Packet Identifier Field

The Packet Identification (PID) constitutes a fundamental element within the USB protocol, providing essential distinctions among various packet types during transmission. Comprising 8 bits, the PID assumes a pivotal role as a primary identifier, facilitating the differentiation between setup packets, data packets, and acknowledgment packets. Within the PID structure, the initial four bits are dedicated to encoding the packet type, while the subsequent four bits serve as a complement utilized for error detection and correction purposes. [4, sec. 8.3.1], [2]

The complete range of PID values is organized into a comprehensive table, facilitating easy reference and interpretation of packet types based on their PID encoding.

PID-Type	PID-Name	PID:3-0	Description
Token	OUT	0001b	host-to-function
	IN	1001b	function-to-host
	SOF	0101b	start-of-frame
	SETUP	1101b	setup via control pipe
Data	DATA0	0011b	even data packet
	DATA1	1011b	odd data packet
Handshake	ACK	0010b	successful
	NAK	1010b	not ready for transmission
	STALL	1110b	endpoint halted

Table 2.3: Packet Identifier Types [4, sec. 8.3.1]

The tabular representation illustrates four distinct packet identifications employed during the transmission of token packets within USB communication. Each packet identification serves a specific function, contributing to the efficient exchange of data between hosts and devices.

Initially, the OUT/IN packet identification indicates whether the forthcoming data packet pertains to a host-to-function or function-to-host transmission. Following this, the SETUP PID signifies the imminent transmission of a setup packet, which encompasses device-specific requests and configurations. [4, sec. 8.3.1], [2]

Following the initial token packet, the subsequent data packet within the USB transaction is distinguished by the DATA0 and DATA1 packet identifications. These identifiers play pivotal roles in discerning the nature of the second packet in the transaction. The DATA0 PID indicates an even packet transmission, where the sender's toggle bit is set to 0, while the DATA1 PID denotes an odd data packet transmission, with the sender's toggle bit set to 1. [4, sec. 8.3.1], [2]

Finally, the handshake packet, representing the third and thus final component within a transaction, incorporates diverse responses to the transmitted data packet. These responses encompass ACK, denoting successful transmission; NAK, indicative of endpoint unavailability or busyness; and STALL, signaling a substantial error necessitating resolution prior to further progression. [4, sec. 8.3.1], [2]

Data Synchronization

The data toggle bit plays a crucial role in ensuring data synchronization between the sender and receiver during USB communication. Its primary function is to facilitate error detection and synchronization, allowing both parties to determine if the data transfer

was successful or if errors occurred necessitating retransmission. The toggle bit exists in two states: 0 and 1, corresponding to the transmission of DATA1 and DATA0 packets, respectively. Upon each successful transmission, the toggle bit toggles to the opposite state, ensuring synchronization between the sender and receiver. [4, sec. 8.6], [2]

In the event of a successful transmission, the sender initially transmits the data packet with the corresponding PID of DATA0 or DATA1 based on the current state of the toggle bit. Upon receiving the data packet, the receiver toggles its internally stored toggle bit and verifies the transmission's success. Subsequently, a handshake packet indicating successful transmission is sent back to the sender. The sender, upon receiving the acknowledgment, toggles its toggle bit to prepare for the next transmission. [4, sec. 8.6.2], [2]

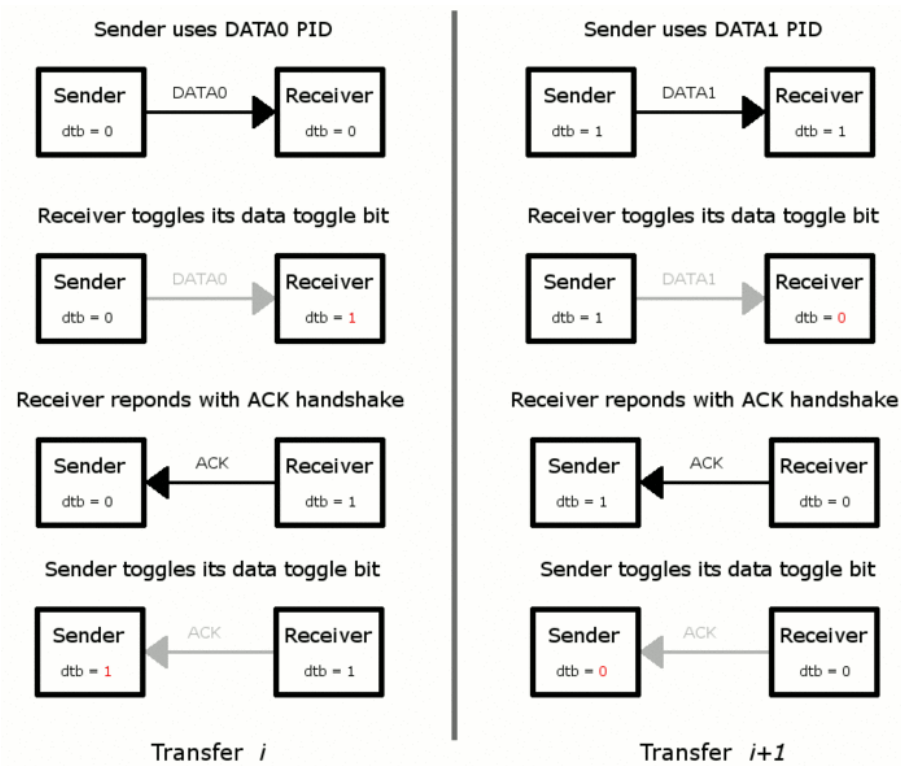


Figure 2.4: Successful Data Transmission [2]

Conversely, in the case of an erroneous transmission, where the data packet is not successfully received, the receiver does not toggle its toggle bit. Instead, an appropriate error indication is sent in the handshake packet back to the sender. Upon detecting the error, the sender maintains the current state of the toggle bit and retransmits the data packet with the same PID to attempt synchronization with the previous transmission. This mechanism allows for the resynchronization of data transmission in case of errors, ensuring reliable communication between the sender and receiver. [2], [4, sec. 8.6.3]

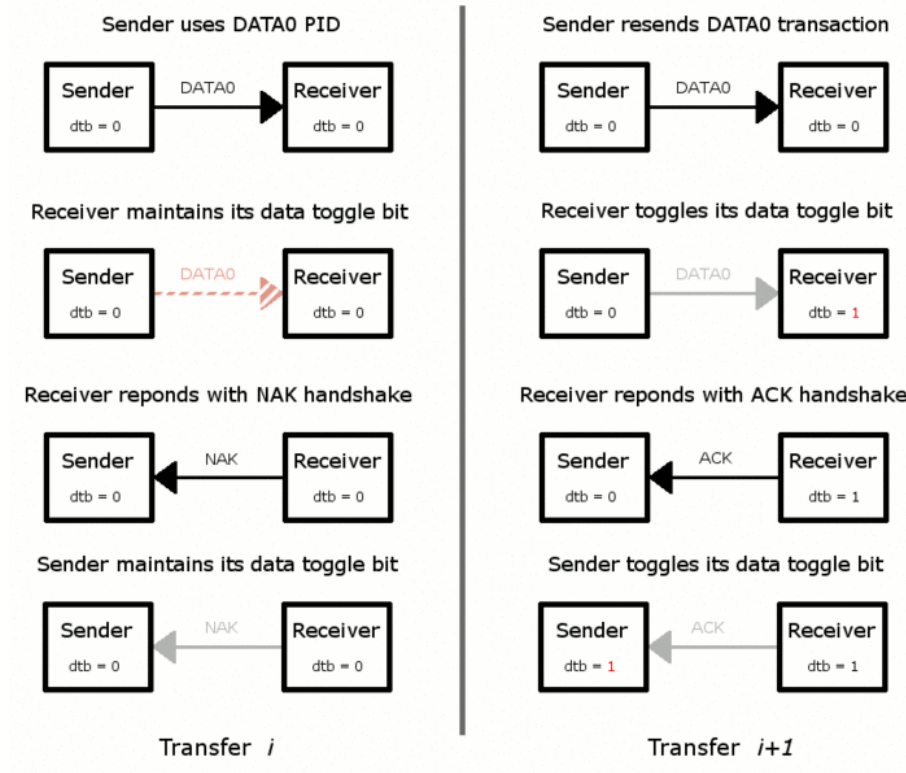


Figure 2.5: Failed Data Transmission [2]

The same principle applies to erroneous handshake packets, where appropriate error handling and toggle bit management enable effective error recovery and resynchronization. [2], [4, sec. 8.6.4]

2.1.4 USB Transfers

Having scrutinized the nuances of packet related concepts, the discussion now redirects attention to the diverse types of transfers inherent in the USB architecture. Through a meticulous examination of each transfer type, the objective is to acquire a comprehensive understanding of their internal mechanisms and operational intricacies.

Control Transfers

Control transfers are characterized by a minimum of two transaction stages: setup and status. It is noteworthy that a control transfer may optionally incorporate a data stage between the setup and status stages. Within the setup stage, information is transmitted to the control endpoint of a function utilizing a setup transaction. These setup transactions bear a resemblance in format to an out transaction, albeit employing a SETUP PID instead of an OUT PID. Notably, a setup transaction consistently employs a DATA0 PID for the data field. [4, sec. 8.5.2], [2]

Proceeding, subsequent to the examination of the structure of the setup data packet, attention shifts to the data stage, where actual data transfer occurs. During data transmission, differentiation is made between sending and receiving packets. When sending

data packets, one or more out transactions are specified, with the token packet toggling between DATA1 and DATA0 for synchronization. Conversely, for receiving packets, one or more in transactions are specified, also toggling the toggle bit per transaction. Notably, the data stage is not mandatory, allowing for 0-length data stages. This is exemplified in scenarios such as setting the device address via the address request2.6, which doesn't require any data transmission as the address is already included in the request. [4, sec. 8.5.2], [2]

The final stage, the status stage, involves the transmission of a zero-length data packet for the purpose of verifying the integrity of the entire transfer process. This packet always contains the toggle bit set to 1, signifying transmission using the DATA1 packet identification. This request serves to confirm the successful completion of the entire transfer, ensuring the absence of transmission errors. [4, sec. 8.5.2], [2]

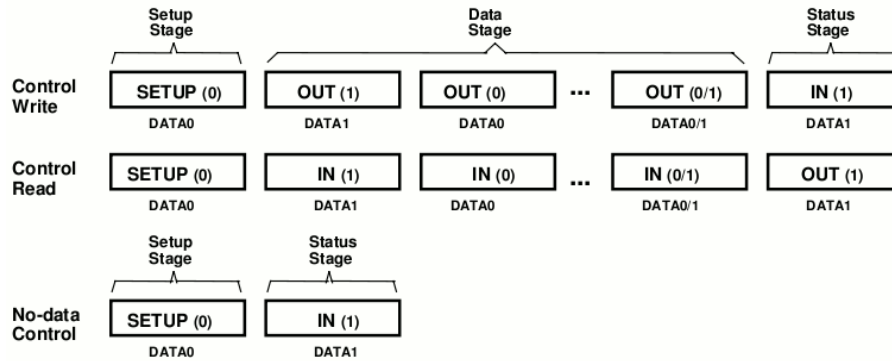


Figure 2.6: Control Transfer [4, sec. 8.5.2]

Interrupt and Bulk Transfer

Following an exhaustive analysis of the intricate operations involved in control transfers, attention is now directed towards the exploration of bulk and interrupt transfers. These two types are examined concurrently owing to their shared underlying mechanisms. Diverging from control transfers, both bulk and interrupt transfers are devoid of both a setup stage and a status stage. Instead, these transfers predominantly comprise multiple data transactions, which collectively constitute the complete transfer process. [4, sec. 8.5.3], [4, sec. 8.5.1]

Nevertheless, akin to control transfers, each data transaction within bulk and interrupt transfers must adhere to a toggle bit protocol. At the outset of a transfer, whether it commences with an IN or OUT transaction, the toggle bit invariably begins with a value of 0. Consequently, the token packet of the initial transaction always identifies a DATA0 packet. Subsequently, the toggle mechanism operates analogously to that described previously. Following the transmission of a DATA0 packet, the toggle bit on the recipient's side toggles to 1, prompting the transmission of a handshake packet back to the sender. Upon error-free reception, the toggle bit on the sender's side transitions to 1. This iterative process ensures the synchronization between sender and receiver, facilitating seamless data transfer. [4, sec. 8.5.3], [4, sec. 8.5.1]

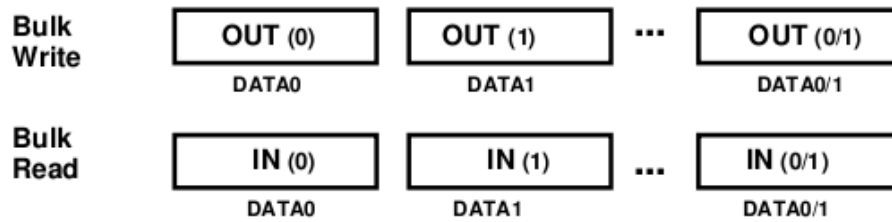


Figure 2.7: Bulk and Interrupt Transfer [4, sec. 8.5.1]

2.2 Universal Serial Bus Device

Following the comprehensive elucidation of each transfer type, it becomes imperative to thoroughly comprehend the operational intricacies inherent in these transfers. Consequently, a transition away from the communication flow is warranted to delve into the internal architecture of USB devices. This exploration necessitates an examination of the structural composition of a USB device, an elucidation of the diverse states a device can assume, and a detailed analysis of the 'descriptors' retrieved from the device's default endpoint.

2.2.1 Architecture

The device architecture comprises three primary components. Firstly, the endpoint serves as the termination point of the device, constituting the conduit through which data flows. Endpoints can assume various types, including interrupt, bulk, control, or isochronous, each delineating distinct data transfer characteristics. Additionally, an endpoint specifies the data direction, which can be categorized into host-to-function and function-to-host directions. This unidirectional flow of data establishes the endpoint as a unidirectional pipe. [2], [5]

In subsequent discussions, the host-to-function endpoint is commonly denoted as the Out-Endpoint, whereas the function-to-host endpoint is termed the In-Endpoint. The interface, the second component, functions as a container aggregating a group of endpoints and manages a specific type of logical connection. Interfaces may be overloaded, allowing for multiple alternate interfaces, each sharing the same endpoints but possessing distinct parameters, such as varied payload sizes to ensure diverse bandwidth allocation. The initial state of an interface is designated by setting 0. [2], [5]

Additionally, a higher level container consolidates all interfaces, facilitating configuration utilization. Although a device can support multiple configurations, only one configuration can be active concurrently. Transitioning between configurations enables the device to access different sets of functionalities, thereby enhancing its versatility and adaptability. [2], [5]

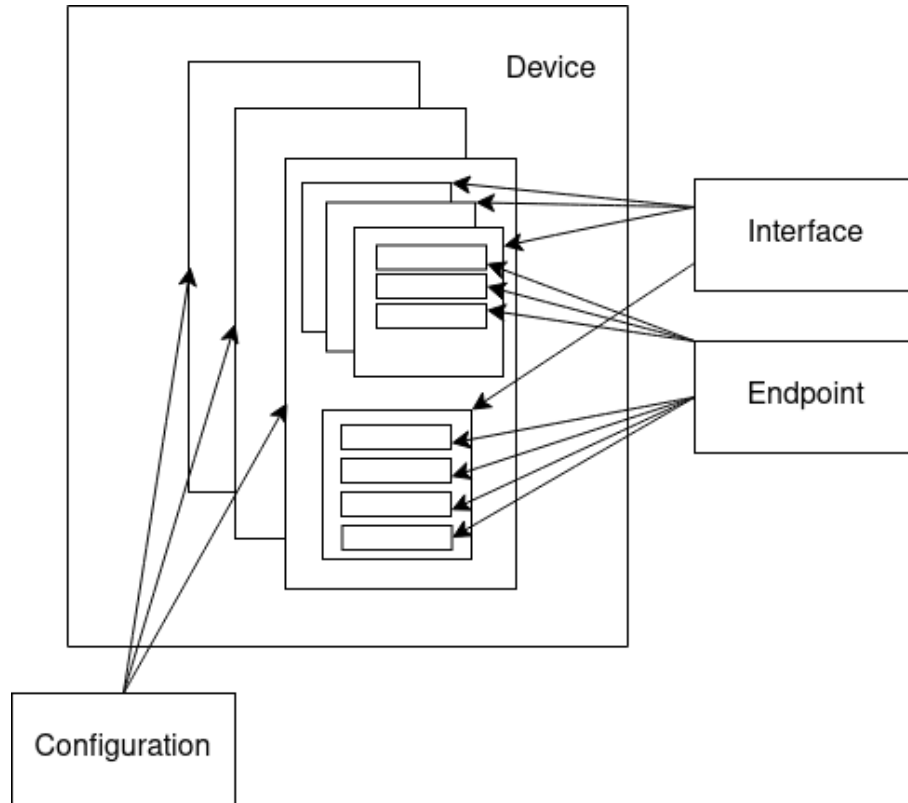


Figure 2.8: USB Device Structure [5]

Understanding the structural intricacies of a USB device necessitates an exploration of its operational states. Initially, upon attachment to a port, the device assumes the attached state, marking the inception of its logical progression. Subsequently, upon powering up, the device transitions to the powered state, signifying its acquisition of power from a viable source. Following power-up and receipt of a port reset, the device enters the default state, thereby enabling its address ability at the default address of 0. [4, sec. 9.1.1.4]

Prior to engaging in device specific operations, a crucial transition occurs into the address state, wherein the device is allocated a persistent address, effective until the power source is deactivated. Acquiring this address and residing in the address state endows the device with access to a spectrum of requests, including the retrieval of the device's descriptor. [4, sec. 9.1.1.4]

Finally, the configured state is attained upon configuring a specified configuration value, delineating the configuration parameters the device should adhere to. Within the configured state, the device gains the capability to transmit data beyond utilizing the default endpoint, thereby facilitating access to each interface and endpoint configured within that particular configuration. [4, sec. 9.1.1.5]

2.2.2 USB Device Request

Subsequent to the exposition on the architecture of the USB device, the 'USB Device Request' structure will be introduced, followed by a meticulous examination of each individual field. These requests invariably manifest within the setup stage of the control

transfer via the default control pipe. In order to generate a particular request, the host must furnish values within the respective fields to instantiate the appropriate request. Further elaboration on these requests will be provided in the section 2.2.3.

Offset	Field	Size	Description
0	bmRequestType	1	[B7, B6, B5, B4, B3, B2, B1, B0] B7 : Data transfer direction <ul style="list-style-type: none"> • 0b : Host-to-device • 1b : Device-to-host B6-5 : Type of request <ul style="list-style-type: none"> • 00b : Standard • 01b : Class • 10b : Vendor B4-0 : Recipient <ul style="list-style-type: none"> • 00000b : Device • 00001b : Interface • 00010b : Endpoint • 00011b : Other
1	bRequest	1	RequestCode
2	wValue	2	RequestParameter
4	wIndex	2	RequestIndex
6	wLength	2	Transfer Len

Table 2.4: USB Device Request [4, sec. 9.3]

At byte offset 0 within the 8-byte data packet, a bitmap is present delineating the transmission characteristics of the data packet. Bit 7 of this bitmap denotes the data transfer direction, where a value of 0 signifies a host-to-device transfer, while a value of 1 indicates a device-to-host transfer direction. Following this, a 2-bit field spanning from Bit 6 to Bit 5 specifies one of four different request types. The standard request type encompasses device specific inquiries, such as retrieving manufacturer details, while the class request type pertains to requests specific to a particular device class (e.g., a mass storage device). Lastly, the vendor specific request type accommodates requests tailored to the device manufacturer. Within the range from Bit 4 to Bit 0, the recipient of the request is identified, signifying the internal component of the device being addressed. For instance,

setting the recipient to 0 designates the device itself, whereas setting it to 2 designates endpoint related information. [4, sec. 9.3.1], [2]

At byte offset 1, a predefined request value is present, encompassing the predefined standard requests that every device is obligated to support. [4, sec. 9.3.2], [2]

bRequest	Value
GET-STATUS	0
CLEAR-FEATURE	1
SET-FEATURE	3
SET-ADDRESS	5
GET-DESCRIPTOR	6
SET-DESCRIPTOR	7
GET-CONFIGURATION	8
SET-CONFIGURATION	9
GET-INTERFACE	10
SET-INTERFACE	11
SYNC-FRAME	12

Table 2.5: Standard Request Codes [4, sec. 9.4]

At byte offset 2, a 2-byte value is situated, serving as a variable parameter [4, sec. 9.3.3], [2]. This parameter's significance and usage will be elucidated further when specific requests are delineated.

Moving to offset 4, another 2-byte value is present, denoted as the index value. This value finds utility in scenarios where the selection of a specific index within an array is necessary, such as when accessing viable configurations for the device. [4, sec. 9.3.4], [2]

Concluding the structure of the 'USB Device Request', a 2-byte value commencing at offset 6 indicates the number of bytes to be transmitted during the data stage of the transaction. Notably, this field is fixed at 2 bytes, as it accommodates the maximum data payload size of 1024 bytes for an endpoint of a device. Given the requirement to store this length, 10 bits are necessary, which when rounded up to the nearest byte, necessitates the allocation of 2 bytes for this purpose. [4, sec. 9.3.5], [2]

2.2.3 Standard Requests

Following this, the inquiry extends to fundamental standard requests essential for this thesis, demanding a comprehensive understanding to facilitate subsequent discourse. It

is imperative that every USB device possesses the capability to respond to these requests and provide appropriate responses.

Set Address

Foremost among these requests is the SET-ADDRESS request, which assumes significance in effecting the transition of a device from its default state to the address state.

bmRequestType	bRequest	wValue	wIndex	wLength
0	5	DEV-ADDR(?)	0	0

Table 2.6: SET-ADDRESS Request [4, sec. 9.4.6]

This request entails the assignment of a specific address to the device, effectively designating the address at which the device will respond to subsequent communications. Notably, this request is exclusively applicable during the default state or address state of the device. Consequently, when the device is already in the address state, invoking this request prompts a transition to the newly assigned address. However, if the device is in the configured state or the specific device address is greater than 127, this request is unsupported, leading to undefined behavior. [4, sec. 9.4.6], [2]

Get Descriptor

The subsequent request of interest is the GET-DESCRIPTOR request, which necessitates a descriptor type value in the wValue field. These descriptor type values are predefined and correspond to specific descriptors to be requested. The allowable descriptor types include:

Descriptor Type	Value
Device	1
Configuration	2
String	3
Interface	4
Endpoint	5
Device-Qualifier	6

Table 2.7: Descriptor Types [4, sec. 9.4]

The actual format of the GET-DESCRIPTOR request looks like this :

bmRequestType	bRequest	wValue	wIndex	wLength
10000000b	6	Descriptor-Info	0 or Lang(?)	Len(?)

Table 2.8: GET-DESCRIPTOR Request [4, sec. 9.4.3]

The bmRequestType and bRequest parameters represent predefined values, yet the interpretation of the other fields necessitates a deeper understanding of their functionality. In the case of wValue, it comprises a high byte and a low byte. The high byte specifies the descriptor type, as enumerated in the aforementioned table, while the low byte denotes a descriptor index, indicating the selection of a specific descriptor from an array. [4, sec. 9.4.3], [2]

Moving on to the wIndex field, its value is uniformly set to 0 since the request does not pertain to a string descriptor, necessitating the default value of 0. Finally, the wLength field specifies the length of data to be transmitted or received. [4, sec. 9.4.3], [2]

Notably, this request is applicable in all states of the device, ensuring its versatility across different operational phases. [4, sec. 9.4.3]

Get Configuration

Next in line is the retrieval of a specific configuration of a device through the GET-CONFIGURATION request, the structure of which is as follows:

bmRequestType	bRequest	wValue	wIndex	wLength
10000000b	8	0	0	1

Table 2.9: GET-CONFIGURATION Request [4, sec. 9.4.2]

The purpose of the GET-CONFIGURATION request is to retrieve the active configuration number of the device. It is important to note that this request is only supported and defined when the device is in the configured state. In the address state, this request returns a value of 0, indicating that the device is not yet in the configured state. This functionality provides insight into the current operational configuration of the device, aiding in the management and control of USB device configurations. [4, sec. 9.4.2], [2]

Set Configuration

Consequently, the existence of the GET-CONFIGURATION request implies the presence of a complementary request for setting a configuration on the device. This request, known as SET-CONFIGURATION, enables the selection and activation of a specific configuration that the device should utilize.

bmRequestType	bRequest	wValue	wIndex	wLength
0	9	Configuration-Info	0	0

Table 2.10: SET-CONFIGURATION Request [4, sec. 9.4.7]

The SET-CONFIGURATION request contains crucial configuration information in its wValue field, specifying the desired configuration value for the device. Given that a USB device can support a maximum of up to 255 configurations, only 1 byte of the 2-byte-sized field is necessary for this purpose, with the remaining bits reserved. This request facilitates the transition of a device from the address state to the configured state, allowing the device to determine which configuration to utilize for its operations. Understanding these requests is fundamental, as they play a pivotal role in device initialization and configuration within the USB architecture. [4, sec. 9.4.7], [2]

These requests represent essential knowledge for further exploration. However, for those interested in delving deeper into the standard USB requests, additional information can be found in related resources. [4, sec. 9.4]

2.2.4 Standard Descriptors

Having elucidated the structural composition of the device, attention now turns to the section dedicated to descriptors. In the domain of USB architecture, a descriptor constitutes a standardized data structure from which device specific details can be extracted. Each descriptor commences with a uniform sequence of two bytes. The initial byte denotes the size of the descriptor in bytes, followed by a byte indicating the descriptor type, as enumerated in table 2.7. Additionally, descriptors may incorporate additional byte fields, such as an index field referencing a specific string descriptor that denotes a Unicode string, a facet to be explored in greater detail later on. [4, sec. 9.5], [2]

Device Descriptor

Presently, attention is directed towards the quintessential standard USB descriptor: the Device Descriptor. Acquired through the transmission of the request delineated in section 2.8, this descriptor encompasses a total of 18 bytes and provides fundamental information pertaining to the device itself.

Offset	Field	Size
0	bLength	1
1	bDescriptorType	1
2	bcdUSB	2
4	bDeviceClass	1
5	bDeviceSubClass	1
6	bDeviceProtocol	1
7	bMaxPacketSize0	1
8	idVendor	2
10	idProduct	2
12	bcdDevice	2
14	iManufacturer	1
15	iProduct	1
16	iSerialNumber	1
17	bNumConfigurations	1

Table 2.11: Device Descriptor [4, sec. 9.6.1]

Initiating at offset 0x02 within the USB device descriptor, the bcdUSB field emerges as a pivotal component, spanning a word in size. This field delineates the binary coded decimal (BCD) representation of the USB specification release number designated for a particular device. For instance, a device conforming to USB 1.0 would manifest as 0x0100, while 0x0101 denotes USB 1.1 compatibility. [4, sec. 9.6.1], [2]

Following this, the descriptor features the three bDeviceX fields, each occupying one byte. These fields harbor fixed values that bestow unique identifiers, crucial for discerning the device type. [4, sec. 9.6.1], [2]

Positioned at offset 0x07, the bMaxPacketSize0 field, also one byte in length, assumes significance by prescribing the maximum payload permissible for endpoint 0. [4, sec. 9.6.1], [2]

Following suit are two uniquely assigned fields designating the vendor and product of the device, aggregating to a total of four bytes. [4, sec. 9.6.1], [2]

Subsequent to this, a BCD field reappears, housing the device's release number in binary-coded format, spanning a two-byte expanse. [4, sec. 9.6.1], [2]

Following this, three byte fields serve as indices to determine the appropriate string descriptor when retrieving specified information. These fields are discretionary and may

remain unset, defaulting the indexes to 0. [4, sec. 9.6.1], [2]

Finally, culminating at offset 0x30, the bNumConfigurations field assumes presence, tasked with enumerating the configurations supported by the device. [4, sec. 9.6.1], [2]

Configuration Descriptor

Following the examination of the device descriptor, the subsequent crucial component in the hierarchy is the configuration descriptor, spanning a modest 9 bytes. Retrieving this descriptor involves transmitting a specific request, detailed in 2.8, thereby requesting a particular configuration from the device. These configuration values can range from 0 to bNumConfigurations-1, as specified in the device descriptor. Notably, when requesting the configuration descriptor, an important observation arises: along with the configuration descriptor, all interface descriptors are transmitted, and in tandem, all associated endpoint descriptors are also conveyed. Thus, the retrieval process entails the transmission of the configuration descriptor initially, followed by the transmission of the first interface descriptor within that configuration. Subsequently, all endpoint descriptors pertaining to that interface are transmitted. This sequential transmission process persists for all interfaces within the configuration and all endpoints corresponding to each interface. Given this protocol, the focus now shifts towards understanding the structure and composition of interface and endpoint descriptors, which are exclusively accessible when requesting the configuration descriptor. [4, sec. 9.6.2], [2]

Offset	Field	Size
0	bLength	1
1	bDescriptorType	1
2	wTotalLength	2
4	bNumInterfaces	1
5	bConfigurationValue	1
6	iConfiguration	1
7	bmAttributes	1
8	bMaxPower	1

Table 2.12: Configuration Descriptor [4, sec. 9.6.2]

Commencing with the configuration descriptor, located at offset 0x02, the first byte indicates the total length in bytes, encompassing the combined lengths of all descriptors (Configuration, Interface, and Endpoint). [4, sec. 9.6.2], [2]

Subsequently, at offset 0x04, the bNumInterface field specifies the number of interfaces supported by this configuration. [4, sec. 9.6.2], [2]

Proceeding to offset 0x05, the `bConfigurationValue` delineates the configuration value to which this configuration is assigned. [4, sec. 9.6.2], [2]

Following this, the `iConfiguration` field, situated after the offset, designates the index of a string descriptor containing a Unicode string describing this configuration. Notably, this field is optional, and when set to 0, no string descriptor is associated with it. [4, sec. 9.6.2], [2]

The subsequent fields encompass hardware capabilities of the device, which, for the current purpose, are non essential.

Interface Descriptor

Following this, the interface descriptor is encountered, forming a component of the configuration descriptor request and spanning a total of 9 bytes. It is essential to note that the interface and endpoint descriptors are not directly accessible; rather, they can only be accessed when requesting the entire configuration, as detailed previously. The format of the interface descriptor is as follows:

Offset	Field	Size
0	<code>bLength</code>	1
1	<code>bDescriptorType</code>	1
2	<code>bInterfaceNumber</code>	1
3	<code>bAlternateSetting</code>	1
4	<code>bNumEndpoints</code>	1
5	<code>bInterfaceClass</code>	1
6	<code>bInterfaceSubClass</code>	1
7	<code>bInterfaceProtocol</code>	1
8	<code>iInterface</code>	1

Table 2.13: Interface Descriptor [4, sec. 9.6.3]

At offset 0x02 of the interface descriptor, the interface number corresponding to that specific interface is encountered. [4, sec. 9.6.3], [2]

Subsequently, at offset 0x03, the `bAlternateSetting` field is located, which specifies the current alternate setting for this interface. This field indicates which setting the interface currently adopts, a particularly pertinent aspect in scenarios where multiple alternate settings exist for an interface. [4, sec. 9.6.3], [2]

Continuing, at offset 0x04, the `bNumEndpoints` field is found, denoting the number of endpoints supported by this interface. It is noteworthy that the default endpoint (`endpoint0`)

is excluded from this enumeration. [4, sec. 9.6.3], [2]

Subsequent to this, constant numbers associated with the specific interface are present from offset 0x05 to offset 0x07, each unique to the interface in use. [4, sec. 9.6.3], [2]

Finally, at offset 0x08, an index is located, pointing to a string descriptor describing this interface. However, it is essential to acknowledge that this index is optional and remains set to 0 if no descriptor is associated with the interface. [4, sec. 9.6.3], [2]

Endpoint Descriptor

Subsequent to the interface descriptor, the endpoint descriptor is encountered, consisting of 7 bytes, structured as follows:

Offset	Field	Size	Description
0	bLength	1	7
1	bDescriptorType	1	5
2	bEndpointAddress	1	[B7, B6, B5, B4, B3, B2, B1, B0] B7 : Direction <ul style="list-style-type: none"> • 0 = OUT endpoint • 1 = IN endpoint B6:4 : Reserved B3:0 : Endpoint number
3	bmAttributes	1	[B7, B6, B5, B4, B3, B2, B1, B0] B7:6 : Reserved B5:4 : Only for Iso support B3:2 : Only for Iso support Transfer Type <ul style="list-style-type: none"> • 0b00 = Control • 0b01 = Iso • 0b10 = Bulk • 0b11 = Interrupt
4	wMaxPacketSize	2	B11:0 specify the maximum packet size B12:11 only for high-speed devices B15:13 reserved
5	bInterval	1	Full- and Low-speed interrupt : <ul style="list-style-type: none"> • range from 1 to 255 High-speed interrupt : <ul style="list-style-type: none"> • range from 1 to 16

Table 2.14: Endpoint Descriptor [4, sec. 9.6.4]

The initiation of the endpoint descriptor is demarcated by the determination of direction, enclosed within bit 7 at offset 0x02. When this bit is asserted high, it denotes an IN endpoint; conversely, its low state signifies an OUT endpoint. While bits 6 to 4 remain reserved, the lower four bits are designated for the endpoint number, accommodating a maximum of 15 endpoints (inclusive of endpoint 0). [4, sec. 9.6.4], [2]

Following this, at offset 0x03, spanning a single byte, lies the bmAttributes field. Notably, only the lower two bits of this field are pertinent to the discussion, as the remaining bits are relevant solely to isochronous endpoints, which fall outside the scope of support. The provided listing delineates the distinctive bits associated with control, bulk, and interrupt endpoints. [4, sec. 9.6.4], [2]

Subsequently, at offset 0x04, another one-byte-long field is encountered. Here, the primary significance pertains to the first 12 bits, which specify the maximum payload allowable for the given endpoint. [4, sec. 9.6.4], [2]

Finally, at offset 0x05, the last field emerges, representing the interval for polling of interrupt endpoints. Given the exclusive support for full and low speed devices within USB 1.x, the values within this field range from 1 to 255 milliseconds, as prescribed. [4, sec. 9.6.4], [2]

String Descriptor

Shifting focus to the final descriptor in the sequence, the string descriptor presents itself as an optional feature for devices. Diverging from its predecessors, this descriptor has the flexibility to assume one of two distinct formats. The initial format is activated when the index provided in the request is set to zero. In such cases, the returned data consists of an array of language IDs, without null termination. Consequently, each entry within this array denotes a supported language ID, which can subsequently be referenced when retrieving a string descriptor other than zero. [4, sec. 9.6.5], [2]

Offset	Field	Size	Description
0	bLength	1	$N + 2$
1	bDescriptorType	1	3
2	wLangID[0]	2	Lang id 0
...
N	wLangID[x]	2	Lang id x

Table 2.15: String Descriptor v1 [4, sec. 9.6.5]

Proceeding to the alternative format, the string descriptor accommodates a Unicode string. Extracting this string necessitates the provision of a language ID supported by the device, thereby instructing the device on the preferred language for the returned string. Similar to the former format, the data retrieved adheres to a non null terminated structure. [4, sec. 9.6.5], [2] The arrangement of this descriptor is as follows:

Offset	Field	Size
0	bLength	1
1	bDescriptorType	1
2	bString	N

Table 2.16: String Descriptor v2 [4, sec. 9.6.5]

String Descriptor Requests

To expound upon the structure of these requests, their specifications will be delineated. The initial request is directed towards retrieving the array of language IDs from the device. Meanwhile, the subsequent request is geared towards acquiring a valid descriptor at index x , thereby accessing the information designated by the device for that specific index.

The formulation of the request for acquiring the array of language IDs is as follows:

bmRequestType	bRequest	wValue	wIndex	wLength
10000000b	6	(3 \ll 8 and 0)	0	N

Table 2.17: GET-DESCRIPTOR(String v1) Request

In the subsequent fields, the wValue parameter denotes the type of the string descriptor (3), which is shifted eight bits to the left, constituting the high bits of the wValue field. In the lower bits, the descriptor index, denoted as 0 in this case, is stored, indicating the request for language identifiers. The length field is specified as N bytes long, initially retrieved in a preceding request which yielded only 2 bytes. Within these 2 bytes, as previously mentioned, both the length and descriptor type fields are encapsulated. Upon acquiring the actual size required by reading the length field, the wLength field is set to encompass the entirety of the descriptor. For subsequent requests, assuming a lang id of 0x0409 signifying english is returned, and the device furnishes a string descriptor indexed at 1. [2], [6]

The formulation for the second request is as follows:

bmRequestType	bRequest	wValue	wIndex	wLength
10000000b	6	(3 « 8 and 1)	0x0409	N

Table 2.18: GET-DESCRIPTOR(String v2) Request

The primary distinction in this request lies in the need to insert the descriptor index into the lower 8 bits of the wValue parameter, thereby specifying which string descriptor of the device is to be read. As elucidated earlier, these indices can be discerned from other descriptors such as the device descriptor. Assuming a device descriptor index of 1, it is imperative to designate this index within the lower 8 bits of the wValue parameter. Furthermore, the selection of the language id to be employed for the returned string descriptor must also be specified. [2], [6]

Having acquired a comprehensive understanding of the structural components and operational states of USB devices, familiarity with terms such as configuration, interface, and endpoint is essential for subsequent discussions. These foundational components serve as crucial building blocks for comprehending later discourse on USB implementation.

2.3 Universal Host Controller Interface

However, before delving into the intricacies of USB system implementation, it is imperative to direct attention towards the final component, namely the actual controller. This controller serves as the medium for communication with USB devices and serves as the interface for initiating transfers by the host. Here, emphasis will be placed on the UHCI. It is assumed that readers possess a foundational understanding of PCI architecture, as the UHCI's role in USB communication will be explored without exhaustive detail. For readers in need of additional background on PCI, supplementary resources are recommended [7].

2.3.1 PCI Enumeration

The commencement of communication with the UHCI involves accessing a segment of the I/O space, facilitated by information obtained from the device's configuration space. Identification of the UHCI on the PCI Bus entails evaluating three unique values: classcode (0x0c), subclasscode (0x03), and programming interface (0x00), confirming the presence of a UHCI device. [8]

In initializing the PCI configuration space for the UHCI, specific configuration writes are executed. These writes involve setting the Command Register (offset 0x04) to enable I/O space response and bus mastering, while also disabling memory mapped I/O and enabling interrupts. Subsequently, additional writes are conducted: writing 0x00 to 0x38

along with the capabilities pointer offset, and disabling legacy support by writing 0x8F00 to 0xC0. [8]

Following the configuration writes, it becomes necessary to perform reads from the configuration space to obtain essential information for the UHCI. A read of Base Address 4 is conducted to ascertain the location of the I/O space, while another read is executed to determine the interrupt line for the UHCI. [8]

Upon completion of requisite writes and reads, communication with the UHCI via the I/O space is established. The I/O space allocated for UHCI operation encompasses eight registers, which warrant thorough investigation in subsequent analyses.

An essential consideration preceding the analyses is the presumption that the UHCI configuration consists of only two root ports. This assumption holds significance, as the investigation is exclusively centered on accessing the initial two root ports, without engaging subsequent root ports throughout the entirety of the procedures. Therefore, in scenarios where a UHCI configuration encompasses more than two root ports, it becomes imperative to assign devices to the relevant ports accurately, as failure to do so could lead to non functional operation.

2.3.2 I/O Space

The ensuing table presents the I/O space allocated to the UHCI, offering a comprehensive overview of each register and its corresponding functionality. Through a methodical analysis, each register will be thoroughly scrutinized, elucidating its intended purpose and operational significance within the UHCI architecture.

Offset	Name	Bits
0x00	Command	16
0x02	Status	16
0x04	Interrupt	16
0x06	Frame Number	16
0x08	Frame Base Address	32
0x0C	Start of Frame	8
0x10	Port Control 1	16
0x12	Port Control 2	16

Table 2.19: I/O Registers [9]

Command Register

Located at memory offset 0x00, the Command Register assumes a paramount role in governing UHCI. This register, comprising a word (2 bytes), features several significant bits—bit0, bit1, bit2, bit6, and bit7—each serving distinct control functions. [9], [10]

Bit	Name
7	MAXP
6	CF
5	SWDBG
4	FGR
3	EGSM
2	GRESET
1	HCRESET
0	RS

Table 2.20: Command Register [9]

Bit0, designated as the Run/Stop bit, dictates the operational state of the UHCI; toggling it to a low state halts all ongoing transfers, while setting it to a high state resumes transfer processing. Bit1, known as the Host Controller Reset bit, initiates a reset of the UHCI to its default configuration when set high. Notably, this bit automatically reverts to 0 after a predefined timeout period following its activation. Bit3, referred to as the Global Reset bit, triggers a reset across all devices connected to the UHCI's ports. Unlike bit1, this bit maintains its state after a timeout and requires manual reversion. Bit6 serves as a configurable flag indicating the UHCI's configuration status, while bit7 determines whether a 32 or 64-byte packet should be sent at the conclusion of each frame. [9], [10]

Status Register

Proceeding to offset 0x02, the Status Register provides insights into the operational status of the UHCI. Unlike other registers, this register operates on a write-clear mechanism, whereby writing to a set bit resets it to zero. [9], [10]

Bit	Name
5	HC Halted
4	HC Process Error
3	Host System Error
2	Resume Detect
1	USB Error
0	USBINT

Table 2.21: Status Register [9]

Of primary interest are the first two bits: Bit0 indicates the completion of a transfer and the occurrence of an interrupt, while bit1 denotes errors encountered during data transfer. [9], [10]

Interrupt Register

At offset 0x04, another register—a word in size—facilitates the selection of conditions prompting interrupt signals.

Bit	Name
3	SPI
2	IOC
1	RI
0	TO/CRC

Table 2.22: Interrupt Register [9]

Bits 0 and 2, in particular, play crucial roles in this regard: setting bit0 high triggers an interrupt in response to timeout or CRC errors, while activating bit2 prompts an interrupt when the Interrupt On Complete (IOC) flag is set during transfers. [9], [10]

Frame Number Register

Following the Command and Status Registers, the Frame Number Register assumes significance at memory offset 0x06, spanning 2 bytes in size.

Bit	Name
10:0	Frame-Number

Table 2.23: Frame Number Register [9]

This register encapsulates the current frame index, utilizing only the lower 10 bits to accommodate the 1024 frame entries. [9], [10]

Frame Base Address

Subsequently, the Frame Base Address Register, situated at offset 0x08, spans 4 bytes and is pivotal in specifying the physical address of the frame list in memory.

Bit	Name
31:12	Base-Address
11:0	Reserved

Table 2.24: Frame Base Address Register [9]

Bits 31 to 12 denote the base address, while the lower 12 bits serve as an offset to this address. It is imperative that the base address be aligned to a 4KB boundary to enable access to all 1024 32-bit addresses in memory. [9], [10]

Start of Frame Register

Proceeding, the Start Of Frame Register, located at offset 0x0c, governs the timing of each frame, with writes to this register incrementing the frame time by 11936 bit intervals. For simplicity, a value of 64 is typically set, ensuring a consistent frame duration of 1ms, equivalent to 12000 bit times. Consequently, the UHCI executes each frame within a precise 1ms time frame. [9], [10]

Bit	Name
7	Reserved
6:0	Modifier

Table 2.25: Start of Frame Register [9]

Port Register

Lastly, the Root Port Registers 1 and 2, positioned at offsets 0x10 and 0x12, respectively, merit attention. Each register spans 2 bytes and facilitates the management of connected devices.

Bit	Name
12	Suspend
9	Port Reset
8	Low Speed
7	Reserved
6	Resume Detect
5:4	Line Status
3	Port Change
2	Port Status
1	Connect Change
0	Connect Status

Table 2.26: Port Register [9]

Bit0 of these registers verifies device attachment to the port, while bit1 indicates port status changes. Bit2 signals port activity when set to 1, necessitating activation for data transmission. Additionally, bit8 designates low speed device connectivity, with a value of 1 indicating such a connection. Bit9 serves as a port reset mechanism, while the remaining bits hold lesser relevance to specified use cases. Together, these registers constitute essential components for configuring the UHCI appropriately. [9], [10]

2.3.3 Internals

In this section, the examination will focus on elucidating the internal functionality entailing operations such as resetting, configuring, resetting/enabling downstream ports, and the workings of the interrupt routine for the UHCI.

Reset

The reset process of the UHCI commences with the activation of the GRESET bit, initiating a reset signal to all devices connected to the downstream ports of the UHCI. Notably, this bit remains unset automatically by the UHCI, necessitating manual clearance. The prescribed delay for clearing this bit is stipulated to be at least 10ms. Subsequently, following this delay period, a value of 0 is written to the command register, resulting in the cessation of UHCI operations and the resetting of the GRESET bit. Upon execution of this write operation, the UHCI should transition to a halted state, indicated by the setting of the HC Halted bit (bit 5) in the status register. Once the halted state of the UHCI is confirmed, the status register can be fully cleared. This is achieved by writing the value 0x00FF to the register, as it operates on a write-clear basis. The final step requisite for ensuring the correct reset of the UHCI involves the execution of hardware resets, which serves to reset the internal state of the UHCI itself. This action is effectuated by setting the HCRESET bit (bit 1) in the command register. [10]

Configuration

Upon appropriately resetting the UHCI, a delay of 10ms should ensue to allow the UHCI sufficient time before proceeding with the configuration process. [10]

Before the UHCI can initiate traversal of the schedule, a prescribed sequence of configuration steps is mandated. Firstly, the frame list base address must be set to indicate the physical address of the frame list, which must be aligned to 4KB in memory. This is achieved by writing the base address into the frame base address register. Subsequently, the configuration procedure necessitates setting the start index to denote the commencement frame. This index should be set to 0 to commence from frame 0, accomplished by writing a value of 0 to the frame number register. To prompt the UHCI to execute each frame precisely for 1ms, an additional write operation to the start of frame register is mandated. Writing a value of 0x40 into this register instructs the UHCI to execute each frame precisely for 1ms. Following this step, another write operation to the interrupt register is requisite to notify the UHCI to execute interrupts accordingly. In this register, the TO/CRC bit and the IOC bit are written. Subsequent to directing the UHCI to execute specific interrupts, the subsequent write operation involves writing 0xFFFF into the status register to clear all status bits. [10]

The final step entails returning the UHCI to operational status, achieved by writing the RS bit (bit 0) into the command register. [10]

Port Reset and Enable

The reset procedure for the UHCI downstream port initiates with the activation of the reset bit (bit 9) within the port register corresponding to port i . Following the setting of the reset bit, manual intervention is necessitated to unset it. The prescribed delay for the reset process is stipulated to be at least 50ms. Subsequent to the 50ms delay, the reset bit must be cleared, ensuring that the port change (bit 3), connect change (bit 1), and port status (bit 2) remain unset. To accomplish this, a write operation of 0xFCB1 to the register associated with port i is mandated. It is imperative to observe that the connect change bit (bit 1) should not be cleared simultaneously with the reset operation. Following this write operation, a further delay of 300 μ s is indicated before proceeding. [10]

Upon expiration of the delay, the connect change (bit 1) must be cleared before enabling the port. This clearance is achieved by writing a value of 1 to the connect change bit (bit 1). Subsequently, the port is enabled by setting the port status bit (bit 2). A total duration of 50 μ s is allotted to allow the port to enable itself. Finally, the connect change (bit 1) and port change (bit 3) are to be cleared if they are still set, while ensuring that the connect status (bit 0) and the port status (bit 2) remain set. This is executed by writing a value of 0xF to the specific port register. Upon connection of a device to the downstream port, the connect status (bit 0) will be set accordingly, signifying readiness to commence device configuration. [10]

Interrupt Handling

The process of handling interrupts commences with the activation of the interrupt bits in the interrupt register, indicating to the UHCI that interrupts should be sent if the specified criteria are met. Typically, the bits set for this purpose include the TO/CRC bit (bit 0) and the IOC bit (bit 2).

Subsequently, after informing the UHCI regarding when to trigger an interrupt, if the UHCI is successfully configured and running when the criteria, such as a successful transaction with the IOC bit set, are met, the USBINT bit (bit 0) in the Status Register is set by the UHCI. This signals the presence of an interrupt that necessitates handling. Similarly, in the event of an erroneous transfer, the UHCI sets the USB Error bit (bit 1) in the Status Register, indicating the occurrence of an error that requires attention. These interrupts persist until these bits are appropriately cleared. To clear these bits, a write operation is performed to clear them, signifying that writing to them will clear their respective states.

2.3.4 Stack

Frame List

Given the pivotal significance of the frame list, whose base address must be specified in the I/O space, it is imperative to delve into its construction. As previously mentioned, the frame list comprises 1024 consecutive entries in memory, each spanning 4 bytes, resulting in a total memory requirement of 4KB. To ensure proper functionality, this memory must

be aligned to a 4KB boundary, as the high part of the frame base address remains fixed, with only the offset (low part) varying to facilitate access to each entry. [9], [11]

An individual frame entry structure is delineated as follows:

Bits	Description
31:4	Frame List Pointer
1	QH/TD Select
0	Terminate

Table 2.27: Frame Entry [9]

The initial bit denotes a terminate bit, indicating whether the UHCI should proceed to follow the subsequent link. If this bit is set high, the UHCI will cease following the link. Subsequently, bit 1 specifies whether the subsequent structure is a Queue Head (QH) or a Transfer Descriptor (TD). A value of 1 in this field indicates that the pointer points to a QH, whereas any other value indicates a TD. Finally, the frame list pointer spans from bit 31 to bit 4, directing to the physical address of the QH or TD, as specified by the QH/TD Select. Upon following this physical address, the lower 4 bits are zeroed out by the UHCI, necessitating that both the QH and TD be 16-byte aligned in memory to ensure proper functionality. [9], [11]

To gain a better view of the whole frame list and its frames, the following illustration is presented :

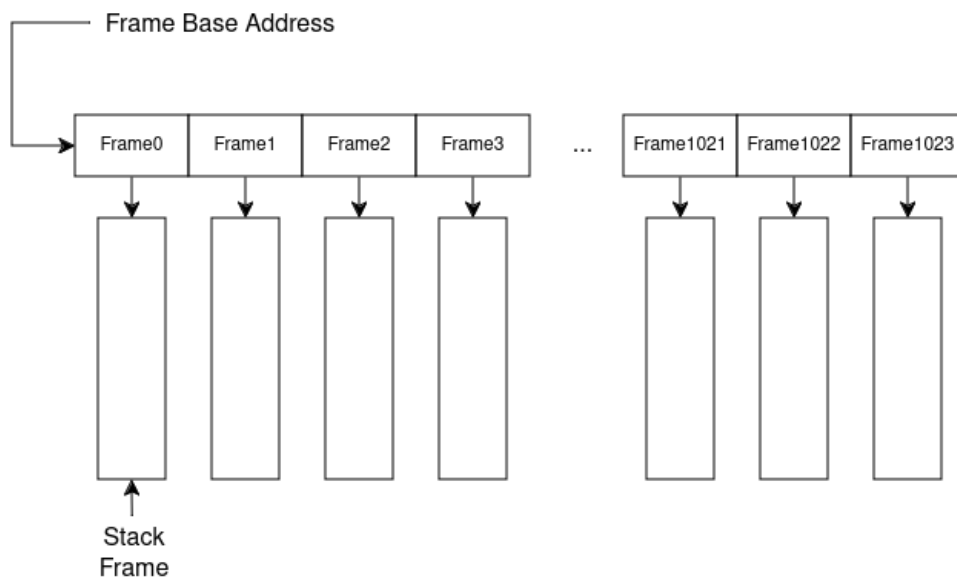


Figure 2.9: UHCI Frame List Structure [11]

Within this context, the Frame Base Address designates the initiation point of the frame list, comprising 1024 frames commencing from index 0. Each frame encompasses its

individual stack frame containing the transfer intended for execution by the UHCI upon entering that particular frame. [11]

To elucidate the traversal mechanism employed by the UHCI across these frames, the ensuing illustration depicts how the UHCI updates the frame number index, which is stored in the frame number register of the UHCI, to indicate the respective frame.

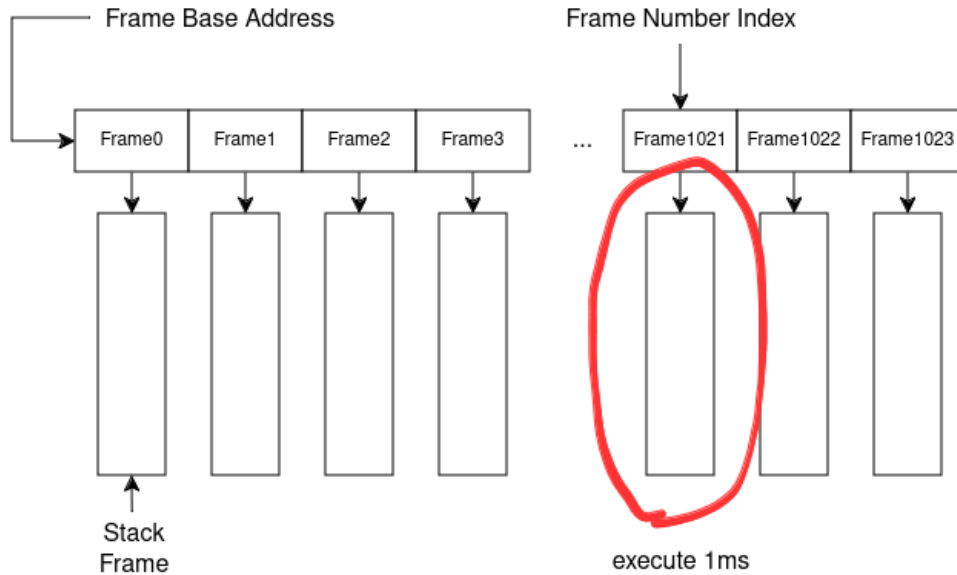


Figure 2.10: UHCI Frame Handling

It is pertinent to note that conventionally, each frame is executed precisely for 1ms. Consequently, the UHCI executes the stack frame corresponding to the designated frame until the elapse of the designated time period. For instance, in this scenario, the UHCI would sequentially process all transfers associated with frame 1021 for a duration of 1ms. Subsequently, after updating the frame number index in the frame number register, the UHCI proceeds to the subsequent frame. To reiterate the execution of this specific frame by the UHCI, it must await precisely 1024ms, attributable to the necessity of processing 1023 other frames before resuming the execution cycle at this frame. [10]

Queue

Considering that each entry points to either a QH or a TD, it becomes imperative to dissect the structure of these entities. Primarily, the QH will be scrutinized, comprising 16 bytes and required to adhere to a 16-byte alignment requirement, as previously elucidated.

Offset	Description
0x00	QHLP
0x04	QELP
0x08	free
0x0C	free

Table 2.28: Queue Head [9]

The QH comprises a Queue Head Link Pointer (QHLP) located at offset 0x00, which serves to link the current QH to the subsequent QH or TD in the sequence. [9], [11] The structure of the QHLP is defined as follows:

Bits	Description
31:4	QHLP
1	QH/TD Select
0	Terminate

Table 2.29: Queue Head Link Pointer [9]

Bit 0 of the QHLP denotes whether the current QH is the final one, thereby signaling the UHCI that no additional QHs follow. A setting of 0 indicates that the UHCI should continue to follow the link to the next QH. Bit 1 functions similarly to previous instances, designating whether the pointer should be treated as a QH. Once again, bits 31 to 4 represent the physical address of the next QH in the sequence. It is imperative to ensure that this memory block adheres to a 16-byte alignment requirement. [9], [11]

Advancing through the QH structure, the Queue Element Link Pointer (QELP) at offset 0x04 is encountered, directing to the first TD associated with the respective QH. [9], [11] The structure of QELP is delineated as follows:

Bits	Description
31:4	QELP
1	QH/TD Select
0	Terminate

Table 2.30: Queue Element Link Pointer [9]

Bits 0 and 1 maintain analogous functionality to the aforementioned bits, serving to indicate the pointer type. However, bits 31 to 4 now signify the subsequent TD or QH in the sequence. Notably, upon completing the processing of the first TD, the QELP within the QH is updated with the address of the subsequent TD. This mechanism is of significance, as it ensures that the QELP in the QH consistently references the next TD to be processed. This preservation of TDs becomes essential should the UHCI fail to handle a frame entry within the specified timeframe. [9], [11]

The subsequent two double words, commencing from offset 0x08, are not reserved and can be allocated for custom applications. In this context, the first double word at offset 0x08 is utilized to contain a physical address pointing to the parent QH, while the subsequent double word at offset 0x0c is employed to denote various flags indicating crucial aspects regarding the QH. [9], [11]

Transfer Descriptor

Analogous to QHs, TDs must also adhere to a 16-byte alignment requirement. These components serve as the linchpin for actual data transfers, with each TD akin to a single transaction. The aggregation of TDs linked to a singular QH constitutes the entirety of a transfer.

Offset	Description
0x00	Link Pointer
0x04	Control and Status
0x08	Token
0x0C	Buffer Pointer

Table 2.31: Transfer Descriptor [9]

At offset 0x00 of the TD, the Link Pointer is encountered, comprising a double word. The structure of this Link Pointer is outlined as follows:

Bits	Description
31:4	Link Pointer
2	Depth/Breadth Select
1	QH/TD Select
0	Terminate

Table 2.32: Link Pointer [9]

Bit 0 serves as an indicator of whether the TD represents the final one in the sequence. When unset, it signifies the presence of a subsequent TD linked to the current one. Similarly, Bit 1 denotes the type of structure pertaining to the subsequent element: either a QH or TD. Positioned at bit 2, the Depth/Breadth Select bit assumes significance. Activation of this bit prompts the UHCI to operate in depth, thereby traversing all TDs linked to the current QH before advancing to the subsequent QH. Conversely, deactivation of this bit results in the UHCI processing one TD at a time, moving to the next QH only after completing all TDs associated with the current one. For the specified purposes, the former approach is exclusively employed, ensuring thorough traversal of all TDs associated with a single QH before progression to subsequent operations or error handling. [9], [11]

The concluding field spans from bit 31 to 4, housing the physical address of the subsequent TD. Upon following this link, the UHCI treats the lower bits as 0. Proceeding to the subsequent double word at offset 0x04, the control and status field is encountered, a pivotal component housing status bits indicative of the transfer's outcome and control bits instrumental in governing the transfer process. [9], [11]

Bits	Description
29	SPD
28/27	C-ERR
26	LS
25	IOS
24	IOC
23	ACTIVE
22	STALLE
21	DATA-BUFFER-ERROR
20	BABBLE-DETECTED
19	NAK
18	CRC/TIMEOUT-ERROR
17	BITSTUFF-ERROR
10:0	ACTUAL-LEN

Table 2.33: Control and Status [9]

Among the control bits, bit 23 assumes prominence, serving as an activator for the transfer when set. Upon successful transaction completion, this bit reverts to 0. Concurrently, bit 24 orchestrates the generation of an interrupt by the UHCI upon transaction conclusion, with the interrupt dispatched at the culmination of the frame. It's noteworthy that if multiple QHs coexist within a frame, and one QH concludes its processing with the

IOC bit set, the UHCI continues handling other QHs within the frame until the frame's designated time elapses. Subsequently, the interrupt is triggered and must be handled by the appropriate driver. [9], [11]

Furthermore, bit 26 governs the execution of a low-speed transaction, necessitating activation if the attached device operates at a low-speed. Bits 27 and 28 denote the error count, delineating the maximum permissible count of error transmissions accepted by the UHCI. A value of 3 denotes the threshold, beyond which an error-ridden TD is deemed invalid, prompting cessation of processing for the associated QH and transition to the subsequent QH. [9], [11]

The remaining bits, excluding the IOS bit, are allocated for conveying the transfer's status. In cases of erroneous transfers, these status bits assume relevant configurations, reflecting the encountered errors. [9], [11] Transitioning to the subsequent double word at offset 0x08, the token segment is encountered, a critical component delineating various aspects of the transaction.

Bits	Description
31:21	Maximum Length
19	Data Toggle
18:15	Endpoint
14:8	Device Address
7:0	PID

Table 2.34: Token [9]

Initially, it encompasses the packet identification, extending from bit 7 to 0, as elaborated in prior discussions. Subsequently, the device address occupies bits 14 to 8, followed by the endpoint address spanning bits 18 to 15. Bit 19 accommodates the toggle bit, crucial for synchronization, distinguishing between DATA0 and DATA1. [9], [11]

Moreover, bits 31 to 21 represent the maximum length of the data slated for transmission. The permissible range for this value extends up to 0x400, signifying the maximum data payload size. Notably, setting the value to 0x7FF signifies the absence of data transmission. [9], [11]

Concluding the TD, the final double word encompasses the buffer pointer, housing the physical buffer. This buffer, mandated to be contiguous in memory, serves as the conduit for transmitting or receiving data. [9], [11]

Example Stack Frame

Having gained insight into the structural intricacies of the USB communication framework, it becomes imperative to elucidate the interactions among these fundamental components.

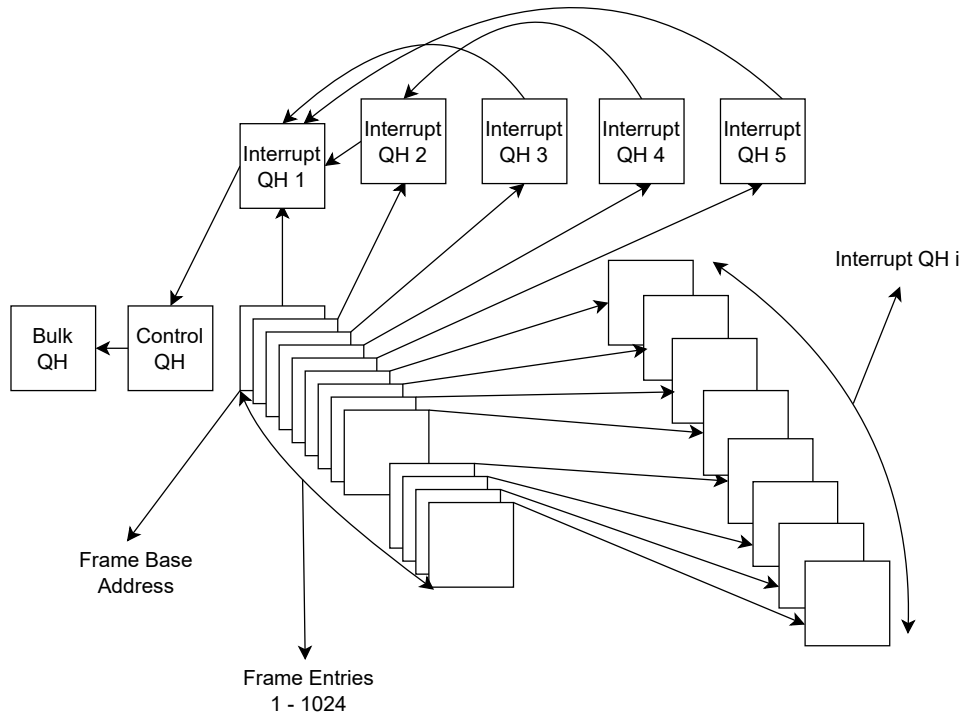


Figure 2.11: Example Schedule Overview

Typically, the frame list serves as the repository of pointers to the physical addresses of each QH, denoting a fixed interval, thereby establishing a periodic transfer schedule. For instance, a frame entry corresponding to a periodic transfer every 5ms would be linked to a QH representing this interval. To construct this periodic scheme cohesively, it is essential to interlink the QHs so that the periodic schedule maintains coherence. Consequently, when, for instance, a frame entry points to a QH indicating a transfer interval of 25ms, it necessitates linkage to the QH indicated by the frame entry corresponding to a 5ms interval, given the divisibility of 25 by 5. This sequential linking of QHs ensures the integrity of the periodic transfer schedule.

Once the QHs are interlinked, forming the foundational structure for periodic transfers occurring at predefined intervals, the integration of aperiodic transfers ensues within this framework. This integration involves establishing a pointer from the last QH executed within the frame list, typically the QH indicated by frame entry 0, to the subsequent QH representing the initial aperiodic transfer. Conventionally, this succeeding QH often pertains to control transfers. Subsequently, the linkage extends to the subsequent QH representing bulk transfers, thus entwining periodic and aperiodic transfers seamlessly within the structure.

Furthermore, examining the hierarchical arrangement within the frame list unveils a strategic bandwidth allocation scheme. Notably, the QHs corresponding to interrupt transfers occupy the foremost positions, thereby commanding the highest bandwidth allocation. Subsequently, QHs representing control transfers follow suit, being executed only after processing all interrupt transfer QHs. Lastly, the QHs pertaining to bulk transfers, occupying the trailing positions, are processed subsequent to handling all other QHs,

underscoring their lower priority in bandwidth allocation.

Queue Insertion

Now that a comprehensive understanding of the structural composition has been attained, it is pertinent to illustrate the procedural aspect through an exemplar scenario, elucidating the process of integrating transfers into this framework.

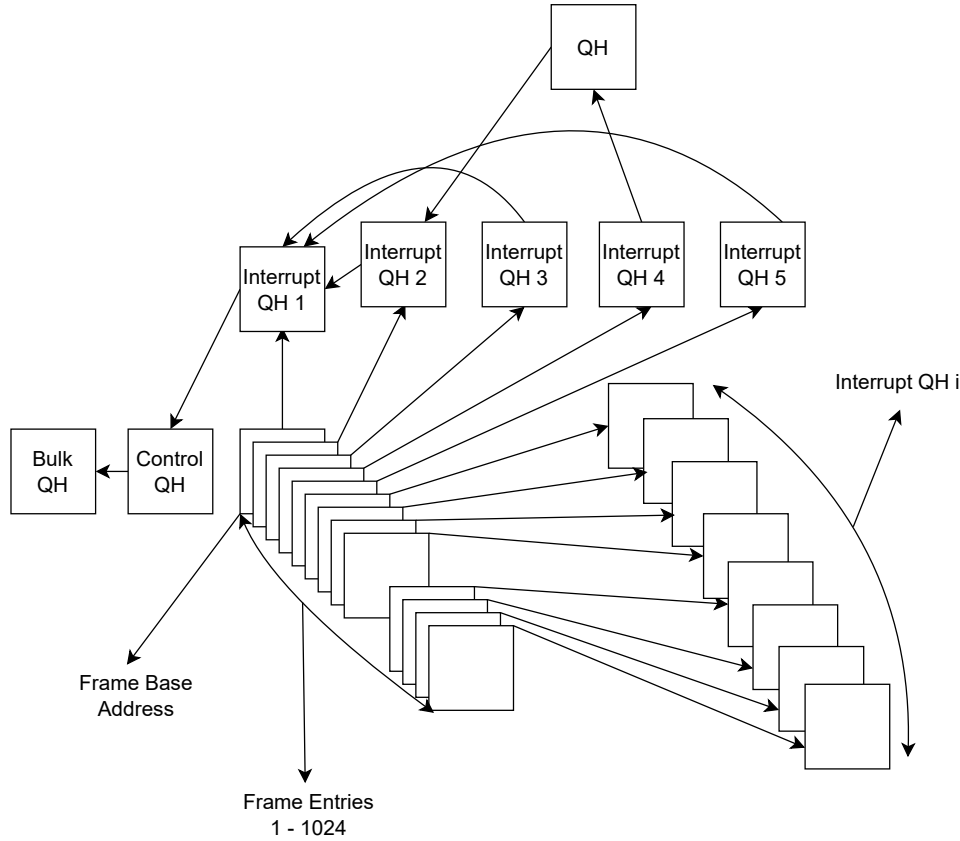


Figure 2.12: Example Schedule Interrupt Transfer Insertion

Consider the addition of an interrupt transfer to the frame structure, scheduled for periodic processing every 4ms, with no QHs linked to the QH indicated by frame 4. To achieve this integration, it becomes imperative to navigate to the QH pointed to by frame 4 and establish a linkage to the desired QH. Crucially, to uphold the integrity of the periodic schedule, the inserted QH must reference the QH currently pointed to by other QHs. Thus, the inserted QH would now be directed to frame 2, facilitating processing every 2ms. Subsequently, the QH pointed to by frame 4 would be reconfigured to point to the newly inserted QH, thereby ensuring the successful addition of the new QH while preserving the stability of the schedule.

In another scenario, envisage the incorporation of a classic control transfer.

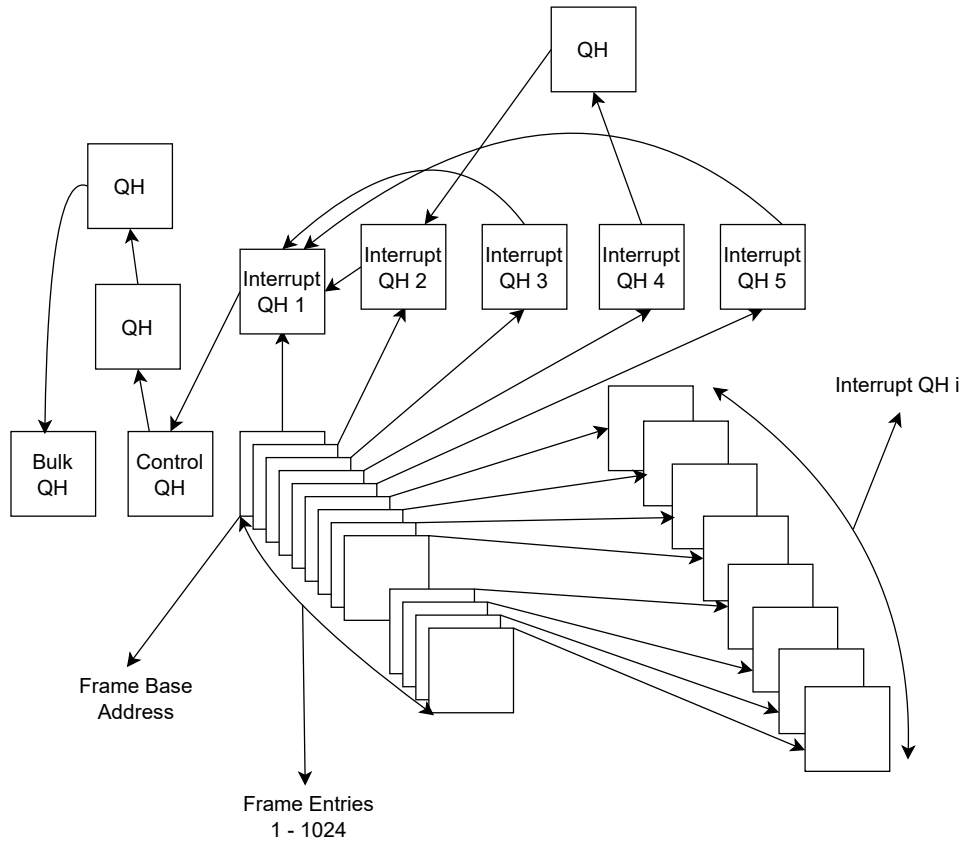


Figure 2.13: Example Schedule Control Transfer Insertion

Assuming the existence of one QH already linked to the QH representing the head of control transfers, the procedure involves bypassing this QH and establishing the connection using the aforementioned process. Consequently, the new QH would be linked to the subsequent QH of the existing one, with the current QH directed to the new QH. This methodology guarantees the preservation of the schedule without disruption.

For the sake of clarity and enhanced visual representation, the TDs associated with the inserted QHs have been omitted. However, in actuality, the structure of the QHs would resemble the following:

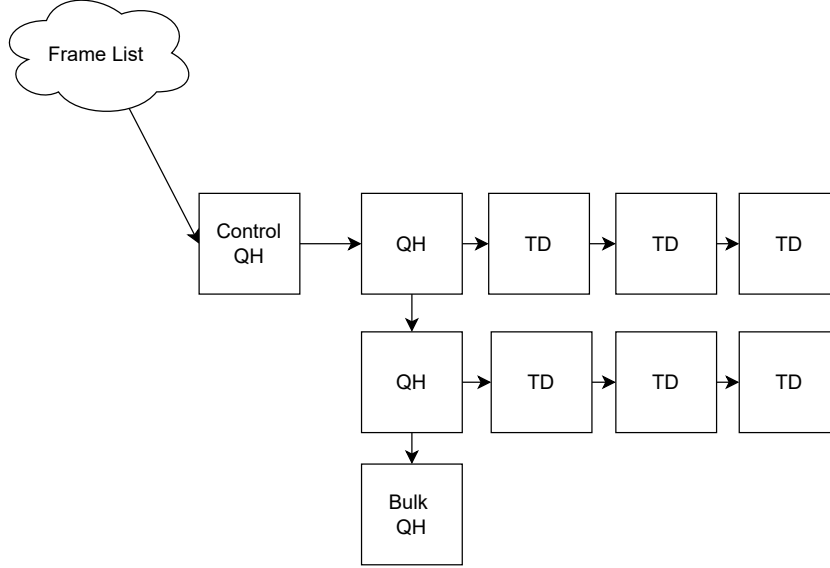


Figure 2.14: QH Visualization

It is noteworthy that the expounded concept serves as a conceptual framework for understanding the operational dynamics and is not directly implemented in this work. Nevertheless, grasping this fundamental concept is pivotal for subsequent discourse and comprehension.

Schedule Overview Low Level

To conclude the exposition provided herein, a closer examination will be conducted regarding the implementation of this schedule at lower levels. To facilitate this endeavor, the illustration below delineates the intricate linkage between QHs and TDs, forming a schedule that can be processed by the UHCI, while also elucidating the configuration of internal fields.

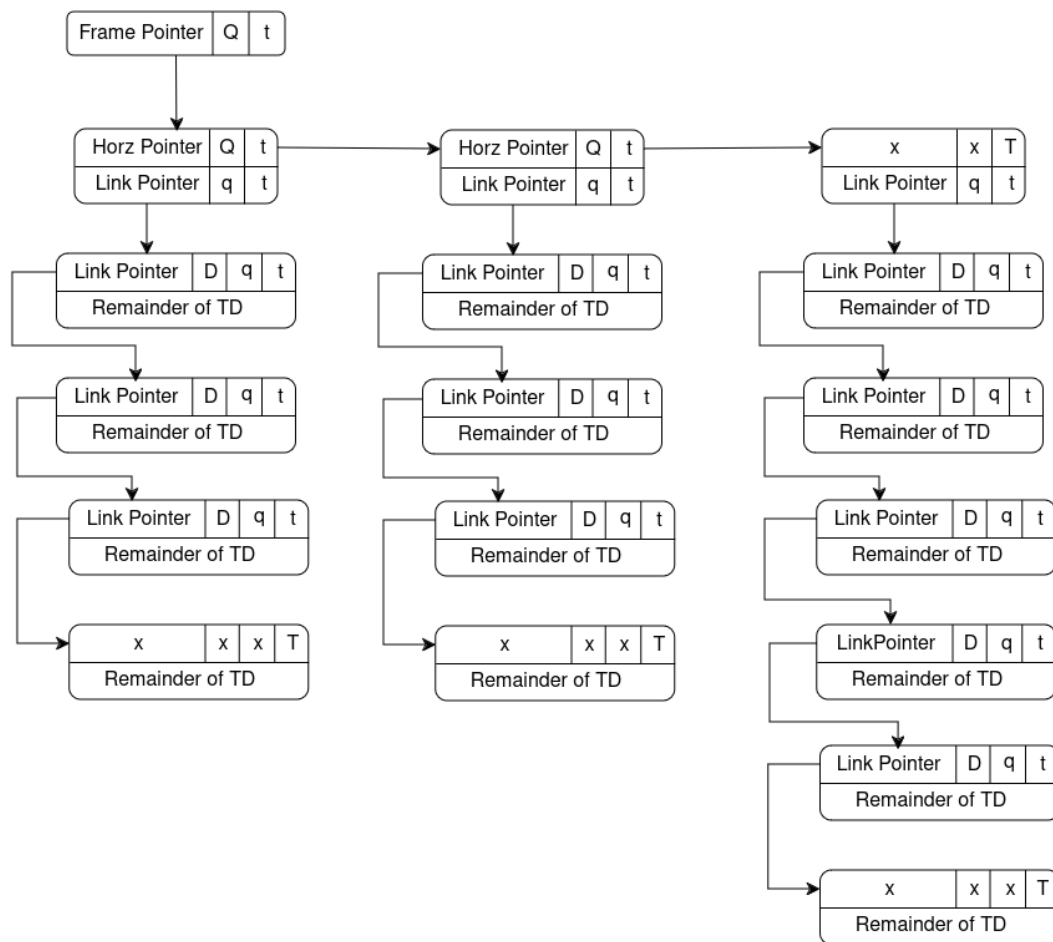


Figure 2.15: Schedule Example;Depth First Execution [11]

Prior to delving into the structural description, it is pertinent to note that the operational procedure consistently activates the depth-first bit. This signifies that the UHCI adheres strictly to the QH link pointer until either all TDs have been processed or an error ensues. To facilitate this, every TD within the schedule necessitates activation of the D flag. In the illustration provided above, this entails execution of all TDs associated with the first QH, followed by those of the second QH, and concluding with those of the third QH. [11]

This schedule encompasses three QHs, with the first containing a total of four TDs, the second containing four TDs, and the last containing six TDs. The initial QH is linked to the frame pointer, as indicated in the frame entry. Within this initial QH, the horizontal pointer field directs to the subsequent QH, denoted by the Q flag, signifying its role as a QH, while the t flag confirms its validity for navigation without error. Similarly, the link pointer, pointing to a TD, is accompanied by the q flag to indicate a TD. Traversing the link pointer to the first TD establishes a chain TD structure, whereby each subsequent TD is linked to its predecessor. Upon reaching the final TD, the T flag is activated, signaling the UHCI to transition to the next QH connected to the current one. This flag renders irrelevant other marked fields, denoted by 'x'. Subsequent QHs and chained TDs follow an analogous pattern to their preceding counterparts. [11]

In the case of the last QH, the sole distinction lies in indicating the absence of a subsequent QH. Thus, the T flag is set to signify that the link should not be pursued further. [11]

Process

Having elucidated the requisite configuration of internal bits within the QH and TD, it is imperative to comprehend the operational mechanism governing the UHCI's processing of this schedule. The ensuing diagram delineates the procedural workflow of the UHCI, commencing with the presence of an extant QH, exemplified in the aforementioned instance by the initial QH.

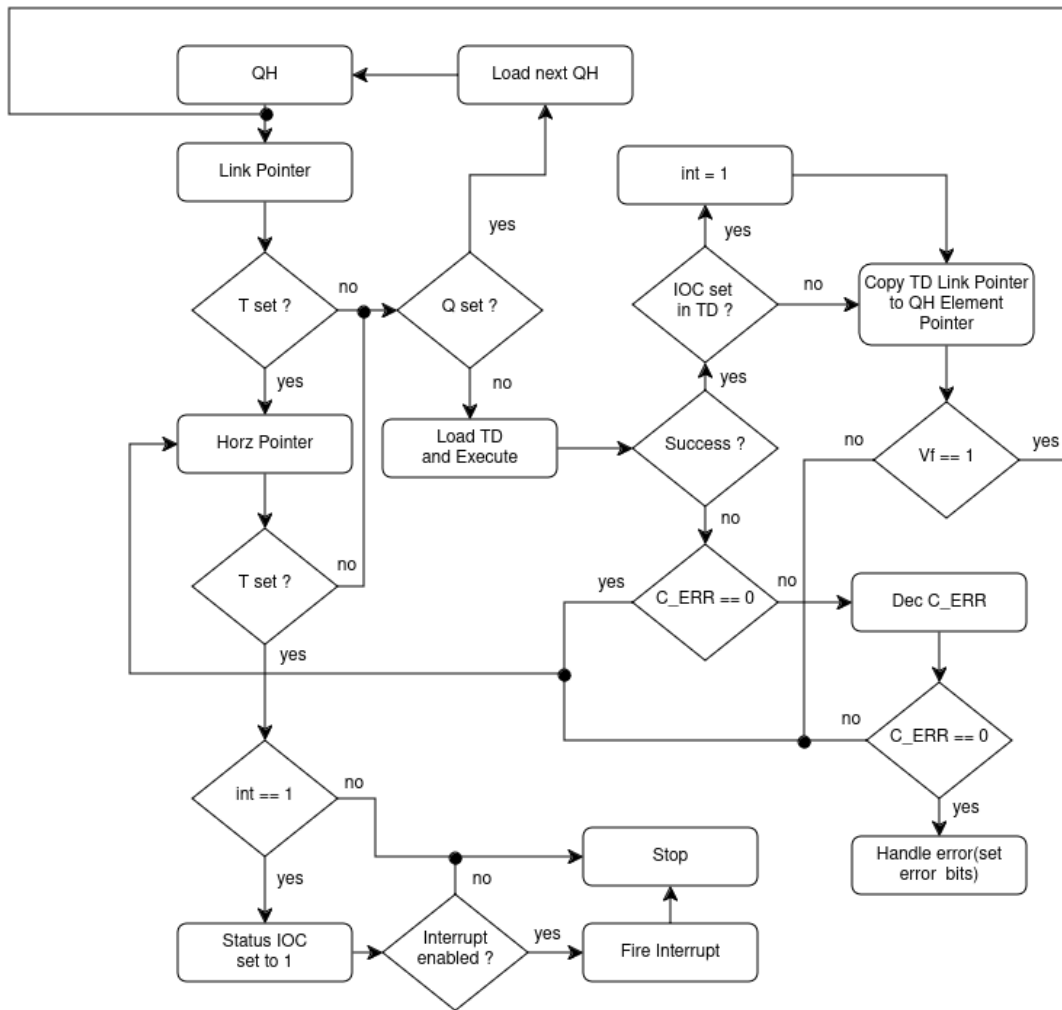


Figure 2.16: UHCI Process [11]

The UHCI initiates its processing routine by examining the link pointer associated with the current active QH. In the event of an invalid pointer, indicating the absence of an associated TD, the UHCI redirects its attention to the horizontal pointer, scrutinizing the status of the T flag. Upon encountering a set T flag, indicative of the completion of processing for each QH, the UHCI proceeds to evaluate the necessity of triggering an interrupt. This interrupt is dispatched if the UHCI is configured to utilize interrupts upon completion (IOC), persisting until the IOC bit is cleared within the status register

by the driver. Conversely, in the absence of a set T flag, the UHCI proceeds to examine the nature of the horizontal pointer, discerning between a QH or TD. Should the Q flag be activated, signifying the presence of a QH, the UHCI loads the QH indicated by the horizontal pointer. Conversely, if the q flag is detected, denoting a TD, the UHCI loads and executes the TD. Subsequently, upon successful execution of the TD, the UHCI verifies the status of the IOC bit within the TD. If set, an internal indicator is activated, signaling the occurrence of an interrupt at the frame's conclusion. Subsequent to the IOC assessment, the UHCI proceeds to copy the link pointer, indicating the subsequent TD, into the element pointer field within the current active QH. For instance, upon the frame's conclusion, the QH retains the TD with which to proceed during subsequent frame execution. Following the link pointer copying process, the UHCI deliberates upon whether to pursue depth or breadth execution. In the case of depth execution, the UHCI reiterates the aforementioned process, progressing through the copied link pointer. However, in the event of unsuccessful transfer execution, the UHCI evaluates the threshold of errors reached. Upon reaching the threshold, the UHCI ceases processing of the TDs within the current QH, transitioning instead to the subsequent QH, as indicated by the horizontal pointer. Should the threshold not be met, the UHCI decrements the C-ERR field. Should this decrement result in the permissible threshold of errors being exceeded once more, the UHCI sets the status bits within the TD, indicating the occurrence of a transfer error. [11]

Returning to the provided example, application of the aforementioned conceptual framework entails the UHCI loading the initial QH and initiating execution of the first TD. Following successful transmission, the link pointer within the current QH is updated, facilitating execution in the subsequent cycle. This iterative process ensues for all remaining TDs within the QH, culminating with the examination of the T flag upon reaching the final TD. Upon detection of the T flag, signifying the conclusion of TD processing within the current QH, the UHCI proceeds to load the subsequent QH, as indicated by the current QH. This sequential progression persists until the final QH is encountered, wherein detection of the T flag within the QH signals the cessation of processing (potentially triggering an interrupt), indicative of the absence of further transfers to be processed within the frame. [11]

Having assimilated the fundamental information requisite for comprehending the underlying mechanism, the subsequent section of this thesis embarks upon the implementation phase. Within this segment, a detailed discussion ensues regarding the construction of the entire system, augmented by code snippets and visual aids aimed at fostering enhanced clarity. Commencing with an elucidation of the architectural framework, the discourse progresses to a meticulous examination of each component individually. This entails a thorough analysis of diverse components, spanning from the UHCI driver implementation to the integration of multiple device drivers.

Before delving further into the intricacies, it is essential to clarify that the actual communication protocol for interfacing with specific devices is not exhaustively integrated here but rather utilized within the system to facilitate communication with said devices. For individuals seeking a deeper understanding of the internal workings of these drivers, an additional resource is recommended, providing detailed insights into their functionality [3]. Furthermore, it is noteworthy that the elucidation of how these drivers were implemented into the system serves primarily to illustrate their integration into the overarching system architecture.

Chapter 3

USB System Implementation

This chapter aims to provide a comprehensive overview of the implementation of the USB system within the operating system. The integration of this system entails the utilization of the UHCI as the primary communication controller for interfacing with USB devices. The chapter is structured into two distinct sections, each delineating critical aspects of the implementation process.

The initial section embarks on an exploration of the system's design considerations, commencing with an in depth analysis of design decisions pertinent to the UHCI. Subsequently, the broader design of the entire system is examined, encompassing component analysis and functional assessments. This section further delves into the practical implementation aspects, ranging from the selection of utilities employed in the implementation process to the actual integration of components into hhuOS. Additionally, it discusses the requisite modifications made to existing components within hhuOS to accommodate the integration of the USB system.

3.1 Design/Architecture

The subsequent section offers a detailed examination of the system's architecture, providing insights into its structural composition and operational framework. This analysis aims to elucidate the underlying architectural principles guiding the integration of the USB system.

3.1.1 UHCI Custom QH Field

As articulated in the preceding section, with emphasis QHs, it was elucidated that the double word located at offset 0x0C serves an internal purpose, functioning as a repository for specific flags. These flags, enumerated in the ensuing table, serve varied operational functions within the context of QH management.

Commencing from the least significant bit (bit0), a bit field is dedicated to indicating whether the QH marks the terminus of the particular sub-schedule it pertains to. This bit assumes significance during both insertion and removal operations within said sub

Bits	Description
31:9	TD/QH Count
8	Reserved
7:6	Transfer Type
5	MQH or QH
4:1	Priority
0	Last Queue Head

Table 3.1: QH Flags

schedule. Subsequently, bits bit1 through bit4 constitute a three-bit field, collectively encoding eight distinct priority levels, ranging from 0 denoting the lowest priority to 7 signifying the highest. By default, this field is initialized to the maximum priority for preliminary transfers, yet affords the flexibility for adjustment by a USB device driver when dispatching transfers to the UHCI driver.

Progressing, bit5 functions as a binary indicator discerning whether the QH participates in the construction of a specific sub-schedule, thereby distinguishing it as a Main Queue Head (MQH), or if it serves solely as a constituent QH within the sub-schedule. A logical high value in this bit designates the QH as an MQH, while a logical low value indicates otherwise.

Continuing, a two-bit segment is allocated for encoding the four distinct transfer types associated with the transfers. Bit8 remains reserved and immutable, mandated to retain a value of 0. Subsequent bits, following the reserved bit, serve to enumerate the quantity of QHs linked to a particular schedule, contingent upon whether the QH in question functions as an MQH. In scenarios where the QH is an MQH, this field tallies the quantity of QHs appended to the respective schedule. Conversely, if the QH does not serve as an MQH, this field denotes the quantity of TDs associated with it.

3.1.2 UHCI Schedule

Before delving into the intricate architecture of the system, an essential precursor lies in comprehending the construction of the schedule within the framework. A notable departure from the previously elucidated schedule pertains to the interrupt schedule, wherein constraints are imposed solely permitting specific intervals for transfers. To elucidate the composition of this schedule, attention is directed towards an accompanying illustration, pivotal in elucidating the underlying concept.

This approach is similarly employed in the referenced source [11], which has been adapted for integration into the implementation.

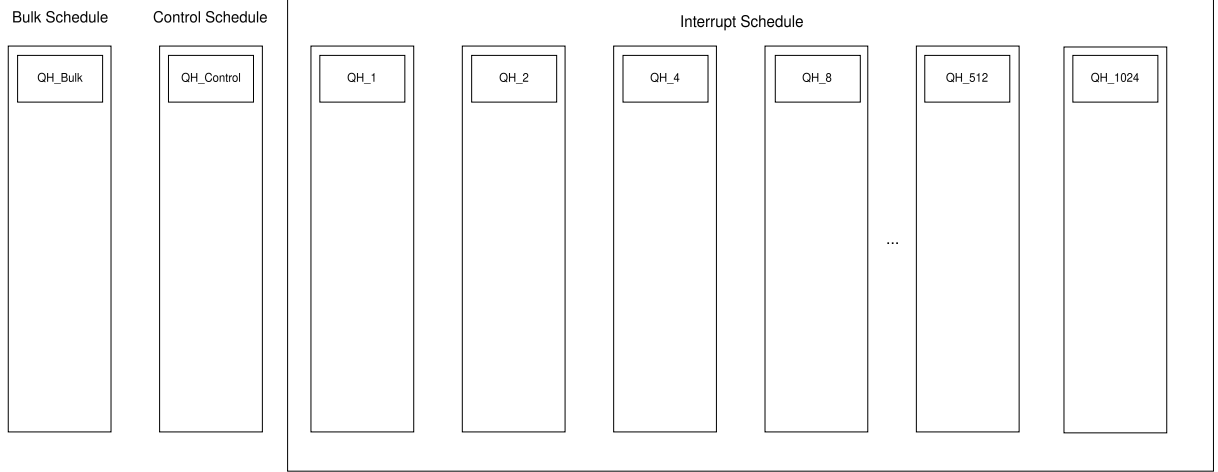


Figure 3.1: Transfer Schedule

Examination of the illustration reveals the persistence of three sub schedules: the interrupt schedule, the control schedule, and the bulk schedule. However, a noteworthy distinction manifests in the interrupt schedule, where only intervals adhering to the form 2^x , where $x \in [0, 10]$, are sanctioned. [11]

The overarching structure is comprised of 11 QHs constituting the foundational framework, augmented by an additional 2 QHs dedicated to the control and bulk schedules. A pertinent query arises concerning frame entries not tethered to any of the QHs comprising the framework, owing to unsupported intervals. This quandary finds resolution in a systematic approach. For a given frame entry, denoted as 'i', a traversal across supported intervals commences, commencing from the highest supported interval, namely 1024. The initial interval that remains indivisible by 'i' is sought, whereupon the frame entry is linked to the corresponding QH associated with the identified interval. [11]

This approach, albeit efficacious, is not devoid of drawbacks and merits. An inherent limitation surfaces in the prescribed utilization of specific intervals, potentially incompatible with devices interfacing with the controller. Consequently, instances arise where interrupts, occurring at intervals not aligned with those specified, are rounded down to the nearest supported interval, engendering surplus interruptions and compromising timing precision. This surplus interruption may entail adverse consequences, inducing operational instability, particularly for devices reliant on precise timing mechanisms. [11]

Conversely, the salient advantage of this methodology lies in its judicious utilization of memory resources. By limiting the total QH count to 11, in contrast to a theoretical maximum of 1024, a substantial memory saving is achieved. The resultant memory conservation is calculated as $(1024 - 11) \times 16$ bytes, equating to 16208 bytes.

3.1.3 System Overview

Following the elucidation of the schedule's construction, a high level overview of the system is provided, presenting an abstract perspective of its architecture.

Commencing at the apex of the hierarchy, the `UsbNode` component assumes prominence,

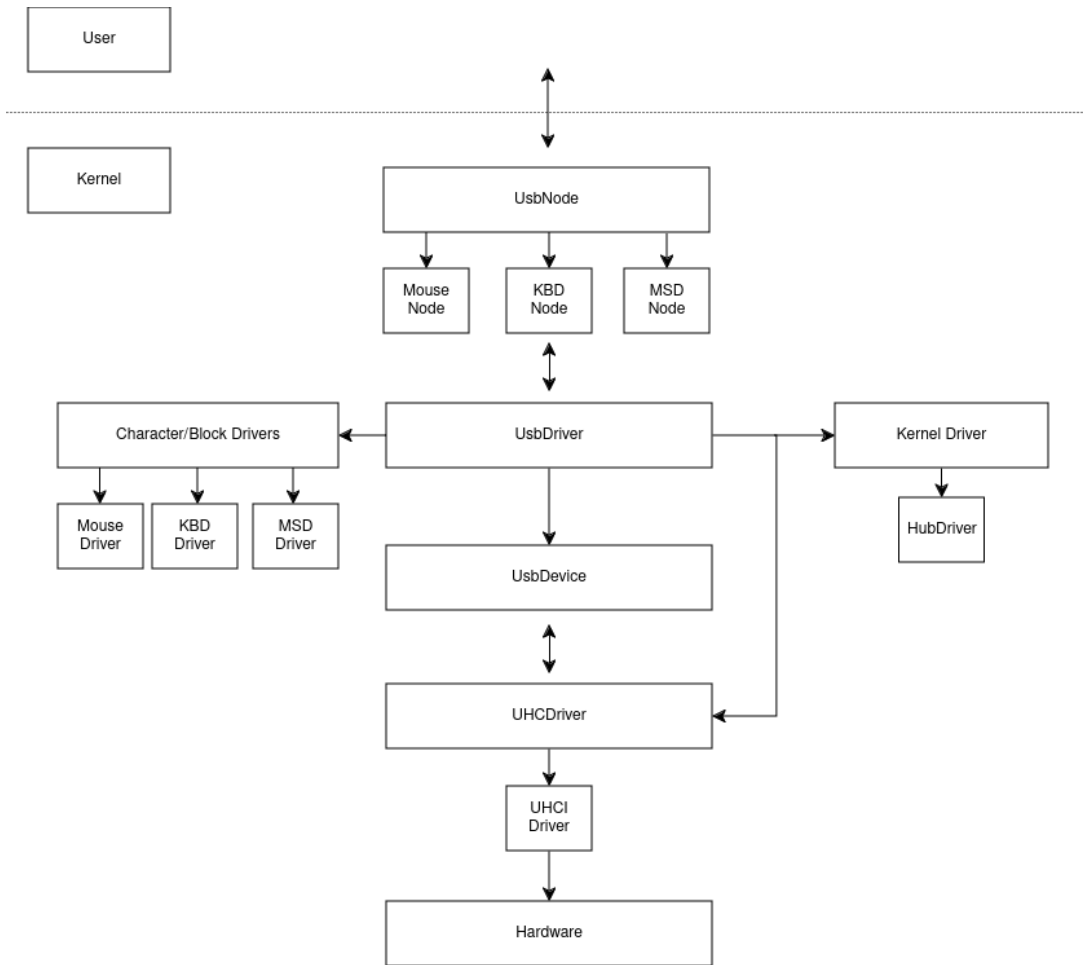


Figure 3.2: System Overview

serving as the interface facilitating communication between user space and USB devices domiciled within the kernel space. This interface is manifested as a file within the file system, thereby rendering it accessible to users. Presently, the system extends support to several `UsbNodes`, including those corresponding to mouse, keyboard, and mass storage device functionalities.

Subsequent to the `UsbNode` component, the `UsbDriver` component is encountered, intricately linked to a specific `UsbNode`. Within this component, the internal logic encompassing tasks such as reading and writing is executed. Notably, the `UsbDriver` is subdivided into Character/Block Drivers and Kernel Drivers. The latter category, devoid of association with any node, resides exclusively within the kernel space, precluding direct access from user space. Conversely, the former drivers are tethered to specific nodes, thereby enabling interaction with user space. Within the realm of Kernel Drivers, the solitary component comprises the `HubDriver`. Conversely, each remaining driver corresponds to its associated node.

Furthermore, each `UsbDriver` possesses the capability to accommodate one or multiple `UsbDevice`. Each instance of the `UsbDevice` component corresponds to a physical device interfacing with the UHC. Given that transfers are invariably instigated by the UHC, a bidirectional communication conduit is established between the `UHCI` and `UsbDevice`.

In the context of the discussion, with emphasis on the UHCI, it is imperative to note that support is exclusively extended to the UHCI. Consequently, the system exclusively supports the `UHCI` component in this capacity.

With a foundational understanding of the comprehensive system structure established, a more nuanced exploration of its intricacies is embarked upon. The fundamental aim underlying the system's design endeavor was the creation of an extensible framework capable of accommodating the integration of new drivers seamlessly, without necessitating substantial modifications to the overarching components. To achieve this objective, a bifurcation of the system into two distinct subsystems was conceived, each operating autonomously and interfacing with one another through a designated transmitter component. Facilitating this inter-subsystem interaction is the `UsbService`, serving as the pivotal conduit through which both subsystems engage in mutual communication and coordination.

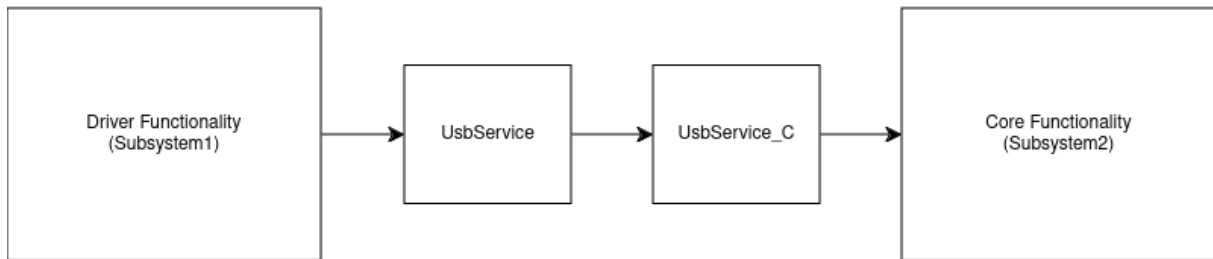


Figure 3.3: Interaction Subsystems

The illustrative depiction delineates the bifurcation of the system into two distinct subsystems: subsystem1 and subsystem2, each assuming delineated roles within the overarching architecture. Subsystem1 is designated as the repository wherein the functionalities of all drivers are defined, delineating the operational parameters and behaviors of the individual drivers. In contrast, subsystem2 encapsulates the core logic underpinning the USB system, orchestrating the intricate interplay between various components to facilitate the seamless operation of USB devices within the system.

Facilitating communication and interaction between these disparate subsystems is an intermediary component known as the `UsbService`. Positioned strategically within the system architecture, the `UsbService` serves as the linchpin that enables the exchange of data, commands, and signals between subsystem1 and subsystem2, ensuring cohesive operation and coordination across the entirety of the system.

3.1.4 Core System Component Analysis

Presently, an examination of each subsystem in isolation shall be undertaken, commencing with the core subsystem. To facilitate this endeavor, a component diagram has been prepared, delineating the modular framework inherent to this particular subsystem.

- **UsbController:** This component serves as an interface facilitating communication with other system components. It encompasses functionalities such as submitting transfers, including interrupt, control, and bulk transfers, as well as handling the registration and deregistration of drivers, and the initialization of the controller itself.
- **UHCI:** This component embodies a specific implementation of a controller within the USB system.
- **UsbDevice:** Representing a physical device interfaced with the controller, this component facilitates communication with the device, encompassing operations such as sending and receiving data, and handling the initialization procedures of the device.
- **DeviceDescriptor:** This component serves as a data container, encapsulating the descriptor information of the device.
- **Configuration:** Representing another data container, this component encapsulates the configuration details of the device.
- **Interface:** This component acts as a data container, encapsulating the interface information of the device.
- **Alternate_Interface:** Serving as yet another data container, this component encapsulates the information pertaining to an alternate setting of an interface.
- **InterfaceDescriptor:** This component serves as a data container, encapsulating the descriptor information of an interface.
- **Endpoint:** Representing a data container, this component encapsulates the information pertaining to an endpoint of the device.
- **EndpointDescriptor:** Serving as a data container, this component encapsulates the descriptor information of an endpoint.
- **UsbDriver:** This component embodies a driver and encompasses functionalities essential for driving devices within the system.
- **SystemService_C:** Representing an abstract service, this component provides a foundational infrastructure within the system.
- **MemoryService_C:** This component represents a specific implementation of a system service, further providing interfaces for memory operations.
- **EventDispatcher:** Serving as a crucial component, this facilitates the orchestration of various listeners, providing functionalities for the registration and deregistration of listeners, and additionally serving as a mechanism for event publication.
- **EventListener:** This component is designed to receive events and transmit them to the registered callbacks.
- **Event_Callback:** Representing a data container, this component encapsulates a callback mechanism for sending events.

- **KeyboardListener:** This component represents a listener specialized for keyboard events.
- **MouseListener:** Similar to the KeyboardListener, this component specializes in handling mouse-related events.
- **Mutex_C:** This component serves as a wrapper, providing interfaces for synchronization operations.
- **SuperMap:** Representing a wrapper component, this facilitates map operations within the system.
- **EventMap:** This component represents a specific implementation of a map, specialized for managing events.
- **Interface_Device_Map:** Specialized for managing the interface-device mappings, this component serves as another specific implementation of a map within the system.

As the overview of the components within the core subsystem concludes, attention now shifts to a more focused examination of components specific to the `UHCI`.

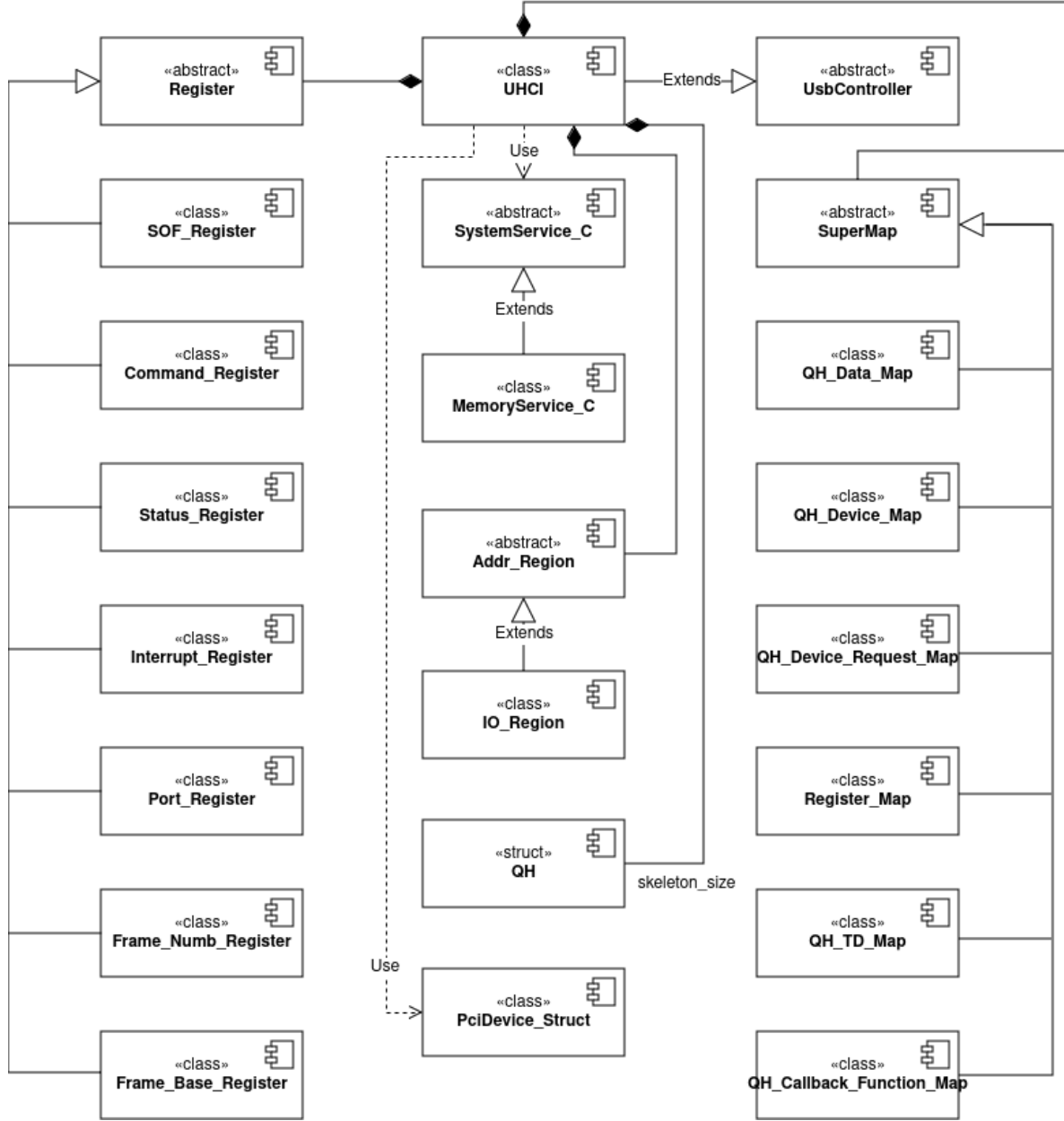


Figure 3.5: Core System Components 2

- **Register:** Serving as an essential component within the controller architecture, this entity embodies a register, housing functionalities for reading, writing, clearing, and setting the contents within said register.
- **SOF_Register:** Representing a specialized implementation of a register, this component pertains specifically to the management of the Start-of-Frame (SOF) register within the controller.

- **Command_Register:** This component serves as a specialized register implementation, dedicated to the management of command-related functionalities within the controller.
- **Status_Register:** Encompassing a specialized register implementation, this component is tasked with managing status-related information within the controller.
- **Interrupt_Register:** Representing another specialized register implementation, this component oversees the management of interrupt-related functionalities within the controller.
- **Port_Register:** This component embodies a specialized register implementation, focusing on the management of port-related functionalities within the controller.
- **Frame_Numb_Register:** Serving as a specialized register implementation, this component is dedicated to managing frame number-related functionalities within the controller.
- **Frame_Base_Register:** This component represents a specialized register implementation, specifically tailored for managing frame base-related functionalities within the controller.
- **Addr_Region:** Acting as a fundamental component within the controller architecture, this entity represents the address region of the controller, incorporating functionalities for reading from and writing to said region.
- **IO_Region:** This component serves as a specialized implementation of an address region, specifically tailored for input/output operations within the controller.
- **PciDevice_Struct:** Representing a wrapper component, this entity encompasses functionalities for reading, writing, and retrieving basic information pertaining to the associated PCI device.
- **QH:** Serving as a crucial component within the system, this entity embodies a data container encapsulating Queue Head information.
- **SystemService_C:** As previously described.
- **MemoryService_C:** As previously described.
- **SuperMap:** As previously described.
- **QH_Data_Map:** This component represents a specialized implementation of a map, specifically tailored for managing Queue Head data within the system.
- **QH_Device_Map:** Representing yet another specialized map implementation, this entity is dedicated to managing the mapping between Queue Heads and devices within the system.
- **QH_Device_Request_Map:** This component embodies a specialized map implementation, focusing on managing the mapping between Queue Heads, devices, and requests within the system.

- **Register_Map:** Serving as a specialized map implementation, this entity is dedicated to managing registers within the system.
- **QH_TD_Map:** Representing another specialized map implementation, this entity focuses on managing the mapping between Queue Heads and Transfer Descriptors within the system.

3.1.5 Driver System Component Analysis

Following an analysis of the structural composition of the core system, the subsequent focus lies on the examination of the drivers subsystem. This endeavor entails a meticulous exploration of the construction and organization of the drivers system, elucidating its architectural underpinnings and operational framework.

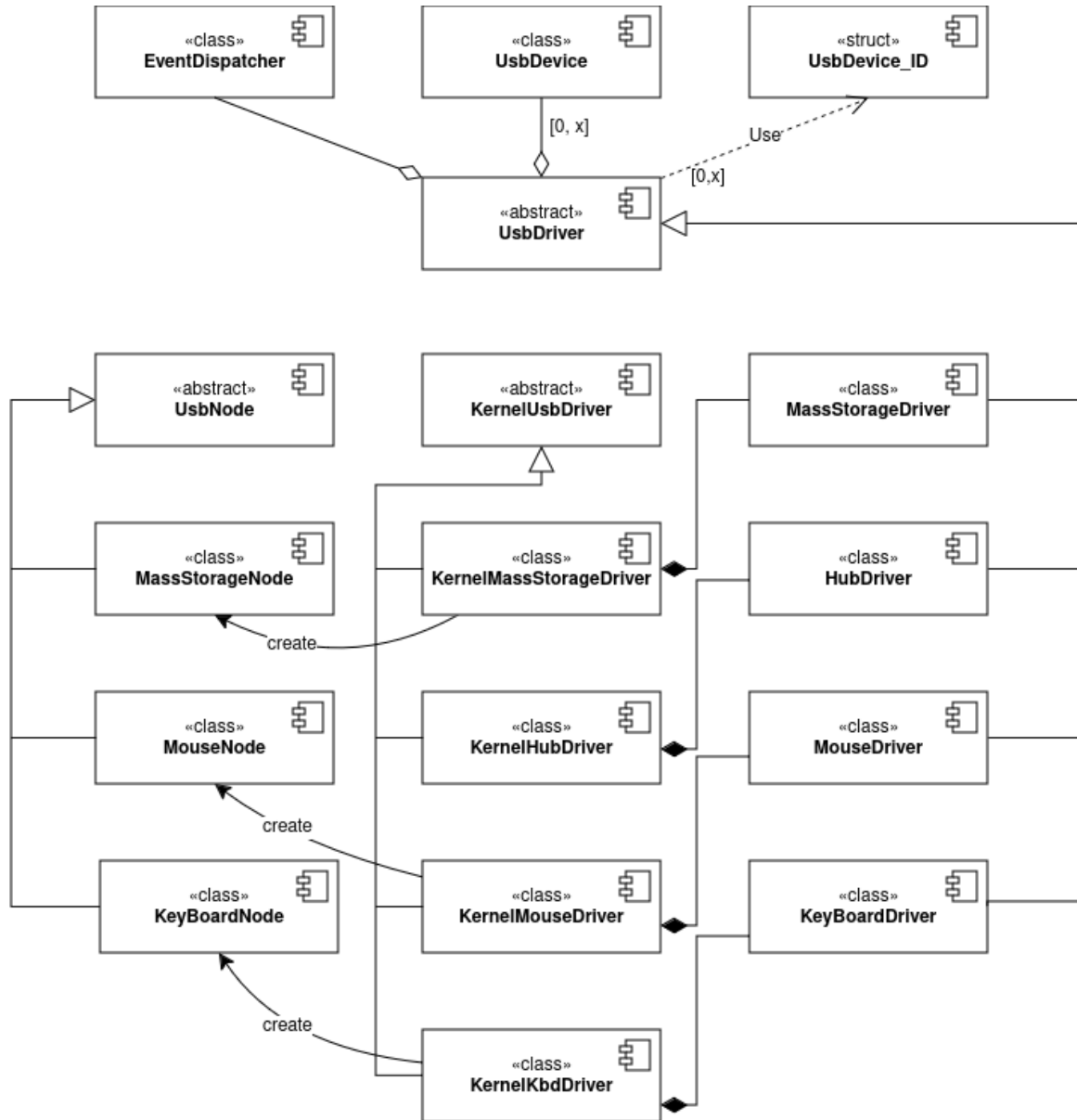


Figure 3.6: Driver System Components 1

- **UsbDriver:** This component serves as an interface facilitating interaction with other system components, encompassing functionalities pertinent to the probing and disconnection of specific devices or interfaces.
- **UsbDevice:** As previously described.

- **EventDispatcher:** As previously delineated, this component plays a pivotal role in orchestrating event-related functionalities within the system.
- **UsbDevice_ID:** Serving as a data container, this component encapsulates information pertaining to the identification of the device targeted by the driver.
- **MassStorageDriver:** This component embodies a specific implementation of a driver, furnishing functionalities tailored for the configuration, and reading/writing operations pertinent to mass storage devices.
- **HubDriver:** Representing another specific driver implementation, this component encompasses functionalities essential for the configuration of hub devices within the system.
- **MouseDriver:** This component embodies a specific driver implementation, specializing in handling mouse-related functionalities within the system.
- **KeyBoardDriver:** Serving as yet another specific driver implementation, this component is tailored for the management of keyboard-related functionalities within the system.
- **UsbNode:** This component represents a node within the USB system, housing functionalities for the generation of a corresponding node within the file system.
- **MassStorageNode:** Serving as a specific node implementation, this component is tailored for the management of nodes corresponding to mass storage devices within the file system.
- **MouseNode:** Representing another specific node implementation, this component is dedicated to managing nodes specific to mouse devices within the file system.
- **KeyBoardNode:** This component embodies a specific node implementation, specialized for the management of nodes corresponding to keyboard devices within the file system.
- **KernelUsbDriver:** This wrapper component encapsulates functionalities for the initialization, submission, and creation of device nodes within the kernel environment.
- **KernelMassStorageDriver:** Representing a specific kernel driver implementation, this component encompasses functionalities essential for reading/writing operations and control mechanisms pertaining to mass storage devices.
- **KernelHubDriver:** Serving as another specific kernel driver implementation, this component is tailored for the management of hub devices within the kernel environment.
- **KernelMouseDriver:** This component embodies a specific kernel driver implementation, specializing in handling mouse-related functionalities within the kernel environment.

- **KernelKbdDriver:** Representing yet another specific kernel driver implementation, this component is dedicated to managing keyboard-related functionalities within the kernel environment.

Having delineated the components comprising the drivers subsystem, the focus now shifts to a meticulous examination of each component's internal structure and functionality. In the subsequent diagram, the specifics of each driver component are delved into, elucidating its architectural intricacies and operational characteristics within the broader system framework.

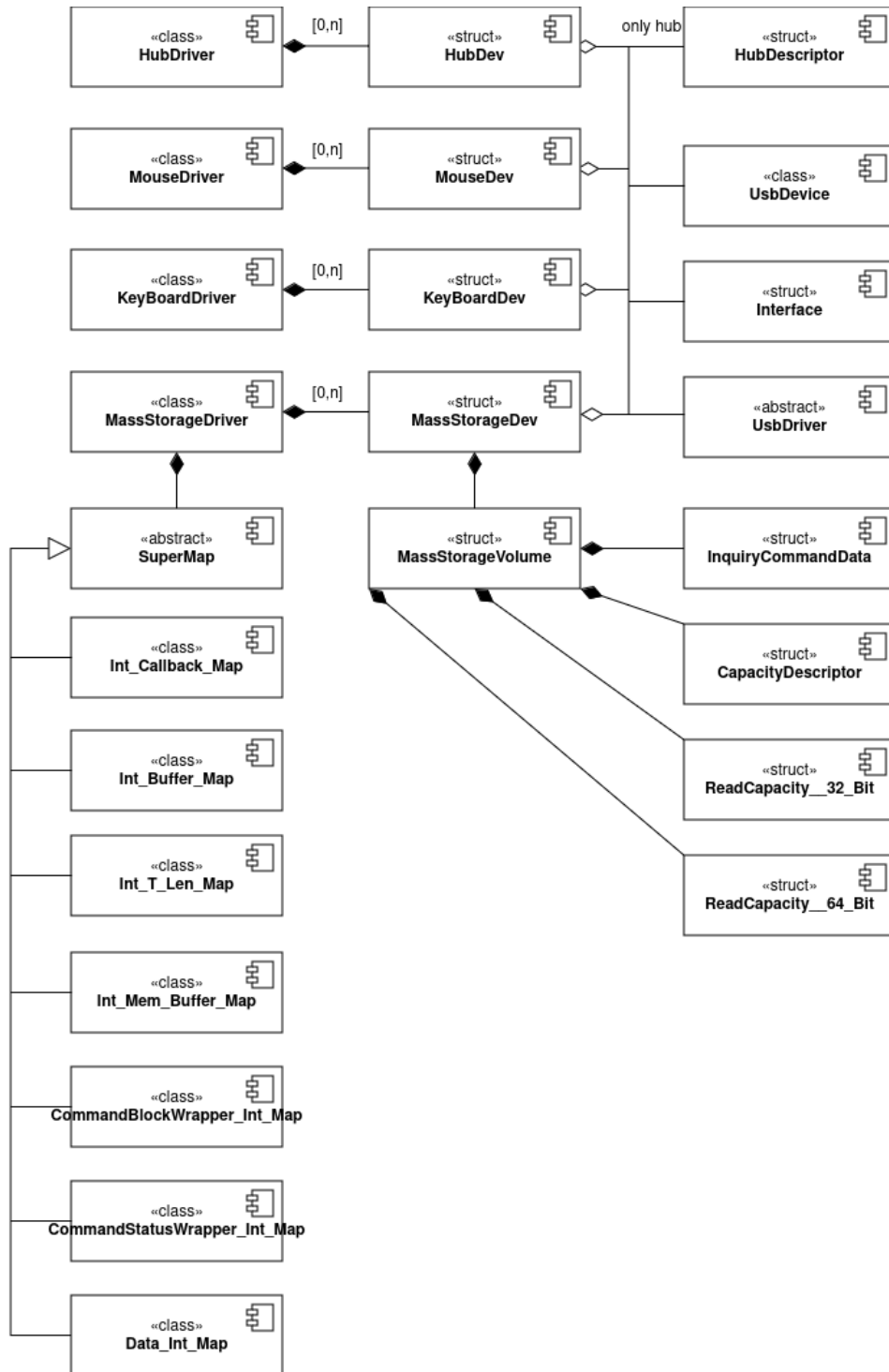


Figure 3.7: Driver System Components 2

- **UsbDriver:** As previously expounded upon.
- **UsbDevice:** As previously detailed.
- **Interface:** As described previously.
- **HubDriver:** As previously discussed, playing a crucial role in managing hub devices within the system.
- **HubDev:** Serving as a data container, this component represents an hub device associated with the hub driver.
- **HubDescriptor:** This component functions as a data container, embodying the descriptor of a hub device.
- **MouseDriver:** As previously elucidated, specialized in managing mouse-related functionalities within the system.
- **MouseDev:** This component serves as a data container, representing a mouse device associated with the mouse driver.
- **KeyboardDriver:** As previously elaborated upon, dedicated to managing keyboard-related functionalities within the system.
- **KeyboardDev:** Representing a data container, this component embodies a keyboard device associated with the keyboard driver.
- **MassStorageDriver:** As previously outlined, specialized in managing mass storage devices within the system.
- **MassStorageDev:** Serving as a data container, this component represents a mass storage device associated with the mass storage driver.
- **MassStorageVolume:** This component functions as a data container, representing a volume within the mass storage device.
- **InquiryCommandData:** Embodied within this component is the data returned upon execution of an inquiry command request.
- **CapacityDescriptor:** Representing a data container, this component contains the descriptor returned upon execution of a capacity command request.
- **ReadCapacity__32__Bit:** This component serves as a data container, encapsulating the read capacity (32-Bit) returned upon execution of a read capacity command (10).
- **ReadCapacity__64__Bit:** Representing a data container, this component encapsulates the read capacity (64-Bit) returned upon execution of a read capacity command (16).
- **SuperMap:** As previously delineated.
- **Int__Callback__Map:** This component embodies a specific map implementation.

- **Int_Buffer_Map:** This component embodies a specific map implementation.
- **Int_T_Len_Map:** This component embodies a specific map implementation.
- **Int_Mem_Buffer_Map:** This component embodies a specific map implementation.
- **CommandBlockWrapper_Int_Map:** This component embodies a specific map implementation.
- **CommandStatusWrapper_Int_Map:** This component embodies a specific map implementation.
- **Data_Int_Map:** This component embodies a specific map implementation.

With an understanding of the constituent components employed within their respective subsystems, the subsequent analysis will focus on elucidating the manner in which these components interoperate, thereby comprehending the control flow intrinsic to these systems.

3.1.6 Interaction between Core System Components

The ensuing sequence diagram delineates the progression of messages and interactions entailed during the initialization process of the **UHCI**.

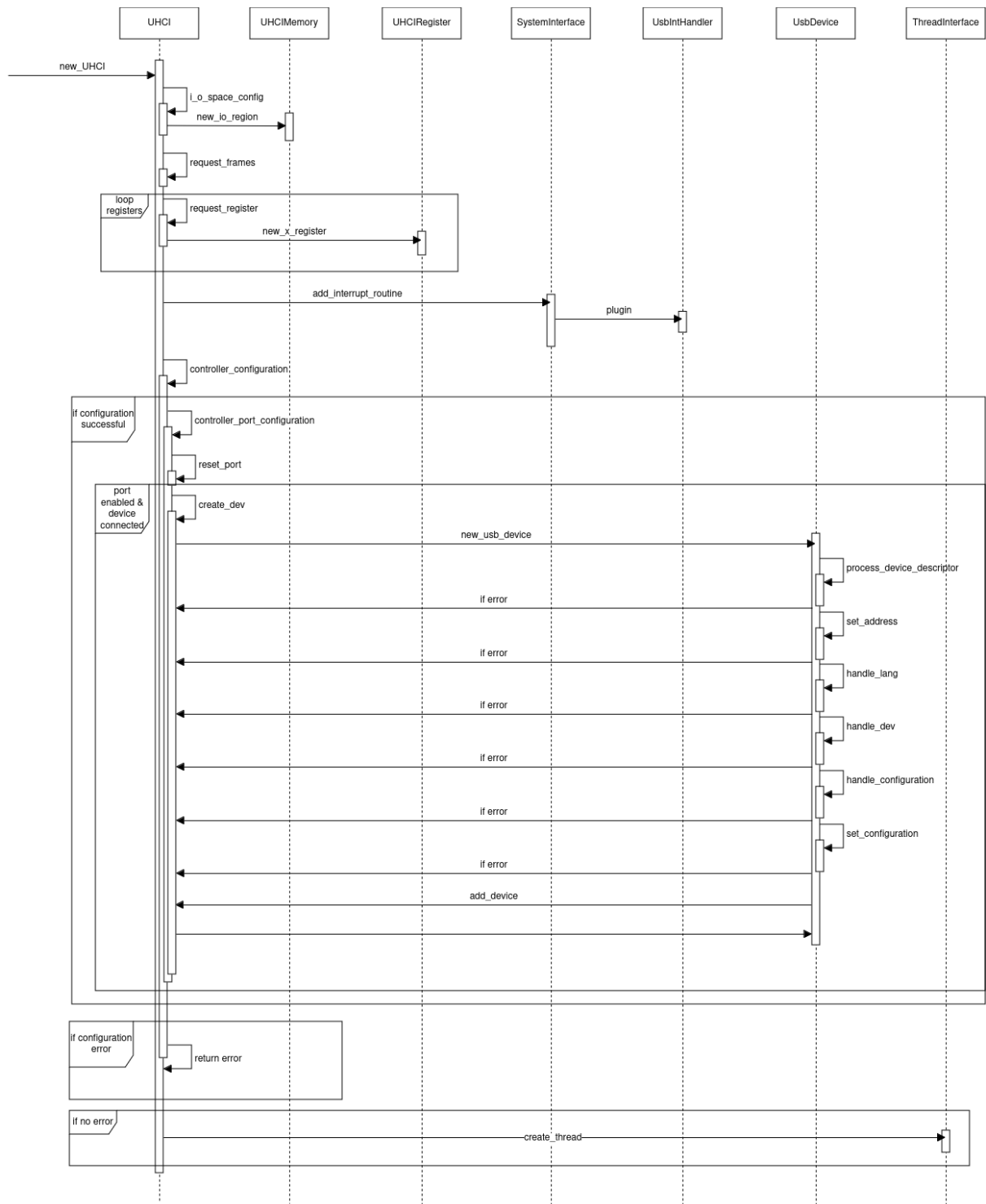


Figure 3.8: UHCI Initialization Process

The procedural steps involved in the process of initialize the `UHCI` are as follows:

1. **UHCI:** Initially, the `UHCI` receives a request from an external source.
2. **UHCIMemory:** Subsequently, the `UHCIMemory` component, upon receiving the request from `UHCI`, generates a new region and promptly returns.
3. **UHCI:** Upon receipt of the response from `UHCIMemory`, `UHCI` proceeds by requesting frames and iterating through all registers within its domain.
4. **UHCIRegister:** This component, upon receiving a request from `UHCI`, generates the requisite register and promptly returns.
5. **UHCI:** Upon completing the processing of all registers, `UHCI` dispatches a request to the `SystemInterface`.
6. **SystemInterface:** Upon receipt of the request from `UHCI`, `SystemInterface` forwards it to the `UsbInterruptHandler` for further processing.
7. **UsbInterruptHandler:** Upon receiving the request from `SystemInterface`, `UsbInterruptHandler` undertakes the necessary processing and subsequently returns.
8. **UHCI:** Following reception of the response from `SystemInterface`, `UHCI` initiates the controller configuration process.
9. **UHCI-config-state:** If the configuration process transpires successfully, `UHCI` proceeds to commence port configuration.
10. **UHCI-port-state:** Upon verifying that the port is enabled, `UHCI` requests the generation of a new device.
11. **UHCI-error-state:** In the event of an error occurrence, `UHCI` dispatches an error response.
12. **UsbDevice:** Upon receipt of a request from `UHCI-port-state`, `UsbDevice` generates a new USB device and issues a response if any errors are encountered during the device creation process.
13. **UHCI-port-state:** Receives response from `UsbDevice`. If response is error free, add device and sends success response.
14. **UHCI:** Subsequent to receiving the response and verifying its error-free status, `UHCI` requests the creation of a thread.
15. **ThreadInterface:** Upon receipt of the request from `UHCI`, `ThreadInterface` creates a thread as per the request.

The forthcoming sequence diagram illustrates the progression of message exchange and interactions entailed in the process of initiating an interrupt transfer.

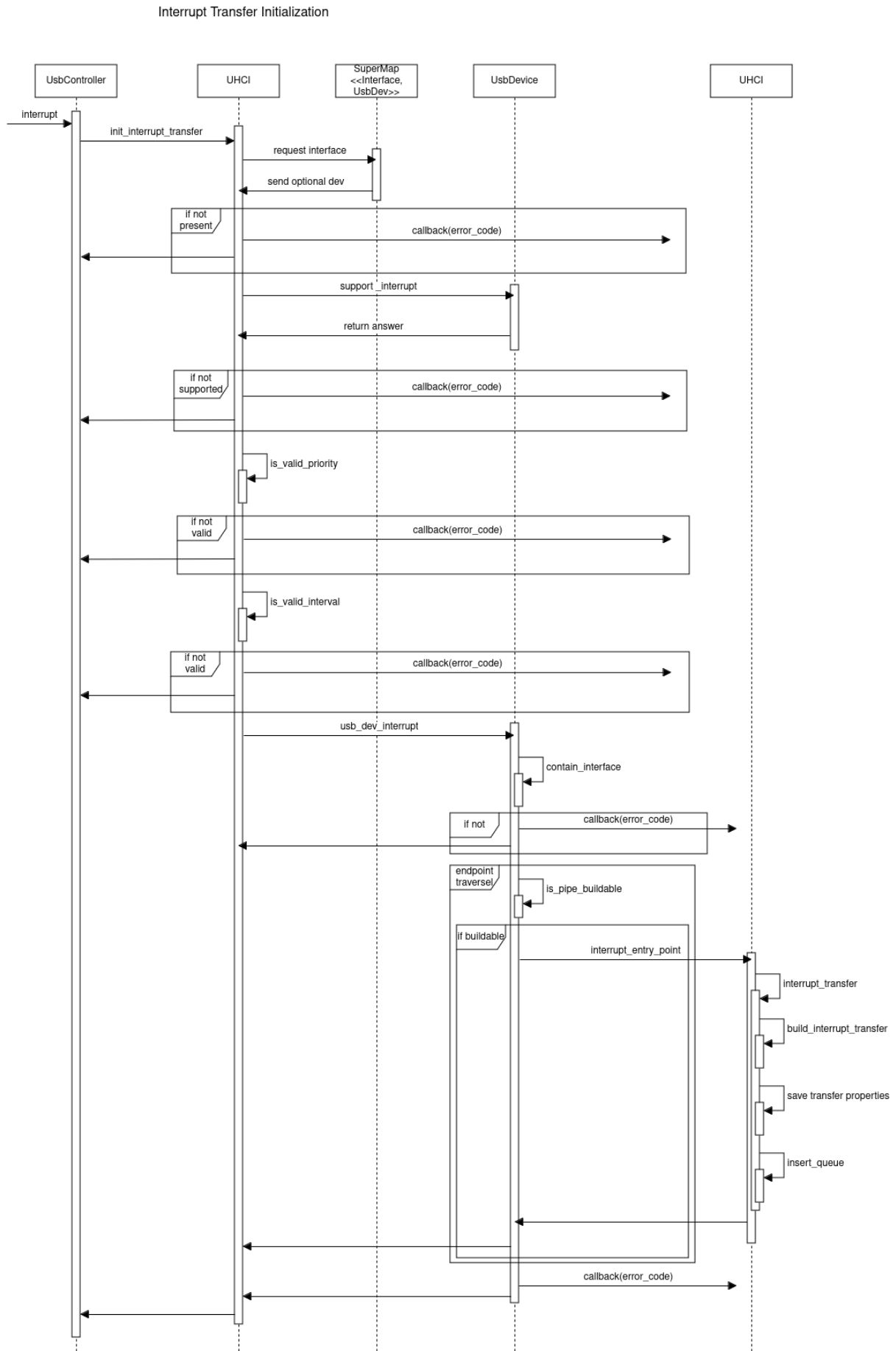


Figure 3.9: Interrupt Transfer Initialization Process

The procedural steps involved in the process of requesting an interrupt transfer are as follows:

1. **UHCI:** Receives request from outside and requests interface
2. **SuperMap:** Receives request from `UHCI`, processes the request and send an optional device as response.
3. **UHCI:** Receives response. If there is no device associated with the interface, call callback with error code and return, else perform request to `UsbDevice`.
4. **UsbDevice:** Receives request from `UHCI`, processes request and responses if the devices supports this specific endpoint.
5. **UHCI:** Receives response from `UsbDevice`. If endpoint is not supported, call callback with error code and return, else continue with validating. If validation success, perform a request to the `UsbDevice`.
6. **UsbDevice:** Receives request from `UHCI`. If interface is not supported, call callback with error code and returns. Else traverse through all available endpoints in the interface.
7. **UsbDevice-pipe-state:** If pipe is buildable, perform a request to the `UHCI`.
8. **UHCI:** Receives request from the `UsbDevice`, processes the request and returns.
9. **UsbDevice-pipe-state:** Receives response from `UHCI` and returns.
10. **UsbDevice-error-state:** If not buildable call callback with error code.

The subsequent sequence diagram delineates the orchestrated exchange of messages and interactions among distinct system components during the initiation of a bulk transfer.

Bulk Transfer Initialization

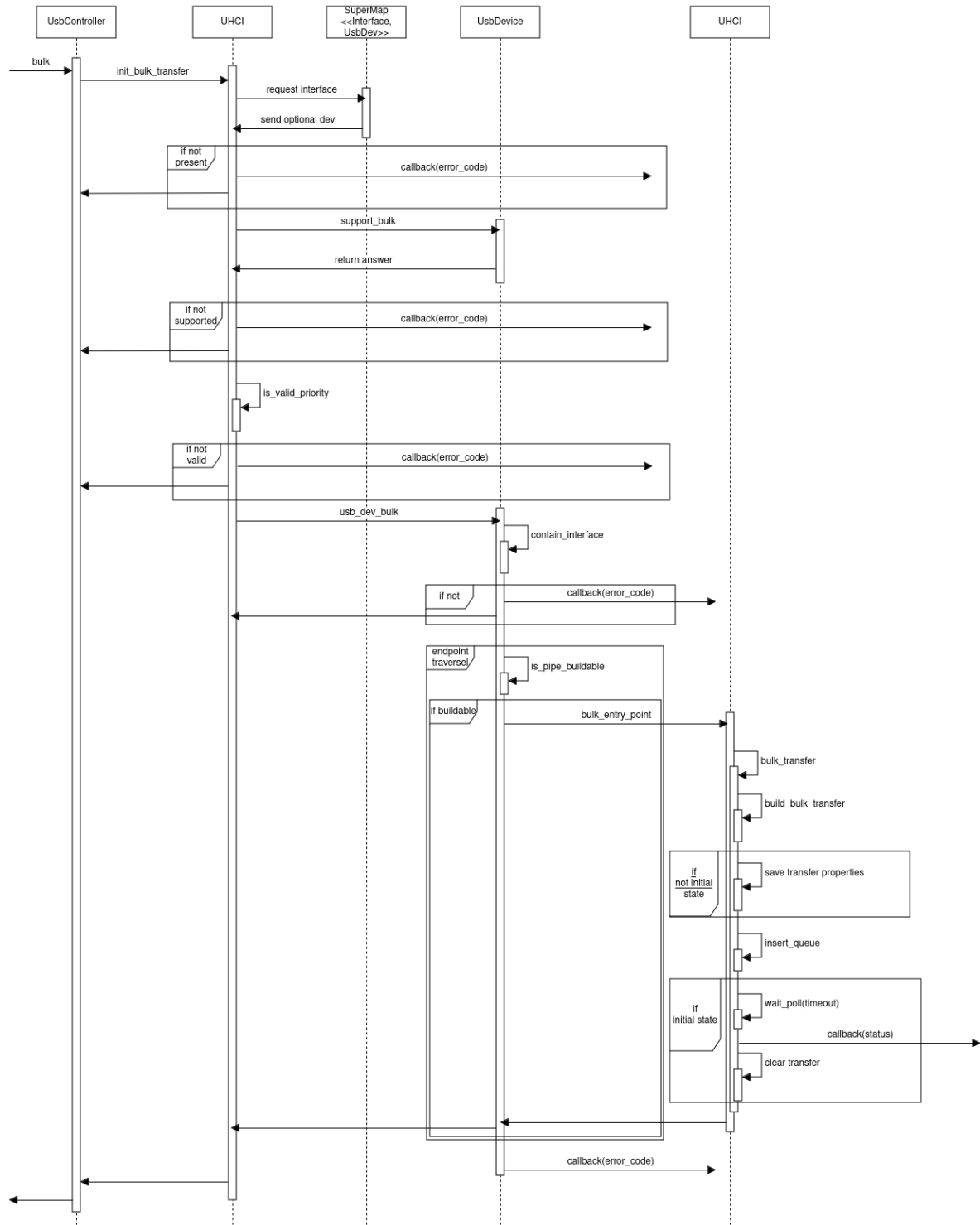


Figure 3.10: Bulk Transfer Initialization Process

The majority of the procedural steps involved in this process mirror those elucidated in the preceding explanation concerning interrupt transfers. Consequently, the examination commences at the final interaction, assuming the existence of a buildable bulk endpoint.

The procedural steps involved in the bulk transfer initiation process are as follows:

1. **UsbDevice:** Initiates communication by dispatching a message to **UHCI**.
2. **UHCI:** Upon receipt of the request from **UsbDevice**, **UHCI** engages in processing the request. Diverging from the interrupt transfer scenario, this process entails two distinct states, each necessitating the invocation of different sets of functions. Subsequent to processing the request within the pertinent state, **UHCI** concludes its operations and returns accordingly.

The forthcoming sequence diagram illustrates the orchestrated flow of messages and interactions among various system components during the initiation of a control transfer.

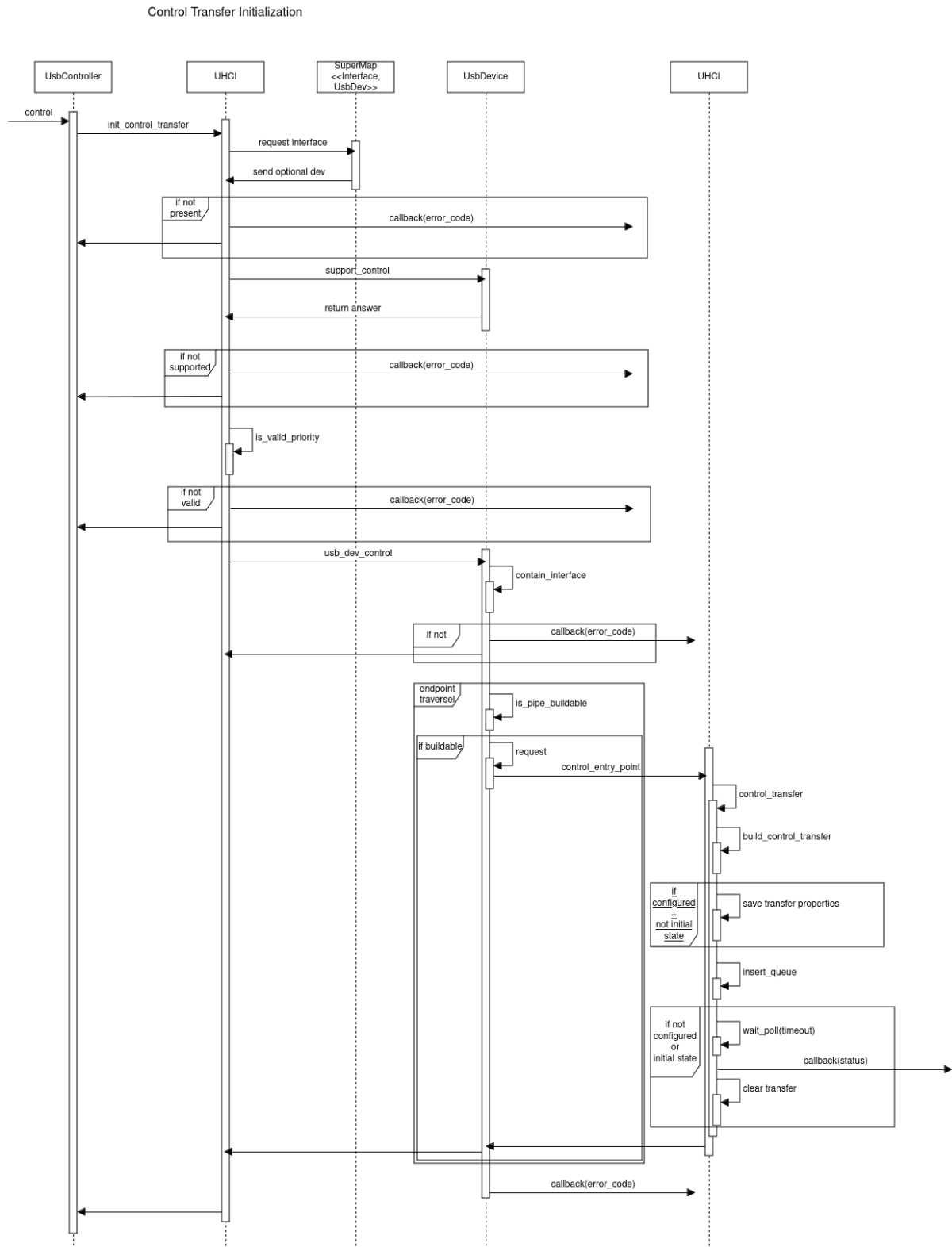


Figure 3.11: Control Transfer Initialization Process

Given the analogy with previously elucidated processes, it is observed that the concluding interaction in the control transfer procedure involves the incorporation of supplementary flags. These flags serve to designate the transition into distinct states and the subsequent invocation of corresponding functions. As comprehensive explanations of the preceding interactions have been provided, a detailed delineation of this final interaction shall be omitted.

The ensuing sequence diagram portrays the orchestrated exchange of messages and interactions among diverse system components during the registration process of a USB driver.

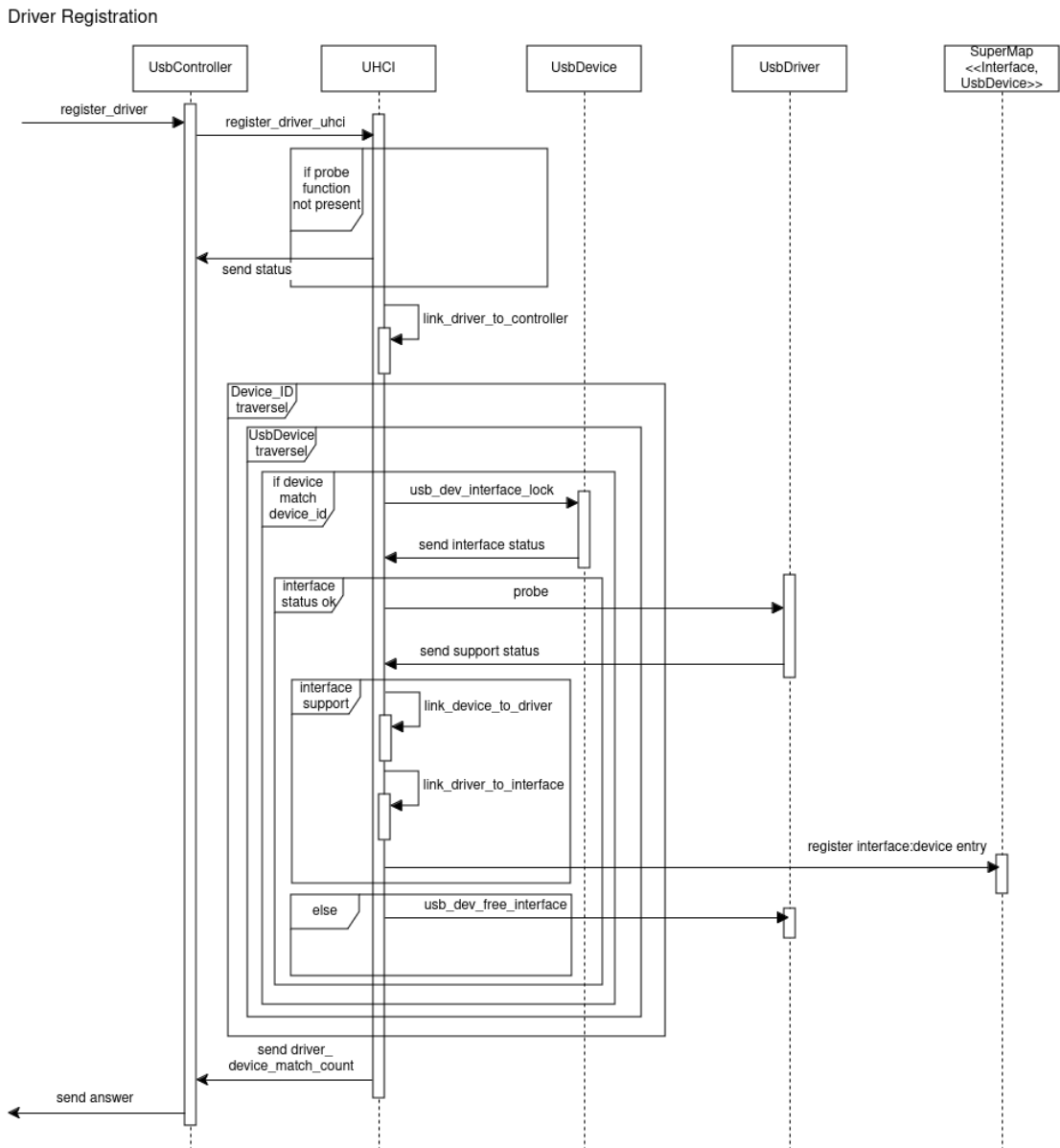


Figure 3.12: Driver Registration Process

The procedural steps involved in the registration of a USB driver are as follows:

1. **UsbController:** initially, the `UsbController` component receives a message from an external source and subsequently forwards it to the `UHCI`.

2. **UHCI:** Upon receipt of the message from `UsbController`, `UHCI` conducts a check to ascertain the presence of a probing function. If such a function is not found, an error response is dispatched.
3. **UHCI-success-state:** In the event of a successful probing function check, `UHCI` proceeds with processing and verifies if a device matching the `Device ID` is present. Subsequently, it requests the `UsbDevice` component for the status of the interface.
4. **UsbDevice:** Upon receiving the request, `UsbDevice` processes it and responds with the interface status.
5. **UHCI-success-state:** Following receipt of the response from `UsbDevice`, `UHCI` evaluates the status. If the status is determined to be successful, `UHCI` invokes the probing function associated with the `UsbDriver`.
6. **UsbDriver:** Upon receiving the message from `UHCI`, `UsbDriver` undertakes the processing of the request and provides a status code indicating whether the driver supports the interface.
7. **UHCI-success-state:** Subsequently, `UHCI` receives the status from `UsbDriver`. If the interface is supported, a registration request is initiated; otherwise, the interface is freed.
8. **SuperMap:** Upon receipt of the message from `UHCI`, `SuperMap` registers the entry and promptly returns.
9. **UHCI-success-state:** `UHCI`, having successfully registered the entry, responds with a hit count.
10. **UsbController:** Finally, the `UsbController` forwards the response accordingly.

The forthcoming sequence diagram illustrates the orchestrated exchange of messages and interactions among diverse system components during the process of registering an event listener or an event callback.

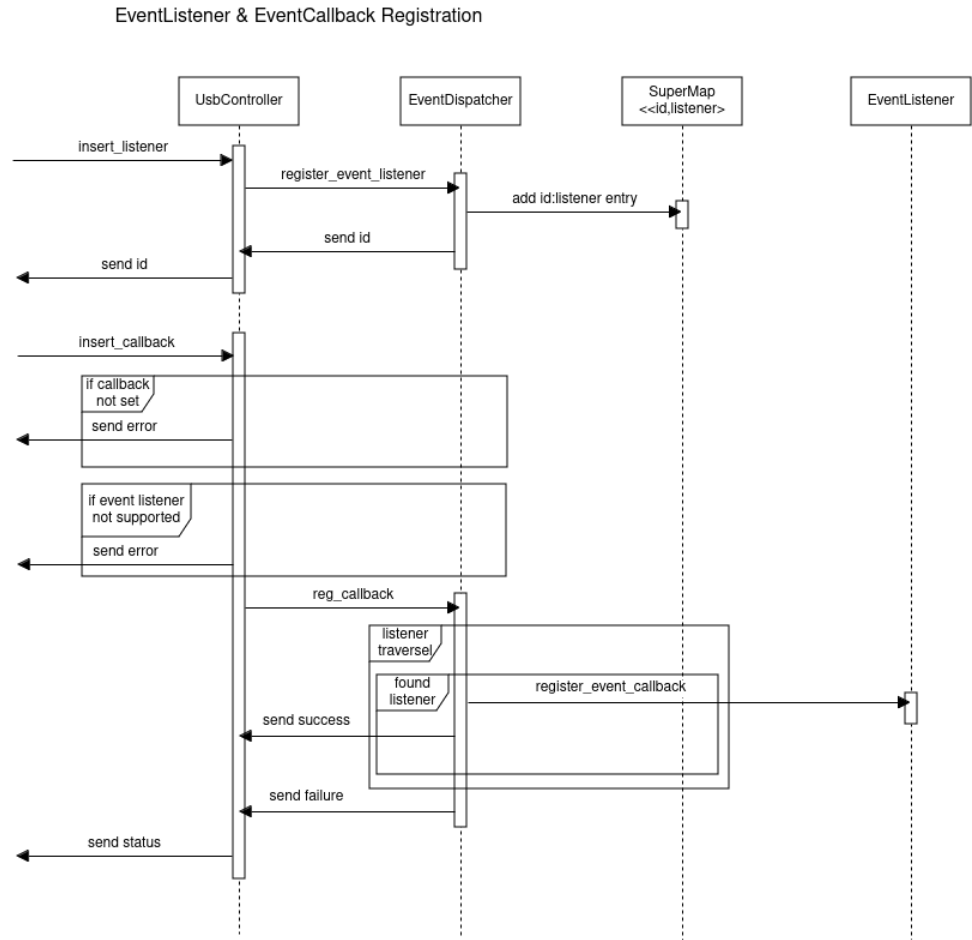


Figure 3.13: Event Registration Process

EventListener Registration:

1. **UsbController:** Initially, the `UsbController` component receives a message from an external source and subsequently requests the dispatcher for event handling.
2. **EventDispatcher:** Upon receipt of the message from `UsbController`, the `EventDispatcher` component processes the request and requests the insertion of the listener into the system.
3. **SuperMap:** Upon receipt of the message from `EventDispatcher`, the `SuperMap` component registers the listener and promptly returns the response.
4. **EventDispatcher** Having completed the registration process, the `EventDispatcher` sends the identifier associated with the listener as a response.
5. **UsbController:** Finally, the `UsbController` forwards the response accordingly.

EventCallback Registration:

1. **UsbController:** Initially, upon receiving a message from an external source, the `UsbController` component conducts validation of the request. If the validation

process detects errors, an error response is dispatched; otherwise, the controller proceeds to request the registration of the callback.

2. **EventDispatcher:** Upon receipt of the request from `UsbController`, the `EventDispatcher` component verifies if the callback can be associated with a listener. If a suitable listener is identified, the dispatcher initiates the registration process for the event callback.
3. **EventListener:** Subsequently, upon receiving the request from `EventDispatcher`, the `EventListener` processes the request in accordance with the registration specifications and returns the appropriate response.
4. **EventDispatcher:** Upon completion of the callback registration process, the `EventDispatcher` sends a response indicating success. In cases where the callback registration is unsuccessful, an error response is dispatched.
5. **UsbController:** Finally, the `UsbController` forwards the response accordingly.

The forthcoming sequence diagram illustrates the orchestrated exchange of messages and interactions among diverse system components during the process of handling an event.

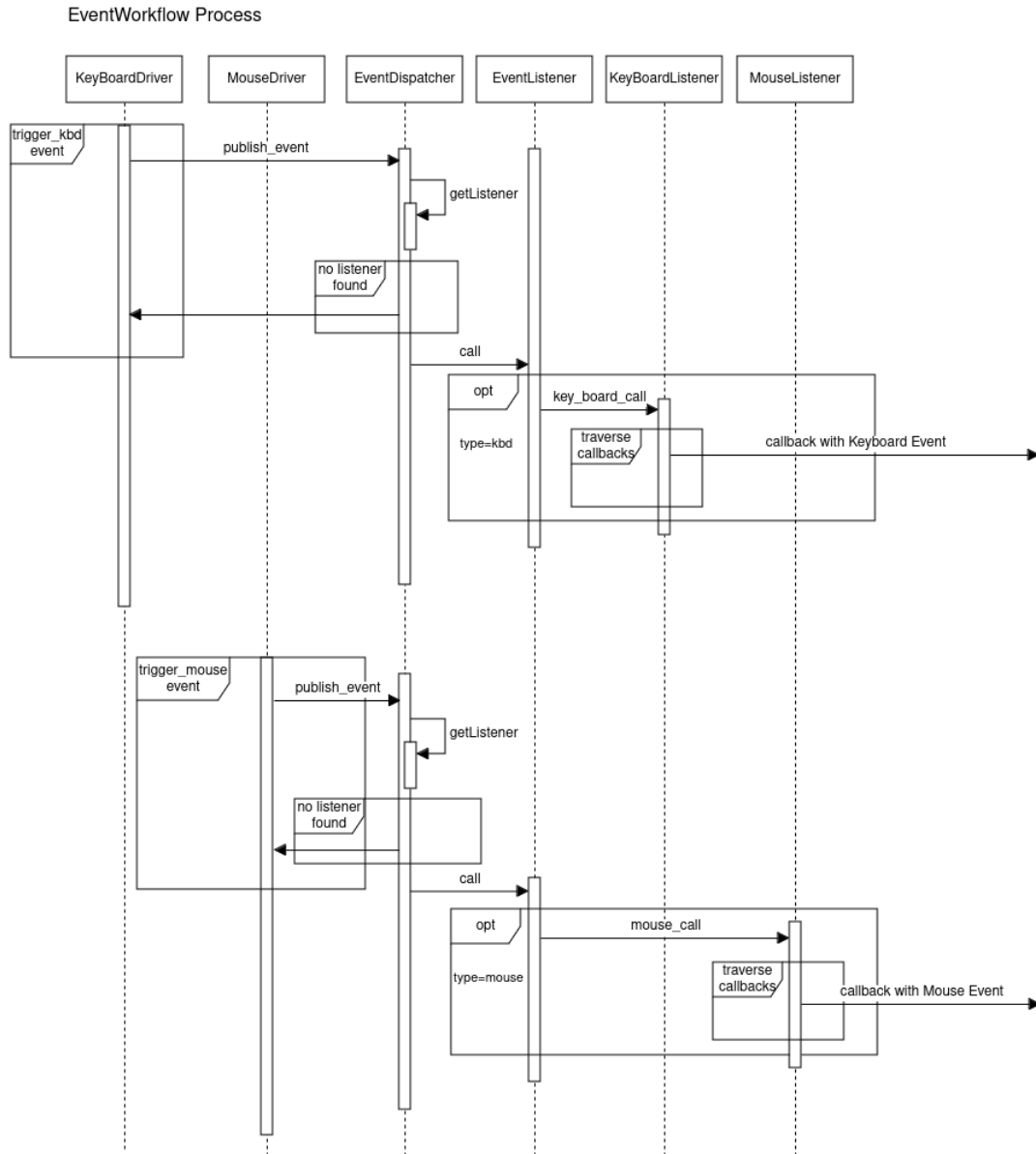


Figure 3.14: Event Handling

KeyBoardDriver EventHandler

1. **KeyBoardDriver:** Initially, upon receiving a message from an external source, the `KeyBoardDriver` component requests the `EventDispatcher` to publish an event.
2. **EventDispatcher:** Upon receipt of the message from `KeyBoardDriver`, the `EventDispatcher` validates the presence of a listener for the specified event. If no listener is found, an error response is dispatched. Otherwise, the dispatcher requests the invocation of the specific listener associated with the event.
3. **EventListener:** Upon receiving the message from `EventDispatcher`, the `EventListener` forwards the request to the `KeyBoardListener`.
4. **KeyBoardListener:** Finally, upon receiving the message, the `KeyBoardListener`

component proceeds to execute all registered callbacks associated with the event and returns the appropriate response.

MouseListener EventHandling

The event handling procedures for both the keyboard and mouse drivers share identical methodologies. For the purpose of illustrating this process, a demonstration will be provided below, focusing specifically on the mouse driver.

1. **MouseListener:** Upon receiving a message from an external source, the **MouseListener** component initiates a request to the **EventDispatcher**, urging the publication of an event.
2. **EventDispatcher:** Upon receipt of the message from **MouseListener**, the **EventDispatcher** conducts validation to ascertain the presence of a listener for the specified event. In the absence of a listener, an error response is issued. Conversely, if a listener is identified, the dispatcher proceeds to request the invocation of the designated listener.
3. **EventListener:** Upon receipt of the message from **EventDispatcher**, the **EventListener** forwards the request to the **MouseListener** for further processing.
4. **MouseListener:** Subsequently, upon receiving the message, the **MouseListener** executes all registered callbacks housed within the listener and returns the processed response.

The forthcoming sequence diagram delineates the orchestrated exchange of messages and interactions among diverse system components during the handling of an interrupt.

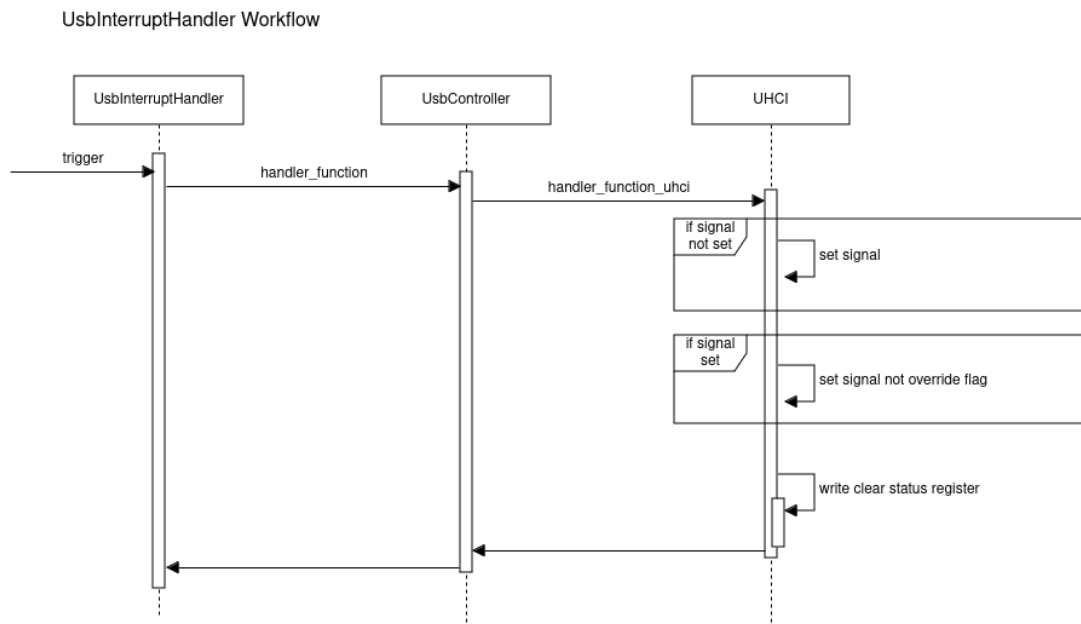


Figure 3.15: Interrupt Handling

The procedural steps involved in handling an interrupt are as follows:

1. **UsbInterruptHandler:** Initially, upon receiving a message from an external source, the `UsbInterruptHandler` component forwards the message for further processing.
2. **UsbController:** Upon receipt of the message from `UsbInterruptHandler`, the `UsbController` component further forwards the message for subsequent handling.
3. **UHCI:** Subsequently, upon receiving the message, the `UHCI` component processes the request and returns the appropriate response.

The forthcoming sequence diagram illustrates the orchestrated exchange of messages and interactions among diverse system components during the execution of the 'runnable' process.

UsbRunnable Workflow

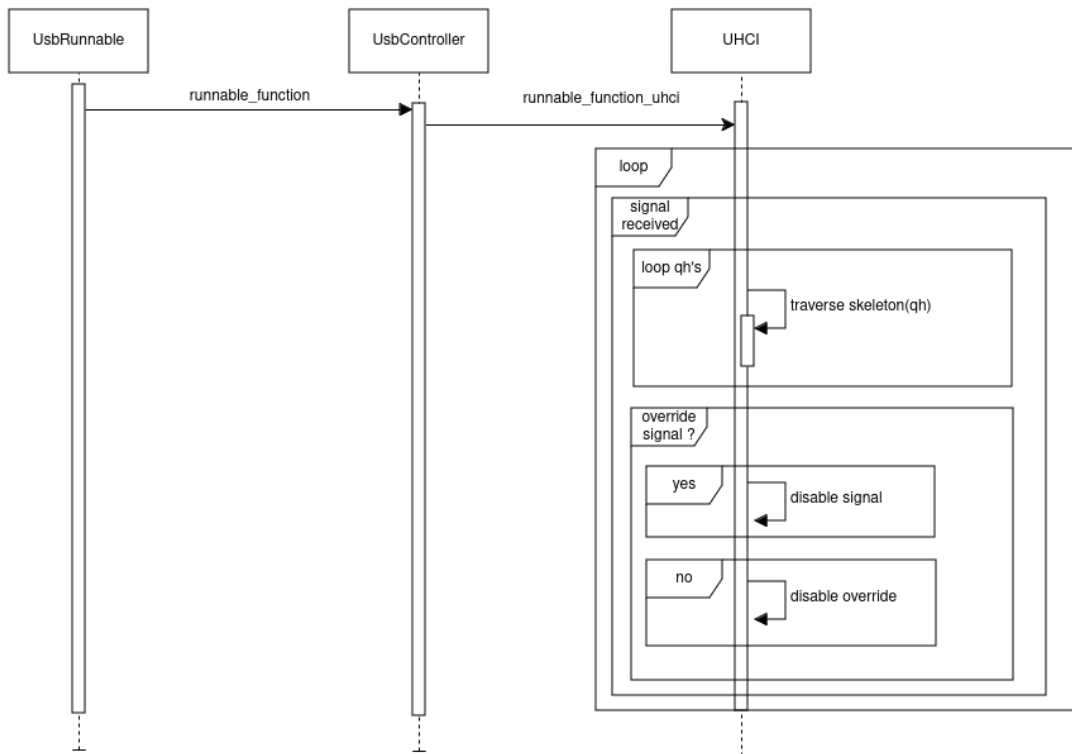


Figure 3.16: Runnable Workflow

The operational sequence entails:

1. **UsbRunnable:** Initially, upon receipt of a message from an external source, the `UsbRunnable` component proceeds to forward the request.
2. **UsbController:** Upon receiving the forwarded message from `UsbRunnable`, the `UsbController` further propagates the message.
3. **UHCI:** Subsequently, upon reception of the message, the `UHCI` component processes the request and continues its iterative operation, persisting until the system is shut down.

The ensuing sequence diagram illustrates the orchestrated exchange of messages and interactions among diverse system components, providing insight into the mechanism of probing within the system architecture.

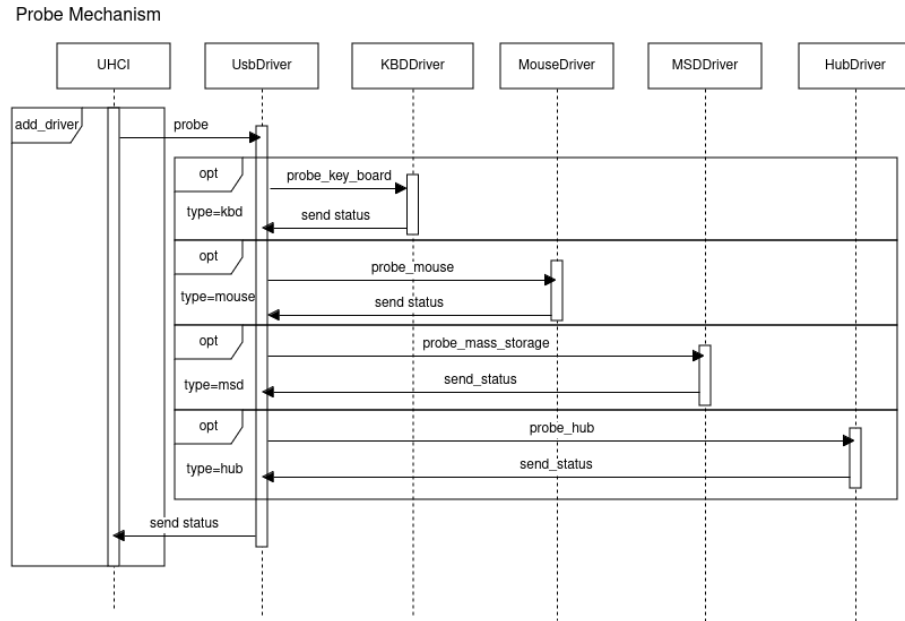


Figure 3.17: Probe Mechanism

The procedural steps involved in the mechanism of probing within the system architecture are detailed as follows:

1. **UHCI:** Currently within the context of the add driver function, the **UHCI** component initiates communication by sending a message to the driver.
2. **case-kbd, KBDriver:** Upon receiving the message from **UHCI**, the **KBDriver** within the case-kbd context processes the request and formulates an appropriate response, ensuring alignment with keyboard driver functionalities.
3. **case-mouse, MouseDriver:** Similarly, within the case-mouse context, upon receipt of the message from **UHCI**, the **MouseDriver** component processes the request and generates a response tailored to mouse driver functionalities.
4. **case-msd, MSDDriver:** In the case-msd context, upon receiving the message from **UHCI**, the **MSDDriver** component undertakes processing of the request and formulates a response coherent with mass storage device driver functionalities.
5. **case-hub, HubDriver:** Analogously, within the case-hub context, upon reception of the message from **UHCI**, the **HubDriver** component processes the request and generates a response aligned with hub device driver functionalities.

The forthcoming sequence diagram delineates the orchestrated exchange of messages and interactions among various system components, providing an illustrative depiction of the mechanism underlying callbacks within the system architecture.

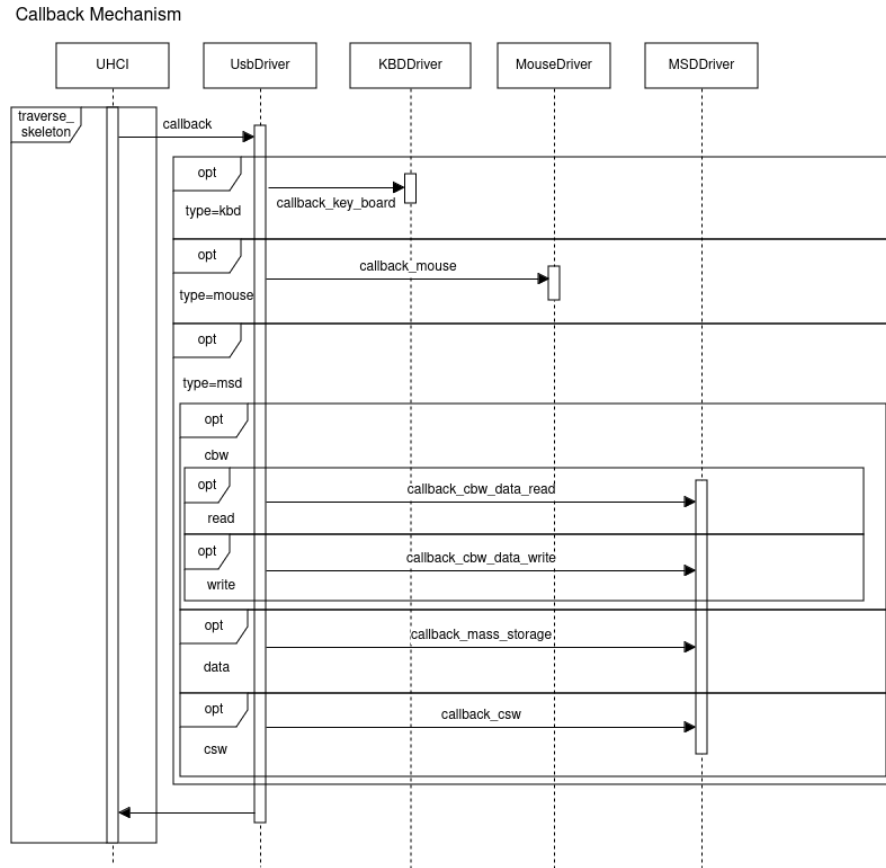


Figure 3.18: Callback Mechanism

The operational steps within the mechanism of callbacks are outlined as follows:

1. **UHCI:** Presently engaged within the traverse skeleton function, the **UHCI** component initiates communication by dispatching a message to the **UsbDriver**.
2. **UsbDriver** Upon receiving the message from **UHCI**, the **UsbDriver** component undertakes the task of forwarding the message to the appropriate driver based on the nature of the request.
3. **case-kbd, KBDDriver:** Within the context of the case-kbd scenario, upon reception of the message, the **KBDDriver** processes the request and subsequently returns the appropriate response.
4. **case-mouse, MouseDriver:** Analogously, in the case-mouse context, upon receipt of the message, the **MouseDriver** component processes the request and returns the corresponding response.
5. **case-msd, MSDDriver:** Similarly, within the case-msd context, upon reception of the message, the **MSDDriver** component processes the request and returns the relevant response.

In the context of the callback mechanism pertaining to the **MSDDriver**, it is noteworthy to highlight that the driver itself encompasses four distinct callbacks, each serving a specific

purpose and invoked accordingly. It is essential to grasp that when a callback function is invoked from the `UHCI`, these callbacks reside within one of these drivers and are invoked accordingly.

Transitioning from the examination of interactions among components within the core subsystem, attention shifts to the driver subsystem. Here, an analysis delves into how the components within this subsystem interact.

Commencing with the initial diagram, it delineates the interactions among components during the process of reading from or writing to the mass storage device.

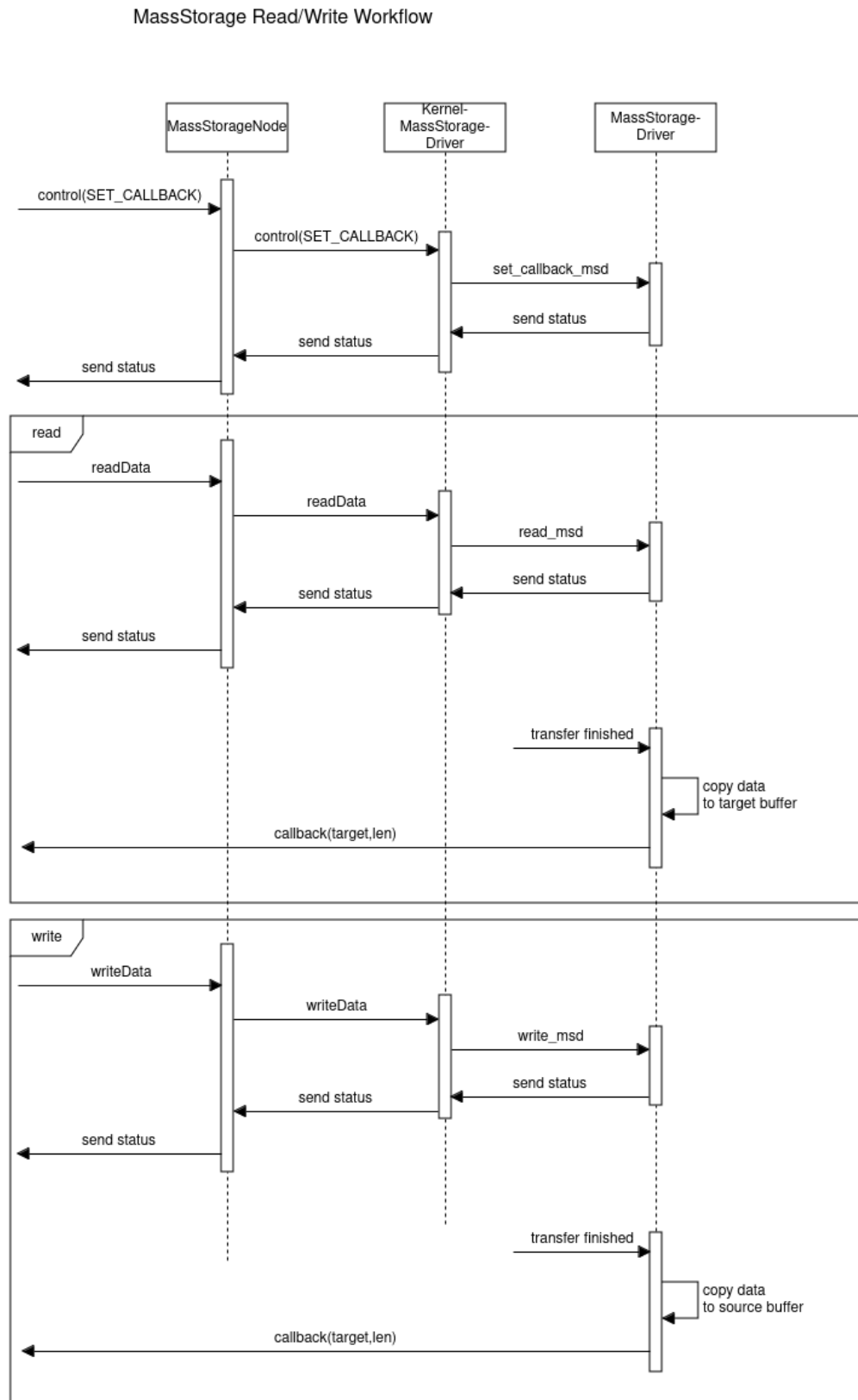


Figure 3.19: Mass Storage Device Read/Write

The procedural steps involved in setting a callback are delineated as follows:

1. **MassStorageNode:** Upon receiving a request to set a callback, the `MassStorageNode` component forwards this request to the `KernelMassStorageDriver`.
2. **KernelMassStorageDriver:** Upon receipt of the request from `MassStorageNode`, the `KernelMassStorageDriver` initiates the process of setting the callback.
3. **MassStorageDriver:** Following the request, the `MassStorageDriver` component processes the received request and formulates a response conveying the status of the operation.
4. **KernelMassStorageDriver:** Upon receiving the status response from `MassStorageDriver`, the `KernelMassStorageDriver` forwards this response.
5. **MassStorageNode:** Lastly, the `MassStorageNode` forwards the status response received from the `KernelMassStorageDriver`.

The subsequent delineation bifurcates the process into read and write calls, initiating with the read call. These interactions are paramount in enabling the smooth execution of read/write operations.

MSD-Read

1. **MassStorageNode:** Receives request to read data and forwards request.
2. **KernelMassStorageDriver:** Upon receipt of the read request, the `KernelMassStorageDriver` initiates the data retrieval process by requesting the `MassStorageDriver` to read from the mass storage device.
3. **MassStorageDriver:** Upon reception of the read request, the `MassStorageDriver` component processes the request and responds with the status, indicating the success of the operation if the build process was successful.
4. **MassStorageDriver-delay-x:** Following a predefined delay, denoted as 'x', the transfer operation concludes, with data successfully copied. Subsequently, the callback function, which was previously set, is invoked to handle further processing.

MSD-Write

This operation closely resembles the read call in both its functionality and procedural aspects, thereby serving as a pedagogical tool to aid in comprehension.

Following this, the interactions among system components involved in the retrieval of keyboard input will be elucidated.

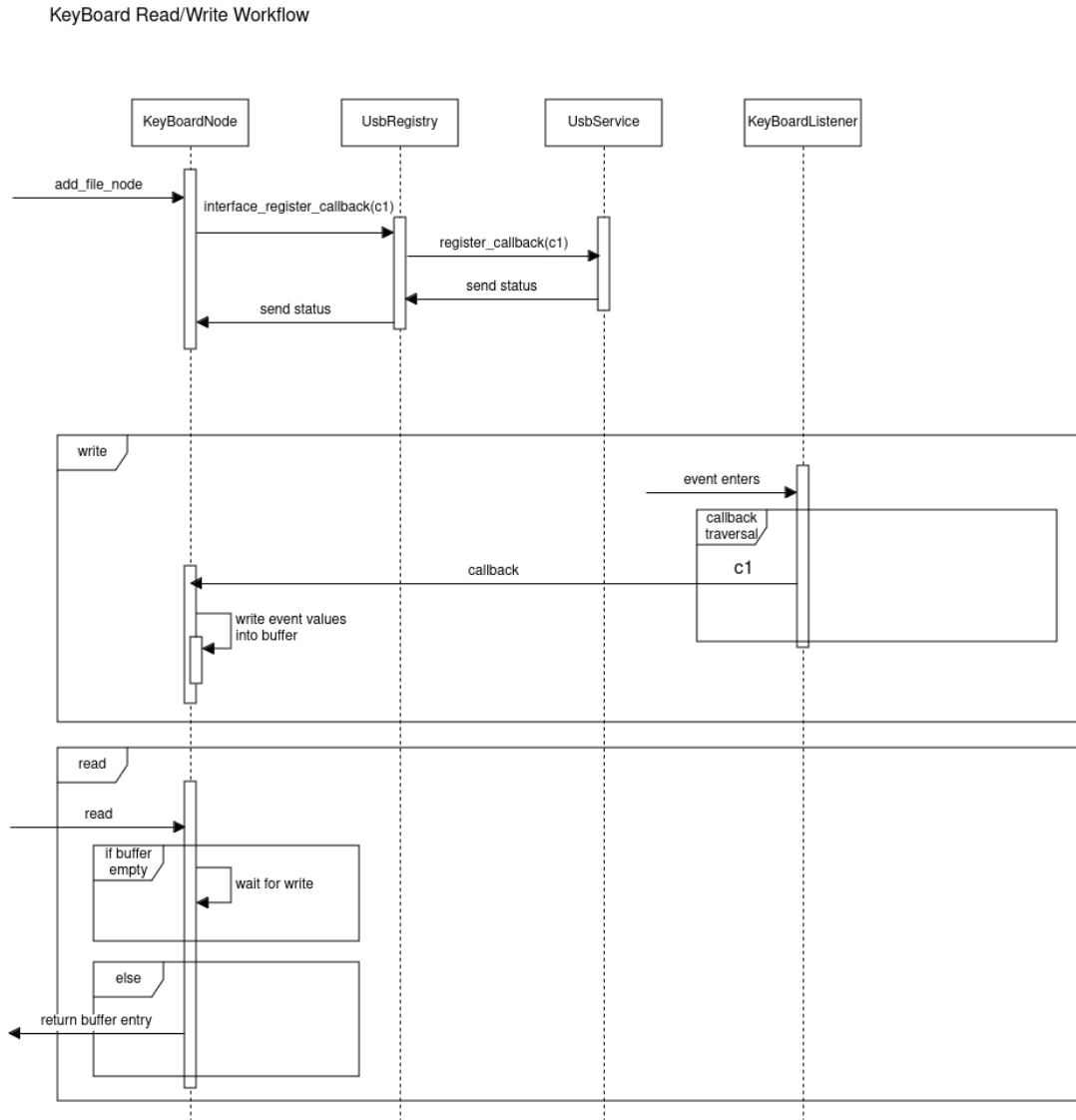


Figure 3.20: Keyboard Read

The procedural steps involved in the interaction between components during the keyboard input retrieval process are outlined as follows:

1. **KeyBoardNode:** Upon receipt of a message to add a node, the `KeyBoardNode` component initiates a request to register a callback.
2. **UsbRegistry:** Upon receiving the request from `KeyBoardNode`, the `UsbRegistry` forwards this request to the `UsbService` for further processing.
3. **UsbService:** Following reception of the request, the `UsbService` component undertakes processing and formulates a response indicating the success or failure of the operation.
4. **UsbRegistry:** Upon receipt of the response from `UsbService`, the `UsbRegistry` forwards this response accordingly.

Divergence into distinct read and write operations ensues. It is critical to understand that the write call is not externally prompted but rather initiated internally by a driver in response to an event occurrence.

With this clarification, the examination of the write call's functionality follows. Emphasis is placed on the prerequisite interactions detailed previously, essential for facilitating the seamless execution of this call.

1. **KeyboardListener:** Upon receipt of a message indicating the occurrence of an event, denoted as 'event entered', the `KeyboardListener` component invokes the callback function, denoted as 'c1', which was previously registered. This callback serves to handle the event processing.
2. **KeyboardNode:** Upon receiving the message from `KeyboardListener`, the `KeyboardNode` component proceeds to write the event data to the buffer and subsequently returns.

Subsequent to this, an elucidation of the functioning of the read process shall ensue.

1. **KeyboardNode:** Upon receiving a request to read, the system undertakes an assessment to ascertain the presence of an event. Subsequently, if an event is detected, the system responds by transmitting the event information.

Subsequently, an analysis of the initialization procedure of the hub driver will be conducted, focusing on the interaction mechanisms involved.

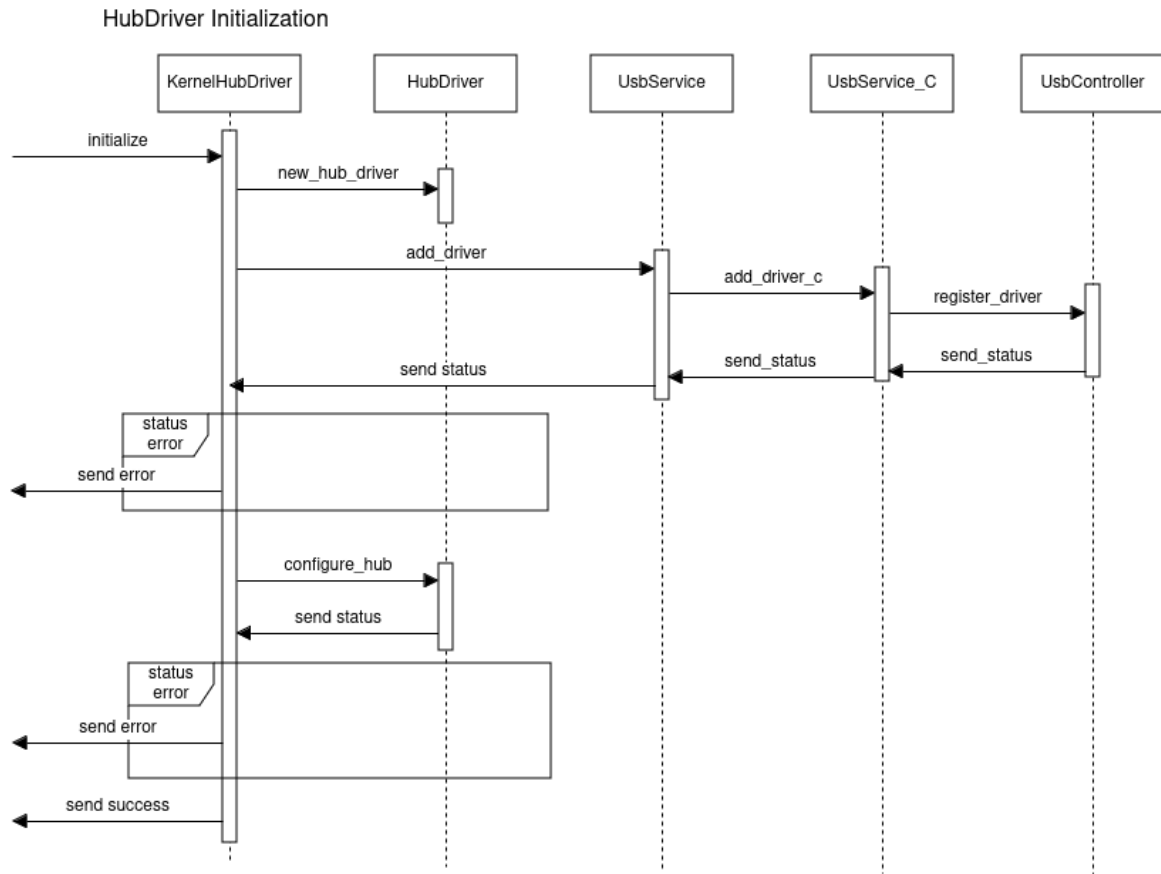


Figure 3.21: Hub Driver Initialization

1. **KernelHubDriver:** Upon reception of an external request, initiates the procedure for generating a new hub driver.
2. **HubDriver:** Upon receipt of the request, executes the requisite processing tasks and returns.
3. **KernelHubDriver:** Initiates a request to include a hub driver.
4. **UsbService:** Upon receiving the request, forwards it to the **UsbService_C** component.
5. **UsbService_C:** Upon reception of the request, proceeds with the registration of the hub driver.
6. **UsbController:** Upon receipt of the request, carries out processing tasks and issues a response indicating the status.
7. **UsbService_C:** Forwards the response to the subsequent component.
8. **UsbService:** Propagates the response to the corresponding entity.
9. **KernelHubDriver:** Inspects the response for any errors; in case of an error, transmits an error response; otherwise, proceeds to request hub configuration.

10. **HubDriver:** Upon receipt of the request for hub configuration, executes the necessary operations and issues a response containing the status.
11. **KernelHubDriver:** Conveys the status response as appropriate.

Following this, the initialization process of the keyboard driver will be explored, shedding light on the interaction dynamics involved.

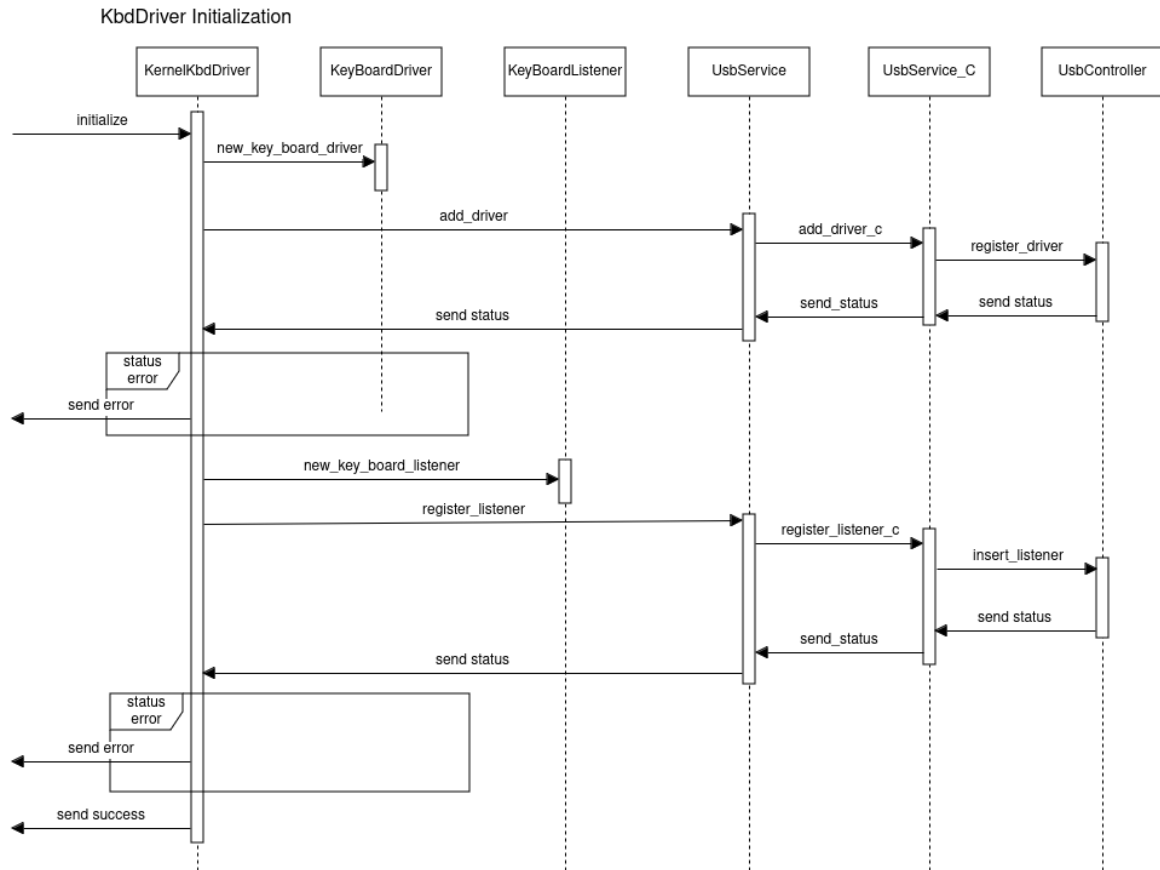


Figure 3.22: Keyboard Driver Initialization

1. **KernelKbdDriver:** Upon reception of an external request, initiates the procedure to generate a new keyboard driver.
2. **KeyBoardDriver:** Processes the received request and returns.
3. **KernelKbdDriver:** Dispatches a request to add the keyboard driver.
4. **UsbService:** Upon receiving the request from `KernelKbdDriver`, forwards it to the `UsbService_C` component.
5. **UsbService_C:** Upon receipt of the request from `UsbService`, proceeds with the registration of the driver.
6. **UsbController:** Upon receiving the message, executes processing tasks and responds with a status indication.

7. **UsbService_C**: Forwards the response to the subsequent component.
8. **UsbService**: Propagates the response to the corresponding entity.
9. **KernelKbdDriver**: Checks the response for any errors; in case of an error, transmits an error response; otherwise, requests a new listener.
10. **KeyBoardListener**: Upon receipt of the message, creates a listener and returns.
11. **KernelKbdDriver**: Requests the registration of the listener.
12. **UsbService**: Forwards the request to the relevant component.
13. **UsbService_C**: Upon receiving the message, proceeds to insert the listener.
14. **UsbController**: Processes the message, generates a status response, and transmits it back to **KernelKbdDriver**.
15. **KernelKbdDriver**: Transmits the status response accordingly.

Subsequently, the initialization process of the mass storage driver will be scrutinized, focusing specifically on the distinct interactions involved, particularly those pertaining to the configuration request of the **MSDDriver**.

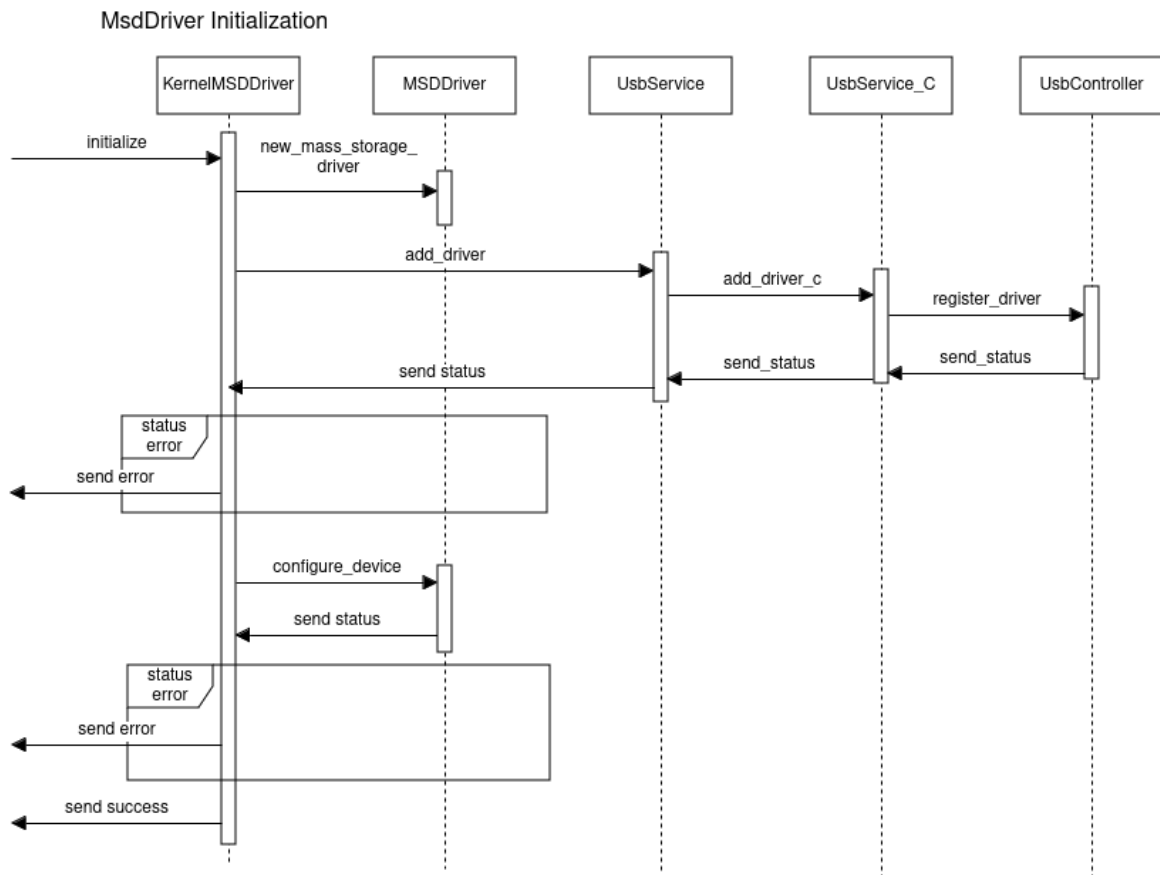


Figure 3.23: Mass Storage Driver Initialization

1. **KernelMSDDriver:** Initiates a request to configure **MSDDriver**.
2. **MSDDriver:** Upon receiving the message from **KernelMSDDriver**, processes the request and responds with a status indication.
3. **KernelMSDDriver:** Receives the status response from **MSDDriver** and handles it accordingly.

Finally, the process of node creation will be examined, as depicted in the diagram below.

Usb Device Node Creation Workflow (KeyBoard Node)

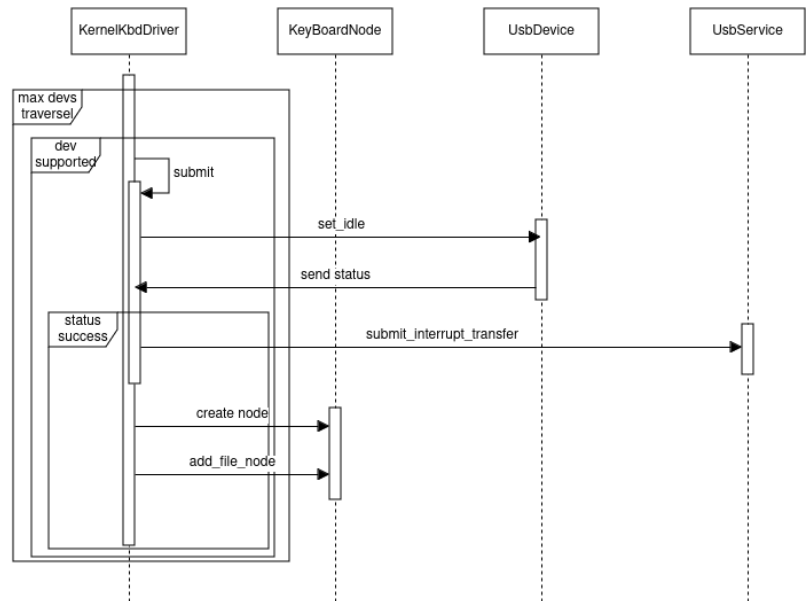


Figure 3.24: USB Device Node Creation

1. **KernelKbdDriver:** The **KernelKbdDriver** component initiates its operations by verifying its current state and the presence of any connected devices. Upon confirming device connectivity, it proceeds to dispatch a set idle request. Subsequently, it iterates through all accessible devices within its purview.
2. **UsbDevice:** Responsible for receiving incoming messages, executing pertinent operations as per the received requests, and subsequently relaying the status of the operation.
3. **KernelKbdDriver:** Following the completion of device status verification, this component proceeds to ascertain the success of the operation. In the event of a successful outcome, it initiates a request for the commencement of an interrupt transfer process.
4. **UsbService:** Tasked with the reception, processing, and subsequent response dissemination of incoming messages.
5. **KernelMSDDriver:** Upon successful validation of the status, this component proceeds to facilitate the creation of a device node.

6. **KeyboardNode:** This component receives requests, performs the necessary actions to generate a device node, and subsequently returns the outcome of the operation.
7. **KernelKbdDriver:** Finally, this component initiates a request for the inclusion of the device node within the file system, thereby ensuring its visibility.
8. **KeyboardNode:** Upon receiving a request, this component incorporates the device node into the file system as specified and communicates the result of the operation.

3.1.7 Functional Analysis of Core System Components

Following a comprehensive examination of the interplay among the constituent elements within each system and an understanding of their architectural compositions, the ensuing focus shifts towards a more nuanced exploration of the internal mechanisms inherent to select components. This endeavor seeks to facilitate a deeper comprehension of the intricate operational frameworks underpinning these entities.

Commencing with an examination of the **UHCI**, the initial scrutiny is directed towards its configuration functionality.

Controller Configuration Internal Logic

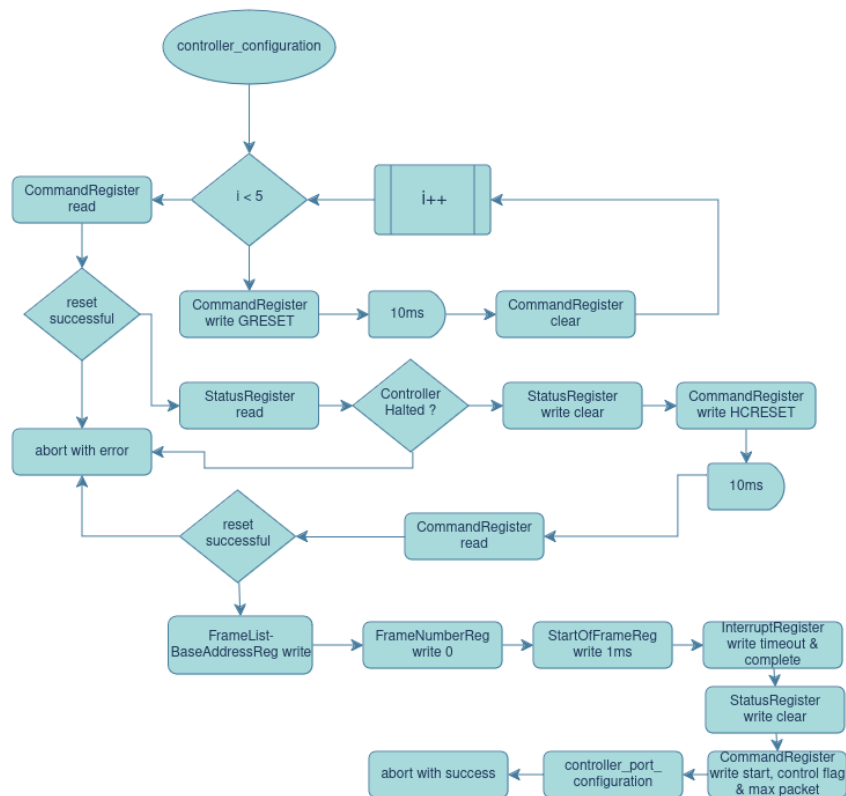


Figure 3.25: Controller Configuration

The configuration procedure commences with the elevation of the `GRESET` bit to a high state for a duration of 10ms, succeeded by the complete clearance of the contents within the command registers. This sequence of resetting operations is iterated fivefold to achieve the requisite 50ms delay. Following the execution of global resets, a scrutiny of the command register ensues, wherein the presence of a non-zero content prompts the immediate cessation of the configuration process. Subsequently, an interrogation of the status register is conducted to ascertain the operational status of the controller, thereby preempting any further progression should the controller be halted. In the event that the controller remains operational and unhalted, the complete purging of the status register content is effected through write-clearing operations. Proceeding henceforth, the `HCRESET` bit is inscribed within the command register, instigating a hardware reset. A 10ms delay ensues, during which the controller autonomously clears the aforementioned bit. Subsequent scrutiny of the command register confirms the successful clearance of the `HCRESET` bit, thereby affirming the efficacy of the reset operation. With the reset deemed successful, the actual configuration process commences with the provisioning of the frame list's base address, obtained through the formulation of the scheduling algorithm, to the frame base address register. Sequentially, a zero value is inscribed within the frame number register, denoting the initialization of frame enumeration from the initial frame. Ensuing, the value 0x40 is designated to the start of frame register, ensuring each frame's temporal span is precisely 1ms. Consequent to the assignment of interrupt parameters, including the setting of the `TMOUT_CRC` and `COMPLETE` bits within the interrupt register to facilitate signaling in instances of successful transfers or timeout occurrences, a comprehensive erasure of the status register content is once again enacted. Subsequent to this, the activation of the run bit, max packet bit, and control flag within the configuration is executed. A subsequent invocation of an auxiliary function ensues to effectuate the reset of UHCI ports, which will be elucidated upon subsequently. Finally, the process concludes with an affirmative termination status.

Progressing to the subsequent functionality within the purview of the UHCI driver, attention is directed towards the port reset mechanism.

UHCI Reset Port Internal Logic

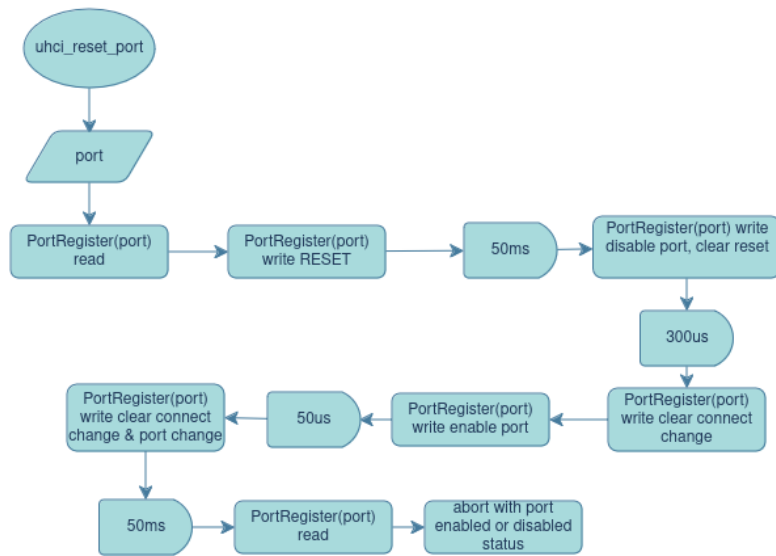


Figure 3.26: Controller Port Reset

The port reset function initiates with the retrieval of the pertinent register associated with the specified port parameter. Following this acquisition, the reset bit is set, thereby signaling the commencement of the reset operation. A prescribed delay of 50ms ensues before the reset bit is cleared. Subsequent to this temporal interval, the reset bit is deactivated, concomitantly with the disabling of the port via the inscription of a zero value. A further delay of 300 μ s is observed before progression. Subsequently, the connect change bit is subjected to a write-clear operation to ensure its deactivation, thereby facilitating the subsequent port enablement. Following the enablement of the port, an additional delay of 50 μ s is observed prior to the execution of a final write-clear operation targeting any lingering change bits. The process culminates with a final delay of 50ms, subsequent to which the port register is scrutinized to discern the current enablement status of the port, the outcome of which is subsequently returned.

Subsequently, attention will be directed towards elucidating the process by which the schedule is constructed.

UHCI Request Frames Internal Logic

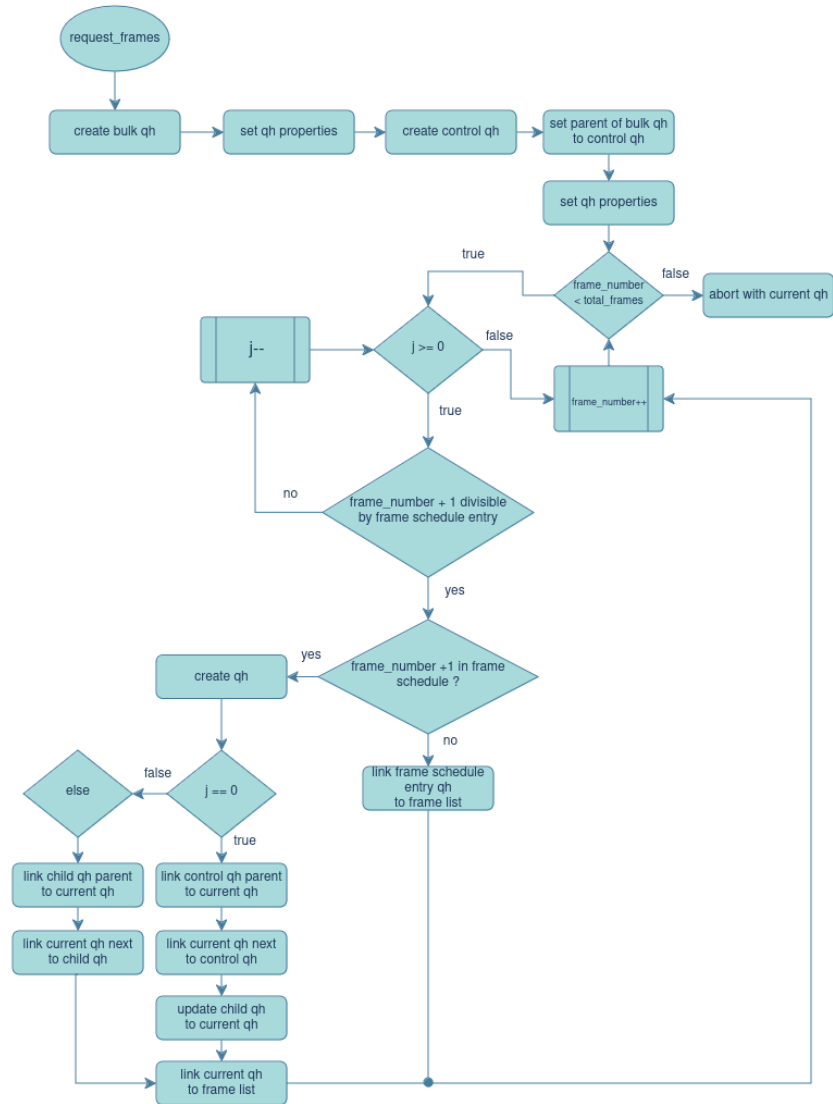


Figure 3.27: Schedule Construction

The process of schedule construction within the **UHCI** entails several sequential steps. Initially, the creation of the bulk QH is undertaken, wherein all pertinent properties are configured. Subsequently, the control QH is instantiated, with its properties set in accordance with the established specifications. These entities are then interlinked, with the parent pointer of the bulk QH referencing the control QH.

Following the establishment of the non periodic schedules, focus shifts towards the construction of the interrupt schedule, which is further subdivided into its constituent sub-schedules. This endeavor involves an iterative traversal across all frames. Within each iteration, a determination is made as to whether the frame number incremented by one is divisible by the frame schedule entry at index j . Should this condition hold true, a subsequent evaluation ensues to ascertain if the incremented frame number corresponds to an entry within the schedule. In instances where the incremented frame number aligns with an entry in the schedule, an interrupt QH is instantiated.

Should the incremented frame number indeed be the initial entry within the schedule (denoted by $j == 0$), the newly instantiated QH is linked to the control QH, subsequently updating the child QH to reflect the current QH. Conversely, if the incremented frame number does not represent the first entry within the schedule (denoted by $j \neq 0$), the current QH is linked to its preceding QH in the schedule, updating the child QH accordingly.

Both frame schedule entries are then linked to the frame list. In instances where the incremented frame number does not correspond to an entry within the schedule, the frame entry within the frame list for the given frame number directs to the nearest frame schedule entry at index j .

Upon the completion of frame traversal, the function returns the last QH generated, thereby furnishing a foundational framework within which QH insertion and removal operations can be effectuated.

Subsequently, an examination will be conducted to elucidate the integration of QHs into the schedule.

UHCI Queue Insertion Internal Logic

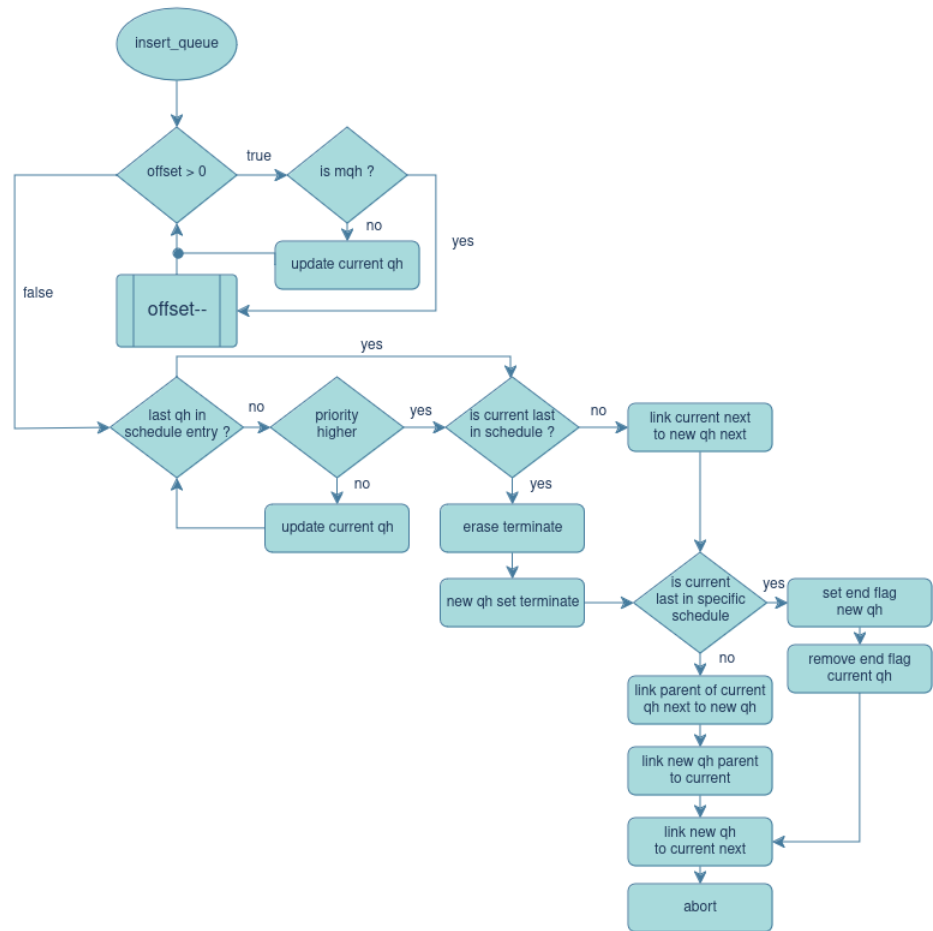


Figure 3.28: Queue Insertion

The process of queue insertion within the **UHCI** commences with the identification of the appropriate schedule entry to accommodate the QH corresponding to the intended transfer. Upon locating the designated QH within the framework, the procedure entails an iterative traversal through all registered QHs within the specified schedule. Termination of the loop is contingent upon encountering either the current QH being the last within the schedule or the priority of the QH to be inserted surpassing that of the current QH.

Subsequently, an assessment is made to determine if the current QH represents the final QH within the entirety of the schedule. Should this condition hold true, the termination bit within the current QH is cleared, while the new QH slated for insertion assumes the termination bit. Conversely, if the current QH does not denote the terminal QH within the overall schedule, indicating the presence of at least one subsequent QH, the QH's Queue Head Link Pointer (QHLP) is directed to the QHLP referenced by the current QH.

Continuing, an evaluation is conducted to ascertain whether the current QH represents the concluding QH within the specific schedule. Should this be the case, the termination flag is set within the new QH, subsequently negating this flag within the current QH, thereby signifying that the schedule's ultimate QH now corresponds to the new QH. In instances where the current QH does not denote the concluding QH within the specified schedule, the parent pointer of the current QH's QHLP is adjusted to reference the new QH. Following this adjustment, the parent of the new QH is linked to the current QH, and the QHLP of the current QH is directed towards the new QH. By adhering to this iterative process, the new QH is seamlessly integrated into the existing structure while upholding the overall integrity of the schedule.

Having gained a better understanding of how the QHs are integrated, it is pertinent to explore the inverse process, namely the removal of QHs from the schedule.

UHCI Queue Removal Internal Logic

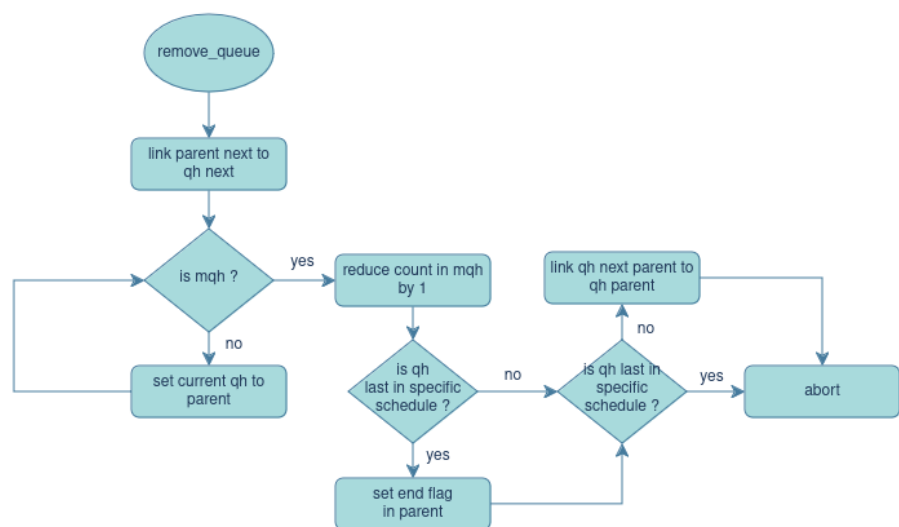


Figure 3.29: Queue Removal

The process of queue removal within the **UHCI** initiates by directing the parent of the designated Queue Head (rQH) – the QH slated for removal – to reference the rQH's

Queue Head Link Pointer (QHLP). Subsequently, an iterative traversal ensues through all registered QHs within the specified schedule until reaching the apex QH, which serves as the schedule's entry point. Upon reaching this terminal QH, the total count of QHs associated with the schedule is decremented by one.

Following the decrement of the QH count, an assessment is conducted to determine if the rQH represents the final QH within the specified schedule. Should this condition hold true, the termination flag within the parent of the rQH is appropriately set. Subsequently, a check is made to ascertain whether the rQH does not represent the terminal QH within the specific schedule. In instances where the rQH does not denote the final QH within the schedule, the parent pointer of the rQH's child QH is updated to reference the parent of the rQH.

Following the provided diagram, the focus of inquiry will now shift towards an examination of the transfer functionalities inherent to the UHCI architecture.

Commencing with an exploration of the packet creation functionality, the accompanying illustration encapsulates the conceptual framework.

Packet Creation Internal Logic

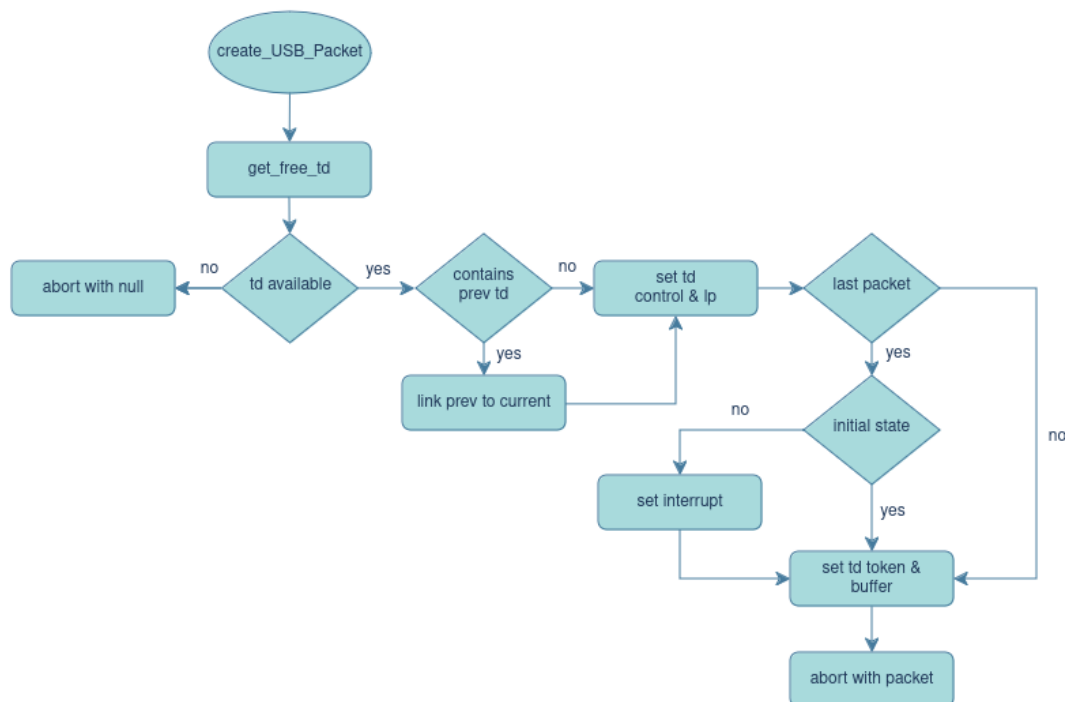


Figure 3.30: Packet Creation

The process of packet creation within the **UHCI** commences with the invocation of a subordinate function responsible for furnishing an optional TD. Should the TD not be available, the procedure is promptly terminated. Subsequently, an examination ensues to ascertain the presence of a preceding TD, passed in as an argument. In the event of its existence, a linkage is established between the preceding TD and the current TD, thereby directing the preceding TD's Link Pointer (LP) to reference the current TD.

Following this linkage, the properties of the current TD are configured in accordance with

the specified parameters. Subsequently, an assessment is made to determine whether the current invocation denotes the final iteration. Should this condition hold true, a further evaluation ensues to ascertain the necessity of transmitting an interrupt. In instances where the system is initially in a state of anticipation, primarily awaiting the conclusion of the transfer before proceeding, the transmission of an interrupt is deemed unnecessary, as the system is configured for polling. Conversely, in scenarios where the system is not in its initial state, the interrupt bit within the current TD is activated.

Following the activation of the interrupt bit, the token properties and the buffer pointer of the current TD are adjusted accordingly. The process culminates with the termination of packet construction.

Subsequently, attention will be directed towards elucidating the construction of specific transfers within the UHCI framework. Commencing with an analysis of the control transfer, the accompanying illustration serves as a visual aid to elucidate the concept.

Build Control Transfer Internal Logic

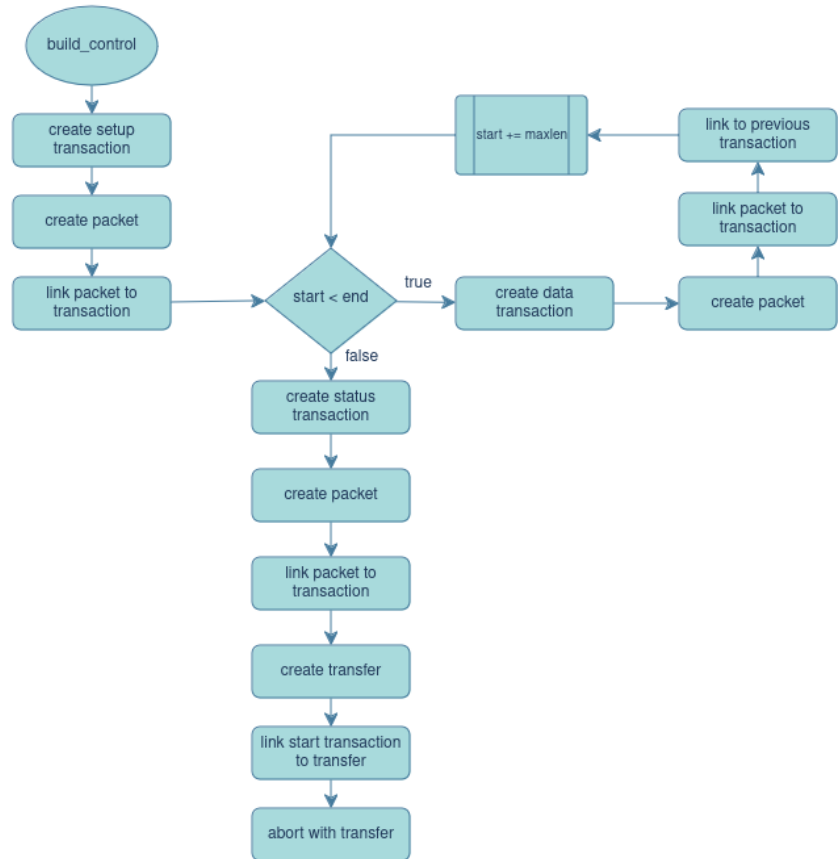


Figure 3.31: Control Transfer Build

The construction process of a control transfer within the `UHCI` initiates with the instantiation of the setup transaction, whereupon a packet is generated by invoking the functionality referenced as `packet_creation`, subsequently establishing a linkage between this packet and the transaction. Proceeding to the data stage, a data transaction is generated along with its corresponding packet, which are then interconnected. Subsequently, the transactions are linked together to form a chain, thereby delineating the entirety of the

transfer, with the start address being updated iteratively by an increment equal to the maximum length allowed for the control endpoint (denoted as *maxlen*). This iterative process continues until all data has been processed, as indicated by the equivalence of the start address and the end address. Upon reaching the conclusion of the data transmission, a final status transaction is instantiated, complete with its corresponding packet. Finally, a transfer is generated and linked to the initial transaction, culminating in the completion of the control transfer, which is subsequently returned.

Following the discussion on control transfers, the subsequent focus will be directed towards the construction of bulk and interrupt transfers.

Build Interrupt & Bulk Transfer Internal Logic

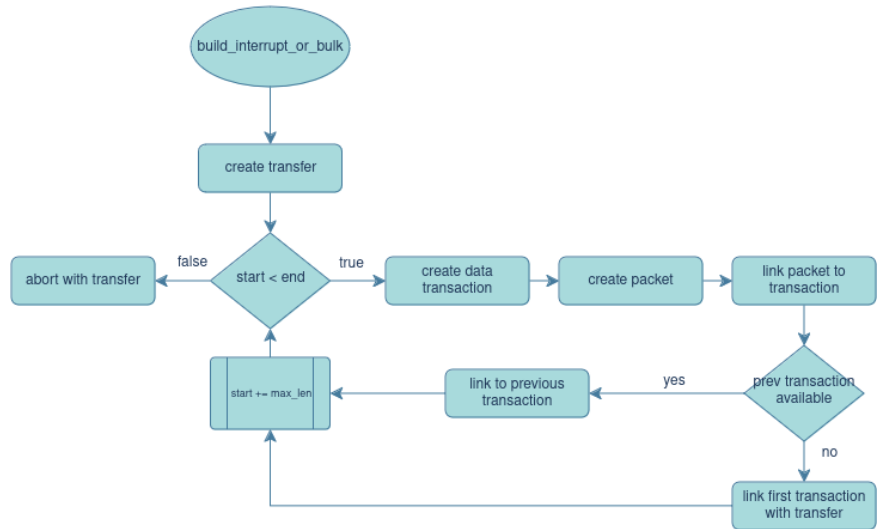


Figure 3.32: Bulk and Interrupt Transfer Build

The creation process of bulk and interrupt transfers within the **UHCI** diverges from that of control transfers, as delineated in 3.31. Notably, bulk and interrupt transfers lack a setup stage and a status stage, comprising solely a data stage. The construction procedure of data transactions mirrors that described in 3.31, wherein transactions are generated iteratively until all data has been processed, with subsequent updates made incrementally by *max_len*.

However, the linkage of these transactions differs in approach. Here, the previous transaction is stored, and the current transaction is linked to the preceding one. In the absence of a previous transaction, the first transaction is directly linked to the transfer, thus establishing an entry point in the transaction chain. Subsequent to this processing, the operation concludes with the return of the constructed transfer.

Subsequent to the elucidation of the construction process for bulk and interrupt transfers, attention will now be directed towards delineating how the **UHCI** navigates through the QHs within the schedule.

UHCI Traversal Internal Logic

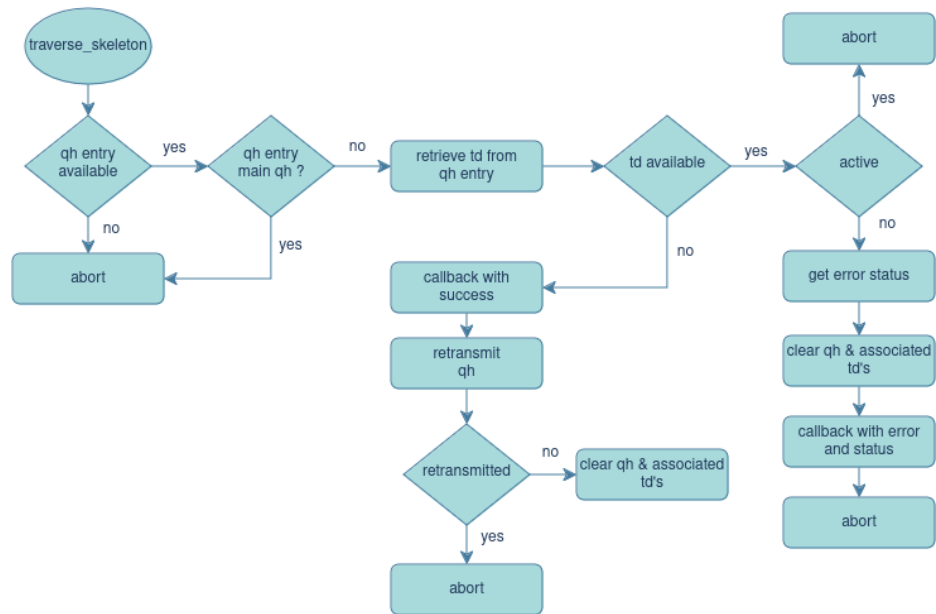


Figure 3.33: Schedule Traversal

The traversal logic implemented within the `UHCI` operates as follows: Initially, it verifies the availability of the QH passed as an argument. In the event of unavailability, the process is promptly terminated. Subsequently, a check is performed to determine if the provided QH corresponds to a Multiple Queue Head (MQH), signifying an entry point within the schedule. Should this condition be met, the operation is aborted, as the MQH does not represent an actual transfer but rather serves as a reference point within the schedule.

Upon confirming that the QH represents a valid transfer, the associated TD is retrieved. If the TD is unavailable, indicated by the QH's fields pointing to a null value, it signifies the successful completion of the transfer, prompting the initiation of processing for a successful transfer state. This entails invoking the designated callback function with a success status to signify the successful completion of the transfer. Subsequently, a check is made to determine if the QH should undergo retransmission, such as in the case of an interrupt QH intended for periodic execution. The mechanics of retransmission are discussed in detail in 3.34.

In the absence of retransmission requirements, the QH and all associated TDs are cleared. Conversely, if the TD is available within the QH, its activity status is examined to ascertain whether it remains active and necessitates further processing. Should the TD be active, signifying an ongoing process, the traversal logic awaits its completion. Conversely, if the TD is inactive, indicative of an erroneous condition, the process transitions to handling the erroneous transfer state. In this state, the error status retrieved from the control field within the TD is obtained, and both the QH and its associated TDs are cleared. Subsequently, the designated callback function is invoked with explicit error details and an error status pertaining to the erroneous transfer, after which the operation is aborted.

The final aspect of functionality within the `UHCI` pertains to the transfer retransmission

logic, as illustrated below.

UHCI Transfer Retransmission Internal Logic

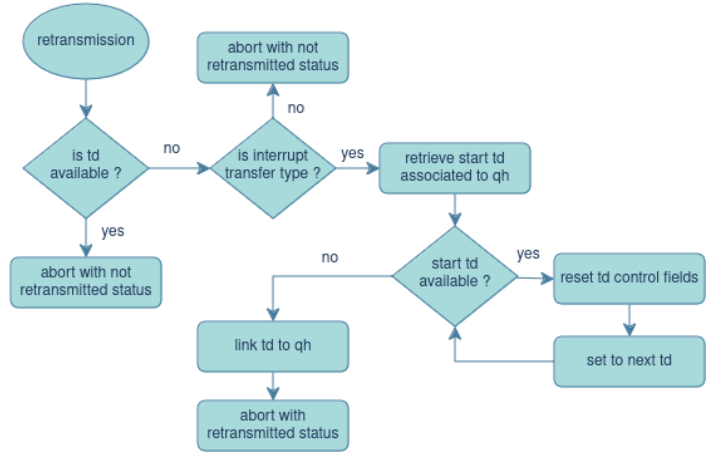


Figure 3.34: Transfer Retransmission

The retransmission process within the **UHCI** initiates with an assessment of the type of QH provided for retransmission. If the QH does not correspond to an interrupt QH, the process is terminated with a status indicating that retransmission is not applicable. Conversely, if the QH is identified as an interrupt QH, the initial TD associated with it is retrieved. Subsequently, all TDs are traversed, with their properties reset to their initial state. Upon completion of this traversal, the first TD is linked to the link pointer of the QH, thereby restoring the QH to its initial transfer state.

With a foundational understanding of the functionality encapsulated within the UHCI driver, attention will now be directed towards delineating the crucial components and functionalities offered by the USB device. Specifically, focus will be placed on the principal functionalities it offers, commencing with the logic governing the handling of the device.

UsbDevice Device Handling

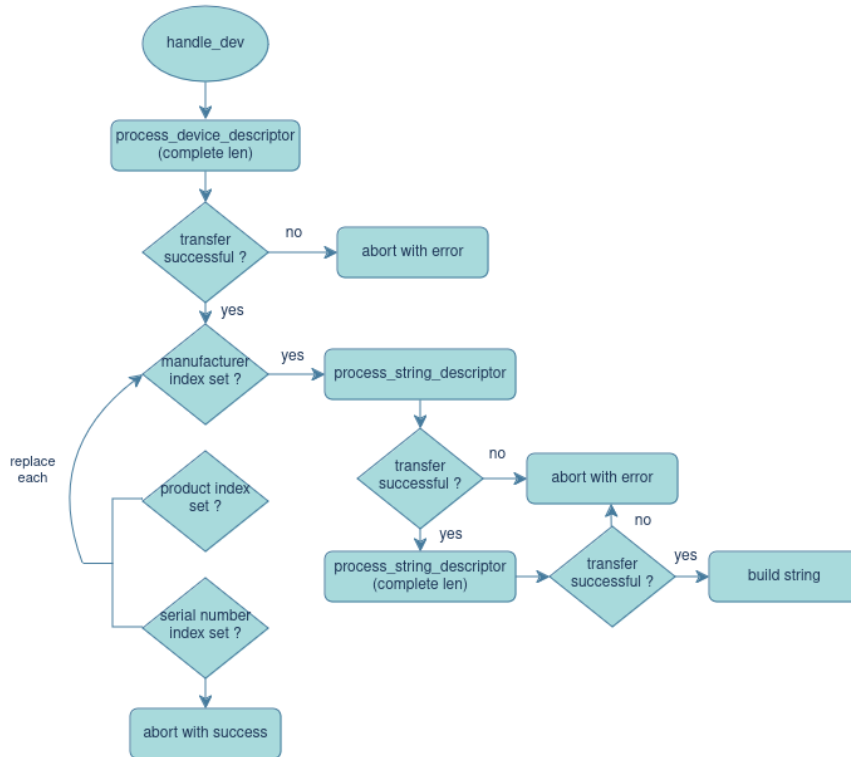


Figure 3.35: Device Handling

Within the `UsbDevice` component, the process of handling the device commences with the invocation of a function to process the device descriptor. In the event of an unsuccessful transfer, the handling procedure is terminated, and an error condition is aborted. Conversely, upon a successful transfer, the device descriptor is obtained, prompting an examination to ascertain whether the manufacturer index within the descriptor is set. If the manufacturer index is set, the processing proceeds to extract the string descriptor corresponding to the specified index. However, the extraction is performed for a partial length, as the length of the descriptor needs to be determined initially.

Following the extraction of the partial string descriptor, a further check is conducted to validate the success of the transfer. In the absence of any errors, the complete length of the string descriptor is obtained, prompting a subsequent request to retrieve the descriptor in its entirety. Upon a successful transfer, the complete string descriptor is obtained, facilitating the construction of the string representation from the retrieved descriptor. A similar approach is adopted for processing the product index and the serial number index.

Upon successful completion of all requisite operations, the handling procedure concludes with an abort signal indicating success.

Subsequently, attention will be directed towards examining the configuration level of the device, which represents a lower level of abstraction within the device hierarchy.

UsbDevice Configuration Handling

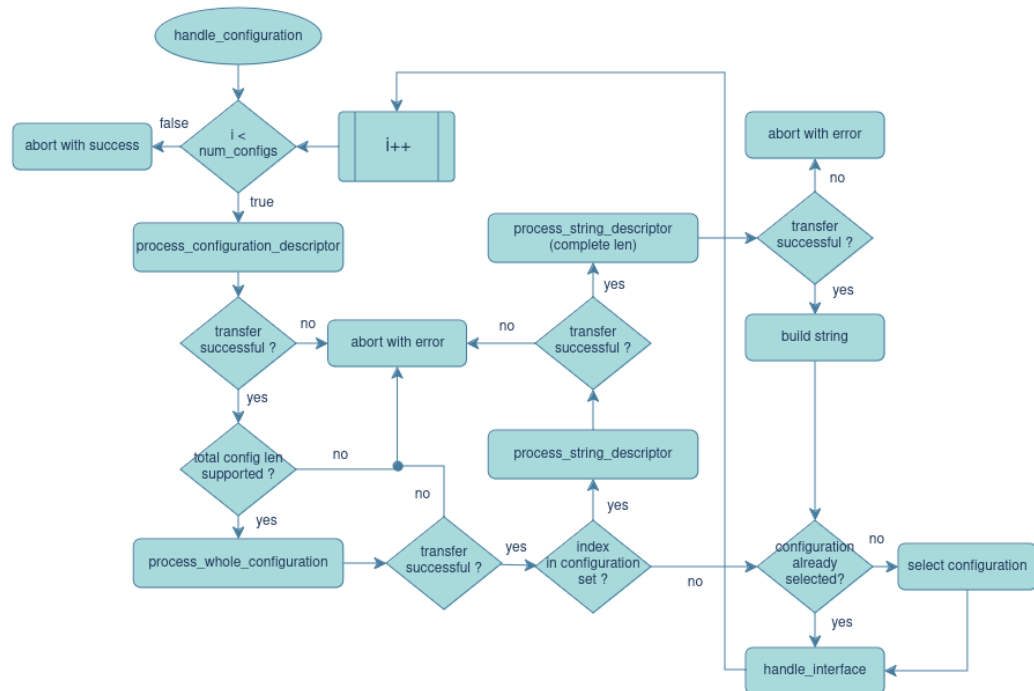


Figure 3.36: Configuration Handling

The management of configuration within the `UsbDevice` component commences with an iterative traversal through all available configurations within the device, as obtained from the device descriptor detailed in 3.35. Within each iteration, the configuration descriptor is processed to ascertain its properties, utilizing a fixed length for the initial request to determine the actual length of the configuration. In the event of an unsuccessful transfer, the process is terminated with an error. Conversely, upon a successful transfer, the length of the configuration, as obtained from the configuration descriptor, is evaluated to determine if it is supported by the system, adhering to predefined limits on configuration length.

If the configuration length is supported, the entire configuration is processed, with a subsequent check performed to validate the success of the transfer. In the event of a successful transfer, further processing ensues to determine if the configuration index within the configuration descriptor is set. If the configuration index is set, the processing proceeds in a manner akin to that described in 3.35. Subsequently, an examination is conducted to determine if the select bit has been set. The absence of the select bit indicates that the current configuration is designated as the default configuration to be utilized, necessitating the setting of the select bit accordingly.

Following the configuration selection process, the functionality proceeds to handle the interface, as delineated in 3.37. In the absence of any errors throughout the process, the operation concludes with an abort signal indicating success.

Having explored the processing of configurations, the focus now shifts towards a deeper examination of the handling of interfaces within the `UsbDevice`.

UsbDevice Interface Handling

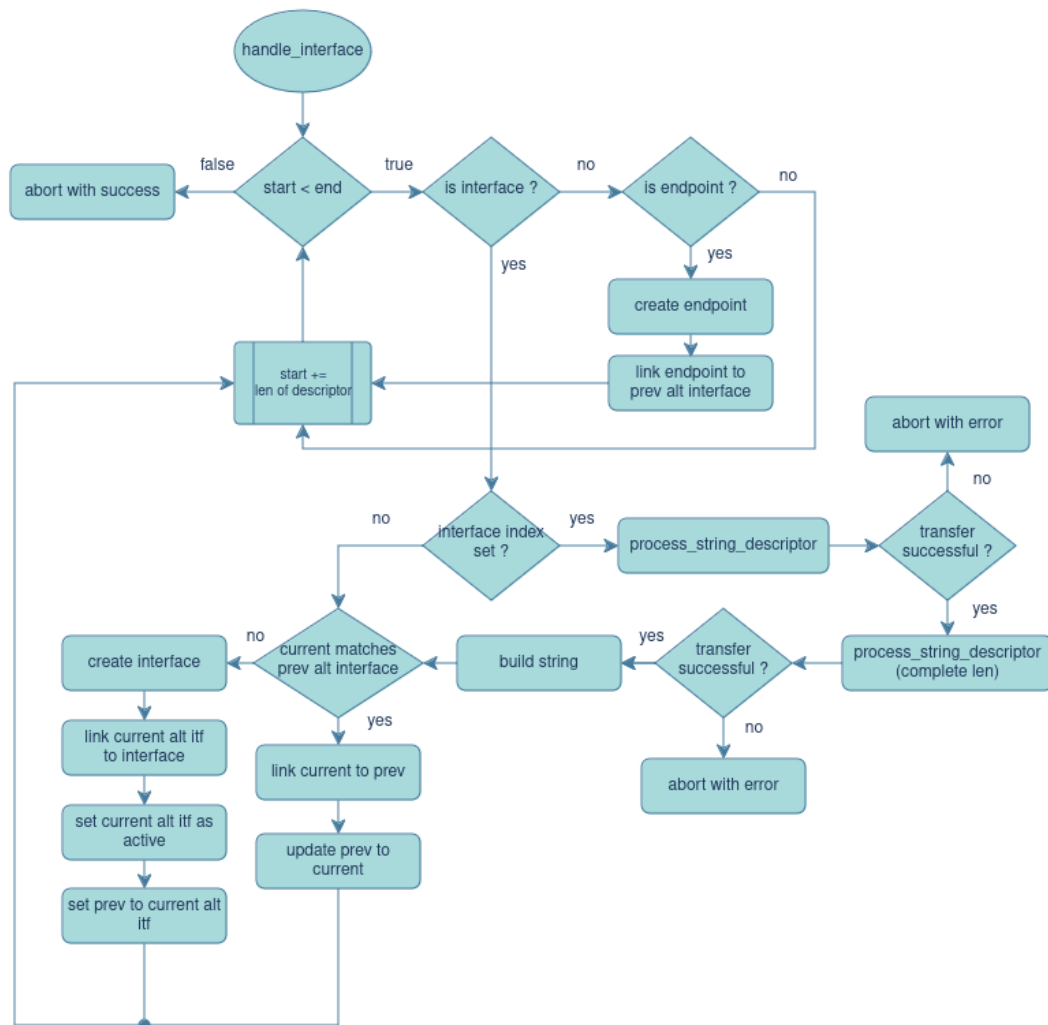


Figure 3.37: Interface Handling

The handling of interfaces within the `UsbDevice` component commences with an iterative process, continuing until the start address matches the end address, as determined by the configuration descriptor. These addresses delineate the boundaries within which descriptors are located. Subsequently, each descriptor is traversed, with the type of descriptor being ascertained to determine its nature. Specifically, emphasis is placed on identifying interface descriptors.

Upon encountering an interface descriptor, the module evaluates whether the interface index is set. In instances where it is set, the string build process described in detail in 3.35 is enacted. Following this step, a comparison is made between the current and previous alternate interfaces. Discrepancies between the two indicate that the current alternate interface does not correspond to the current interface, prompting the creation of a new interface. This new interface is linked to the alternate interface, signifying it as an option for the current interface, with the current alternate interface designated as the default alternate interface. Subsequently, the previous alternate interface is updated to

reflect the current alternate interface, and the iteration advances.

In cases where the current and previous alternate interfaces match, it signifies that the current interface offers multiple options. In such instances, the current alternate interface is linked to the previous alternate interface, establishing a chain of possible alternate interfaces. Following this linkage, the previous alternate interface is updated to reflect the current alternate interface, and the iteration proceeds.

Following the processing of interface descriptors, the module evaluates endpoint descriptors. In the event of an endpoint descriptor, an endpoint is created for the specific alternate interface, designated as the previous alternate interface. Upon completion of successful processing, the operation concludes with an abort signal indicating success.

The subsequent functionality of significance pertains to the processing of the configuration descriptor.

UsbDevice Processing Configuration

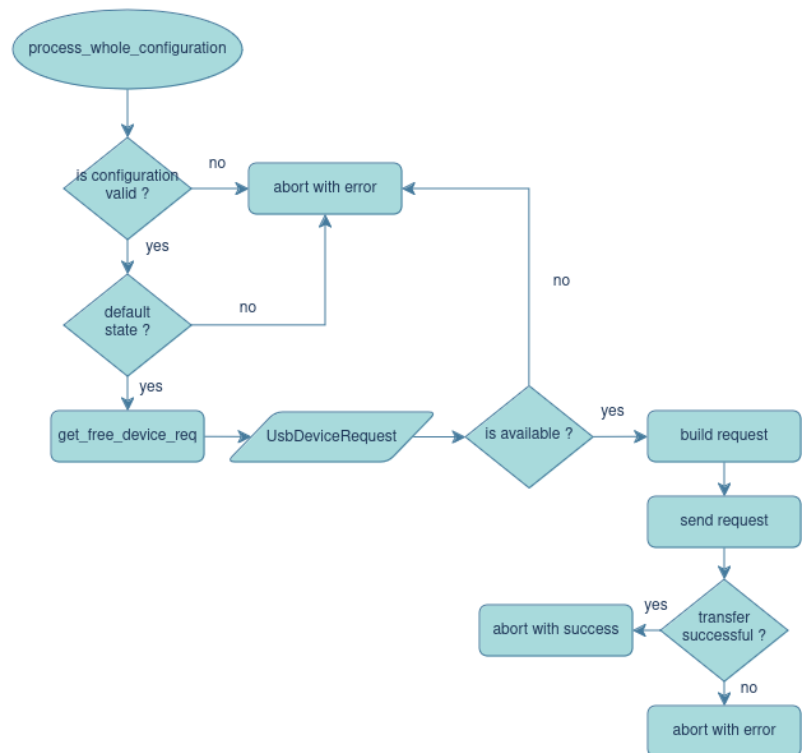


Figure 3.38: Configuration Descriptor Processing

The process of handling the configuration descriptor within the `UsbDevice` component initiates with a validation step to determine the configuration's validity. Should the configuration prove to be invalid, the operation is terminated with an error indication. Subsequently, a verification is conducted to ascertain whether the device is in its default state. If the device is indeed in its default state, the request is deemed impermissible, and the operation is aborted with an error.

Upon successful completion of these preliminary checks, a free structure known as the `UsbDeviceRequest` is requested. This structure serves as the data container for standard requests and is utilized to facilitate communication with the USB device. In the event that

the `UsbDeviceRequest` structure is unavailable, rendering the execution of the request infeasible, the operation is terminated with an error.

Conversely, if the `UsbDeviceRequest` structure is available, the request is constructed, as detailed in 2.9, and subsequently forwarded for processing. Following the completion of the request, the success of the transfer is verified, and the status is returned accordingly.

Another noteworthy aspect of functionality within the system pertains to the processing of string descriptors.

UsbDevice Processing String Descriptor

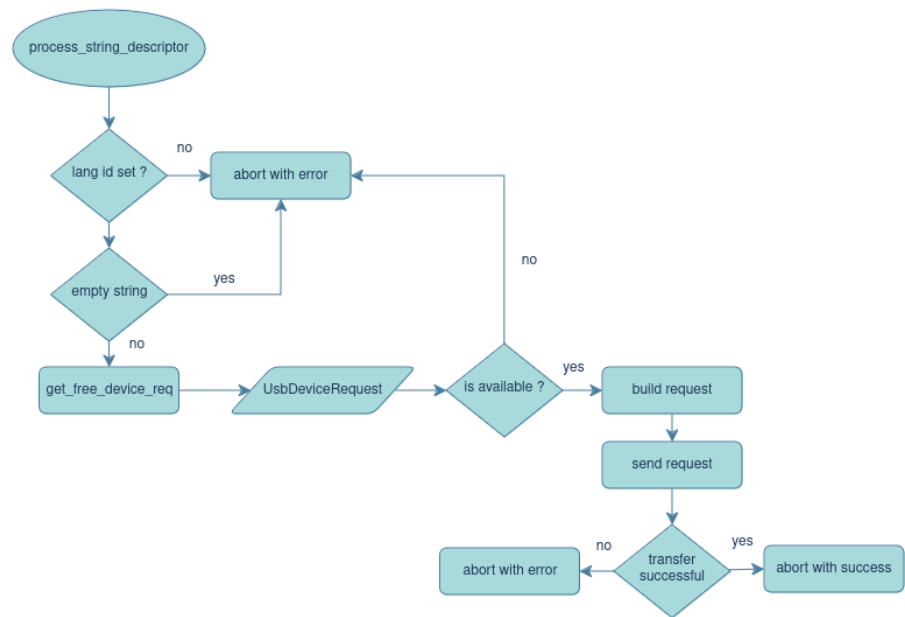


Figure 3.39: String Descriptor Processing

The procedure for processing string descriptors within the `UsbDevice` component commences with a validation step to determine whether a language ID is set. In the absence of a set language ID, each request for a string descriptor would result in an error. Consequently, the operation is halted, and an error is signaled.

Subsequently, checks are performed to verify whether the provided string, denoting the textual representation of the index field, is empty or not. Should the string be empty, indicating a lack of specified index, the operation is aborted with an error.

Upon successful completion of these preliminary checks, an optional `UsbDeviceRequest` is obtained from the relevant function. Following validation of its availability, the request is constructed and sent for processing. Subsequently, the success of the transfer is ascertained, and the corresponding message is returned. Specifically, the request for the string descriptor (2.18) is employed for this purpose.

It is imperative to note that preceding access to this descriptor, an initial request (2.17) is executed within the initial state of the `UsbDevice` component to establish the language ID.

Concluding this functionality, the process of driver registration is outlined, as depicted in the workflow below.

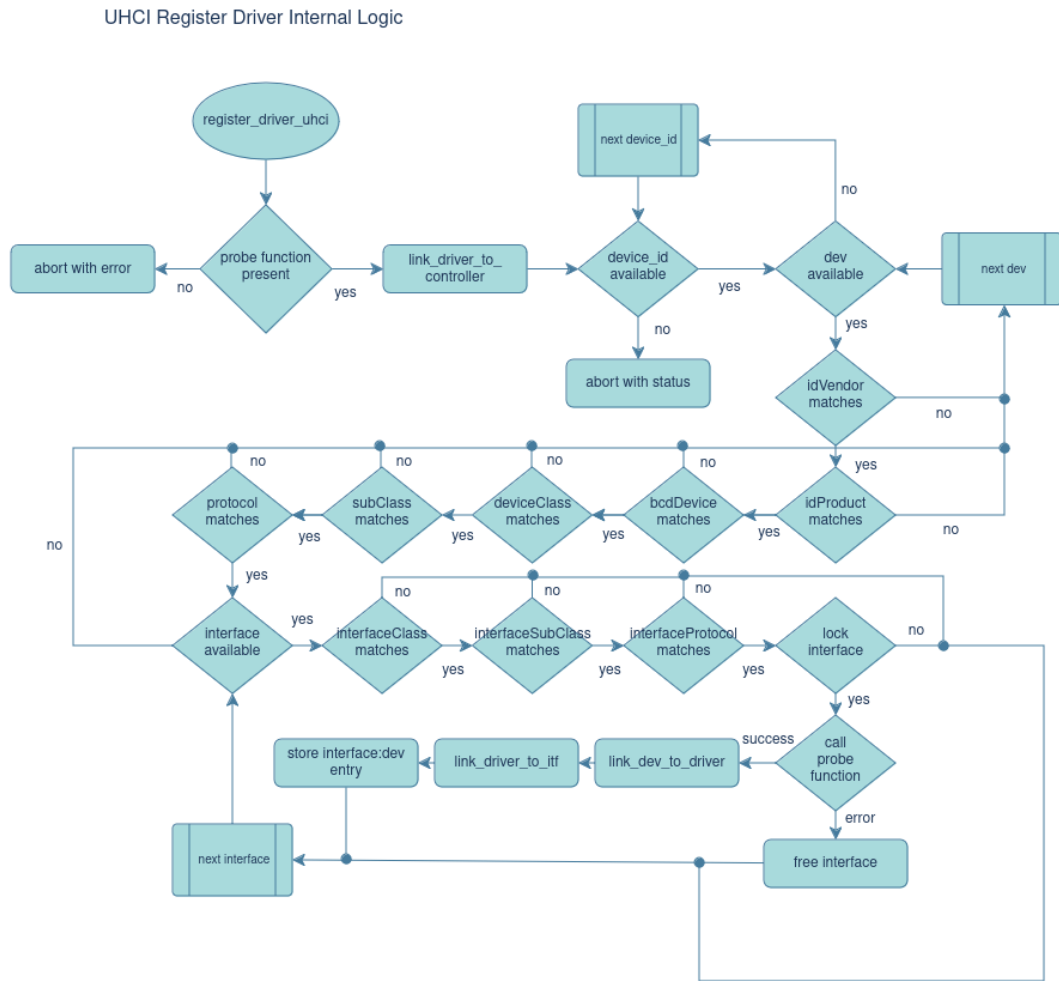


Figure 3.40: Driver Registration

The process of driver registration within the **UHCI** entails several sequential steps. Firstly, the availability of a probe function is verified. In the absence of a probe function, the registration process is immediately terminated. Conversely, if a probe function is present, the driver is linked to the controller, thereby listing the drivers that can be utilized for downstream devices.

Subsequently, an iterative process ensues, wherein each **Device ID** provided by the driver is examined. For each **Device ID**, the UHCI driver traverses all connected USB devices. A series of checks are performed between the specific device and the **Device ID** to determine if the device is compatible with the driver. The initial checks involve device-specific criteria, including device type, vendor ID, product ID, and protocol matches.

Upon successful completion of the device specific checks, the process proceeds to interface checks within the specific device's active configuration. Once again, all available interfaces are examined, and various criteria are assessed to determine if the driver supports the interface type. If the interface checks yield positive results, indicating compatibility with the driver, the associated interface is locked, and the probing function associated with the

driver is invoked.

Should the probing function return an error, signifying that the driver does not support the interface, the interface is freed, and the lock status is removed. Conversely, if the probing process is successful, indicating compatibility with the interface, the driver is linked to the associated devices, and the interface is associated with the driver. Furthermore, an entry mapping the interface to the device is stored for future reference.

3.1.8 Functional Analysis of Driver System Components

Having acquired a comprehensive understanding of the functionalities inherent in the core system components, attention will now be directed towards examining the operational intricacies of the driver system components in greater detail.

Commencing with the hub driver, a meticulous examination of the configuration process is warranted.

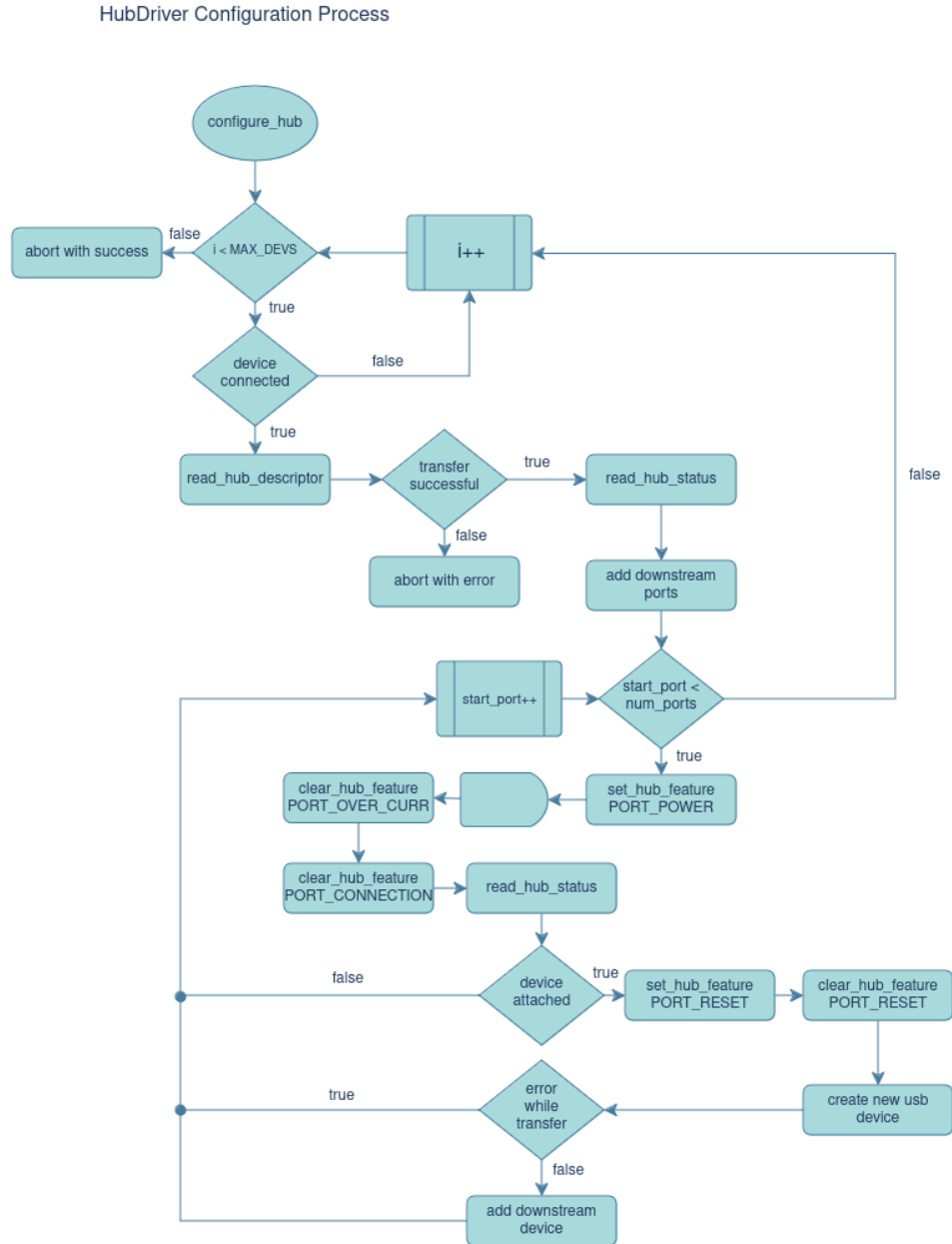


Figure 3.41: Hub Driver Configuration

Initiating the configuration process of the `HubDriver` involves an iterative procedure, iterating through the maximum allowable device limit. For each index i , the presence of a connected device to the driver is assessed. If no device is detected, the process proceeds to the next index i .

Upon detecting a connected device, the hub descriptor is read. Should the transfer fail to succeed, the configuration process is halted, and an error is reported. Conversely, upon successful retrieval of the hub descriptor, the hub status is acquired, thereby providing comprehensive information pertaining to the hub, including the number of downstream ports available. Subsequently, the downstream ports are added to the device.

Continuing with the configuration, each port is individually processed. The process entails powering on the downstream port followed by a variable wait time, as determined by the hub descriptor, multiplied by a factor of 2 for optimal wait time calculation. Subsequently, the `PORT_OVER_CURRENT_CHANGE` and `PORT_CONNECTION_CHANGE` bits are cleared. The hub status is then re-read to determine if a device is attached to the port. In the absence of an attached device, processing proceeds to the next downstream port of the hub.

However, if a device is detected, the `PORT_RESET` bit is set, automatically clearing upon completion of the reset process. Following the reset, if a device is still attached to the port, it is enabled and deemed operational. The `PORT_RESET_CHANGE` bit is then cleared to indicate the completion of the reset process.

Subsequent to port reset, the attached device enters its default state, enabling enumeration. A new USB device corresponding to the attached device is generated. If the device creation process is successful, the downstream device is added to the hub device, effectively chaining all present devices to the hub device. Conversely, in the event of an error during device creation, the device is not listed, and processing continues to the next downstream port.

Moving forward, an examination of the configuration process within the mass storage driver is warranted.

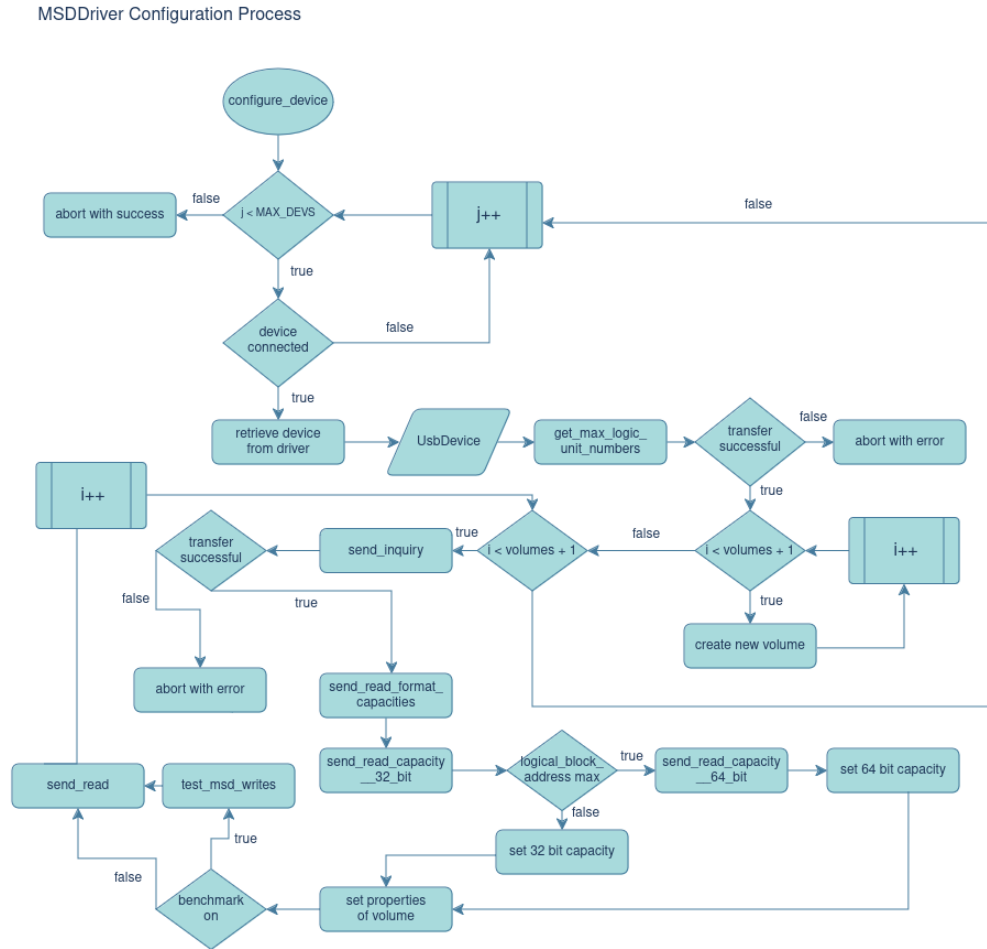


Figure 3.42: Mass Storage Driver Configuration

The configuration process of the `MSDDriver` unfolds through an iterative procedure, looping through the maximum permissible device limit. For each index j , the presence of a connected device to the driver is evaluated. Should no device be detected, j is incremented by one, thereby advancing the iteration. Conversely, upon detecting a connected device, the physical device associated with the driver, denoted as the `UsbDevice`, is retrieved.

Within the `UsbDevice`, a call is made to ascertain the total number of volumes present within the mass storage device. Should the transfer fail to succeed, the configuration process is terminated, and an error is reported. Given that the volume enumeration is zero based, an iteration spanning `volumes + 1` is initiated. For each volume enumerated, a new volume is generated, each associated with a distinct volume number.

Subsequently, a secondary iteration is initiated over all volumes. Within each iteration, the send inquiry command is executed. The success or failure of the transfer dictates the subsequent course of action. Following the inquiry command, the send read format capacities function and the send read capacity (32-bit) function are executed. It is imperative to verify the success of each transfer, aborting in the event of an error.

Upon obtaining the read capacity, an assessment is made to determine if 32-bit addressing is sufficient. If 32-bit addressing proves inadequate, an additional send read capacity call for 64-bit addressing is executed. Subsequently, the 64-bit capacity is designated as active

if 64-bit addressing is deemed necessary; otherwise, the 32-bit capacity is designated as active.

Subsequent to setting the basic volume properties, such as volume size, block size, and block number, a check is performed to determine if benchmarks are to be conducted. In the event that benchmarking is activated, tests for writing and reading are generated.

Finally, a read request is made to obtain the first 64 blocks of the drive. Ordinarily, these blocks would undergo processing to ascertain the file system in use; however, such processing is beyond the scope of this study.

Moving forward, an examination of the probing process within the keyboard driver is warranted.

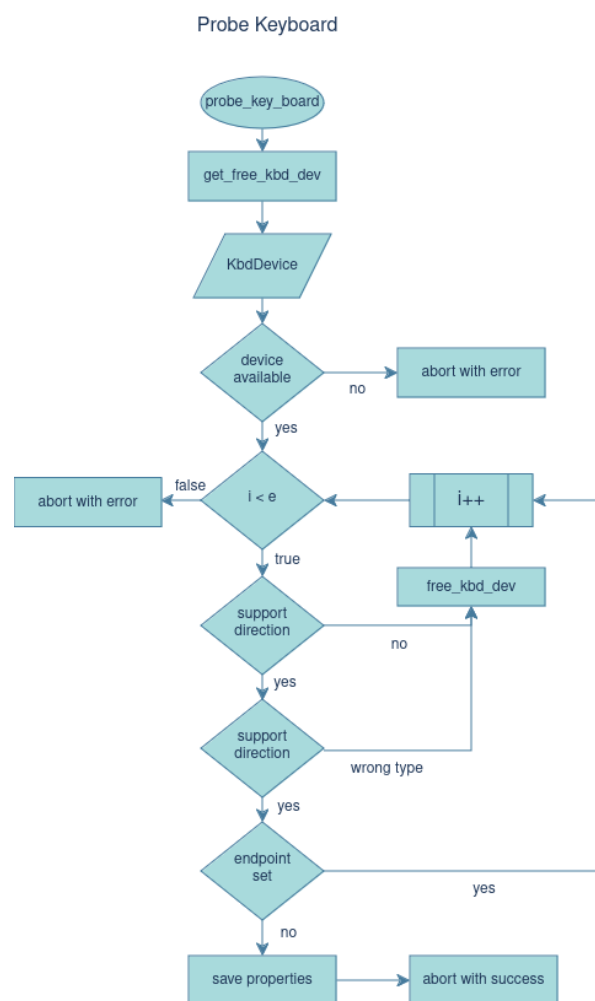


Figure 3.43: Keyboard Driver Probing

The probing mechanism employed by the `KeyboardDriver` begins with a preliminary assessment to ensure that the device limit has not been exceeded. Should the limit be reached, rendering the `KbdDevice` inaccessible, the process is promptly terminated with an error. Conversely, in the event that the device is available for probing, the driver proceeds to iterate through all endpoints within the interface provided as an argument by the `UHCI`.

During each iteration, the driver subjects the endpoints to a series of checks pertaining to directionality and transfer type. These checks serve to filter out endpoints that are pertinent to communication. Subsequently, if an endpoint has not already been configured within the `KbdDevice`, the driver proceeds to set the necessary properties within the device, including the endpoint address and interval, among others.

Upon successfully configuring these properties, the probing process concludes with a status of success, signifying that the interface should be linked to this driver. However, should no suitable endpoint be found within the interface, the probing process is aborted with an error, indicating that the interface is not compatible with the driver and should not be linked to it.

Continuing the exploration, attention is directed towards the callback functions implemented within the drivers.

Commencing with the keyboard driver's callback function:

Callback KBDDriver

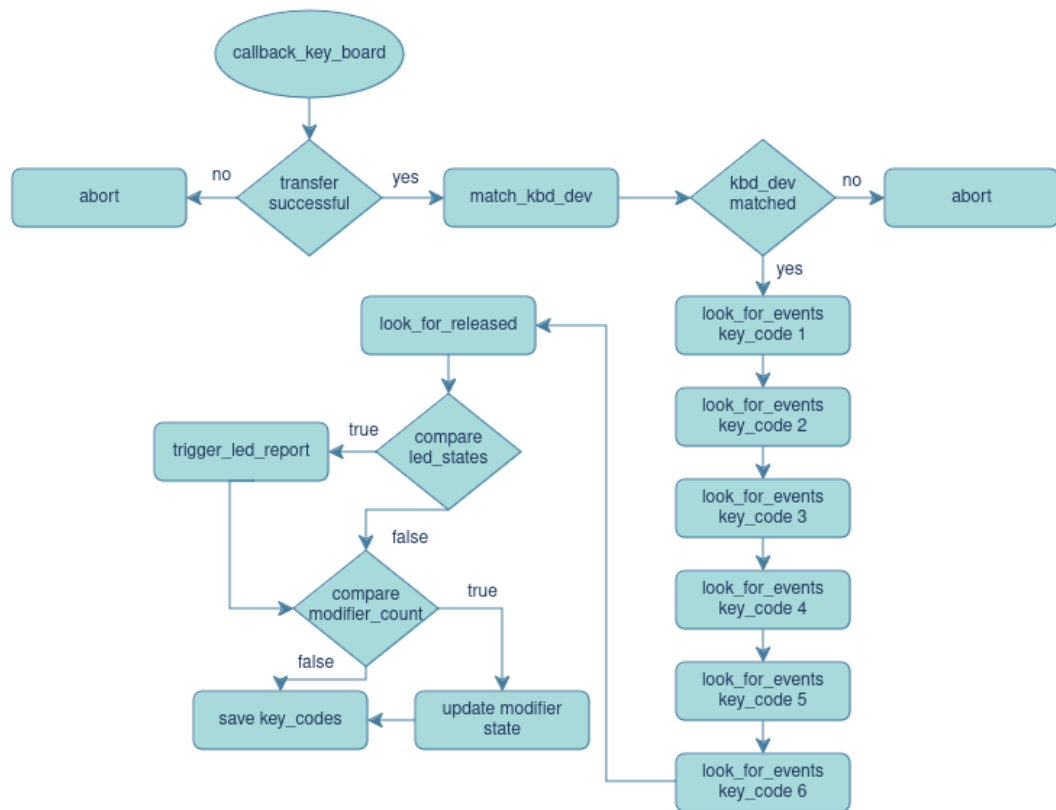


Figure 3.44: Keyboard Driver Callback

The operational flow of the callback function within the `KeyboardDriver` begins with a preliminary assessment of the transfer's success status. This initial check discerns whether the transmission encountered any errors; should the transfer be unsuccessful, the process is promptly terminated. Conversely, upon a successful transfer, the keyboard driver proceeds to retrieve the key data from the connected keyboard device.

Following this, a verification step ensues to ascertain whether the provided `UsbDevice`

aligns with any device stored within the driver. Should this verification fail, indicating a discrepancy between the provided device and the driver's records, the process is halted.

Upon successful validation of the device, the driver proceeds with the processing of the transmitted data. Given the existence of six distinct key codes, the driver iterates through each code individually, discerning the occurrence of key presses and identifying the specific keys activated. Similarly, the callback examines whether any keys have been released.

Subsequently, the callback evaluates whether a LED report needs to be triggered. This determination hinges on the state of LEDs within the driver, such as those associated with the 'Caps Lock' key. Activation or deactivation of these LEDs necessitates a request to modify the corresponding LED state.

Additionally, the callback conducts a modifier count check to ascertain if the modifier count has surpassed a predefined threshold. Upon meeting this threshold, signaling a sufficient number of modifier key activations, the callback initiates the transmission of an empty report solely containing the modifier information. Following this action, the modifier count is reset to its initial state to enable subsequent reports.

Concluding the callback process, each key code obtained is stored within the local buffer residing within the device, facilitating further processing or storage as necessary.

The callback functions within the `MSDDriver` merit an analytical exploration, albeit their intricacy may initially pose challenges to comprehension.

The procedure to engage the chaining mechanism within the mass storage driver commences with the initiation of an appropriate read or write operation. In this exposition, the read function is explicated in detail. Initially, the function endeavors to procure an optional `CommandBlockWrapper`. Should this optional entity be absent, indicating the unavailability of a `CommandBlockWrapper`, the function halts with an error. Subsequently, the `CommandBlockWrapper` undergoes initialization through the invocation of an initializer function, endowed with a range of properties. Failure in this initialization process results in an abort with an error. Conversely, upon successful initialization, the request to dispatch the `CommandBlockWrapper` ensues, accompanied by the specification of the callback invoked upon completion of the transfer, here designated as the read callback within this driver. Should an error arise during the request to transmit the `CommandBlockWrapper`, a corresponding abort is triggered, leading to an error notification.

Upon the culmination of the transfer, denoted by time X, the predesignated callback associated with the request is invoked. Within this callback, the first step entails verifying the success of the `CommandBlockWrapper` transfer. Should the transfer encounter an error, the process is terminated prematurely. Conversely, if the device passed into the callback is validated as an entity within the driver, the subsequent action involves initiating the request to retrieve data from the device. Once again, a callback is designated to handle the conclusion of this transfer, herein referred to as the `MSD callback`. Failure in retrieving the data prompts invocation of the user's callback, signifying the absence of any data copied.

Upon the completion of the data transfer, occurring at time X, the designated `MSD callback` is invoked. Within this context, the success of the transfer is ascertained, leading either to the invocation of the user's callback, indicating a successful data copy, or the notification of an error condition. Proceeding with success, the subsequent step involves querying the status of the transfer through another designated callback, such as the `CSW callback`. Upon the conclusion of this process, a final verification of the transfer status is conducted within the callback invoked by the `UHCI`. Here, scrutiny is directed towards the `CommandStatusWrapper` to ascertain the success of the transfer, based on the contents of its status and data residue fields. In the event of errors, the user's callback is invoked, and the process is terminated. Conversely, in the absence of errors, signifying the completion of a successful data transfer, the acquired data is copied to the target buffer. Finally, the user's callback is invoked, indicating the successful writing of a specified amount of data T to the buffer.

The analogous procedure unfolds in the write operation within the `MassStorageDriver`. However, diverging from the approach of interlinking the read callback with other callbacks, the write callback is instead integrated into the sequence with the other callbacks.

The final aspect of functionality to be scrutinized pertains to the control I/O within the kernel mass storage driver. This component necessitates elucidation to facilitate comprehension of its operational mechanisms. While an exhaustive examination of every facet delineated in the diagram is impractical within this scope, focus will be directed towards elucidating the workflow encapsulated therein.

KernelMassStorageDriver Control I/O

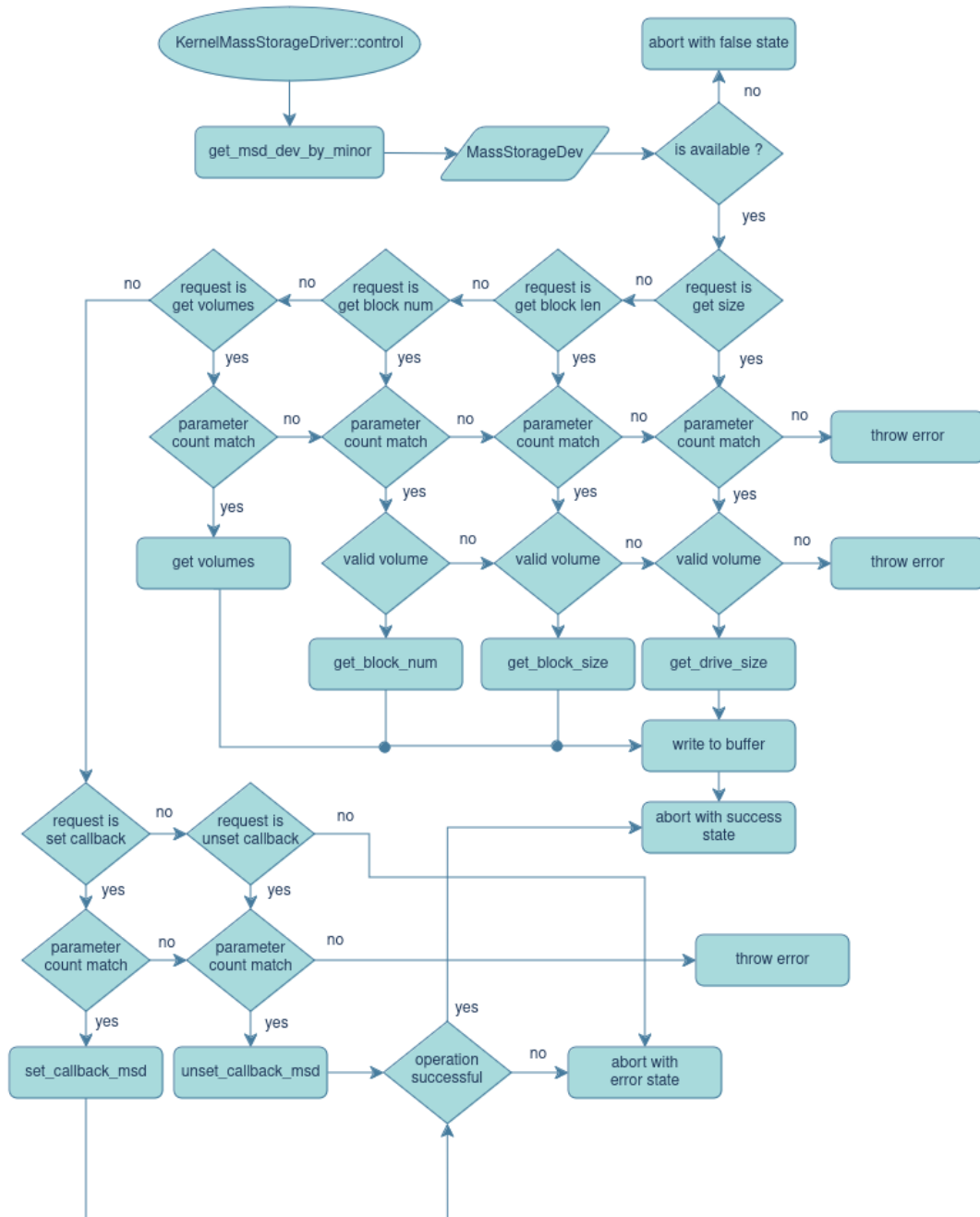


Figure 3.46: Mass Storage Driver Control I/O

The control I/O operations within the `KernelMassStorageDriver` commence by querying the `MassStorageDev` linked to a specific minor number. Should the `MassStorageDev` be absent, signifying that the minor is unsupported by the driver, the process terminates with a return state denoting falsehood. Conversely, if the device is present, a series of cases is enacted to discern the precise request made. Among these requests, particular attention will be devoted to the initial inquiry pertaining to size, followed by an examination of the set callback request.

Initiating with the size request, meticulous scrutiny is directed towards verifying the correctness of the transmitted parameter length. Failure to meet this criterion results in an abortive process, accompanied by an error notification. Subsequently, the pertinent parameters are extracted from the user input. This request expects two parameters: the user address and the volume designation.

It is imperative to note that in typical scenarios, stringent security checks would precede any attempt to access a user-provided address, safeguarding against unauthorized read/write operations. However, due to the absence of a dedicated security interface within this operating system, security considerations were omitted from the implementation.

Upon obtaining these parameters, validation ensues to confirm the legitimacy of the specified volume within the device. Invalidation prompts an exception to be raised. Otherwise, the driver is invoked to retrieve the drive size, which is subsequently inscribed at the provided address. The operation culminates in the return of a true state.

In the context of the set callback request, analogous validation procedures are enacted to ensure the accurate transmission of parameter size. Once validated, two parameters are anticipated: the callback address and the associated unique field. Subsequent to parameter acquisition, the driver is summoned to configure the callback. If the configuration process fails, a false state is returned; otherwise, a true state is returned.

As iterated previously, the remaining requests function in a manner akin to those elucidated above. A comprehensive understanding of these operations renders one sufficiently equipped to navigate the associated functionalities.

Having comprehensively elucidated the architecture of the system, encompassing an in depth understanding of its key components and their interconnections, a solid foundation has been laid for delving into the intricacies of the code base.

3.2 Implementation

In the following sections, attention turns towards a comprehensive analysis of the codebase developed for this project. The code architecture predominantly employs the C programming language, structured in accordance with object-oriented principles to enhance comprehensibility and sustainability. Additionally, complementary wrapper components are implemented in C++, serving to interface with the underlying system and external services. This design ensures seamless interaction between the C-based components and their C++ counterparts, facilitating mutual accessibility and interoperability. Throughout this thesis, emphasis will be placed on dissecting pivotal components and functionalities inherent to the codebase, elucidating their significance and impact.

3.2.1 Utility

Prior to delving into the practical implementation details, it is prudent to discuss certain foundational elements employed therein. One such fundamental construct is the `container_of` macro [12], which plays a pivotal role in accessing specific sub-components within a larger structure. This macro is structured as follows:

```
container_of(ptr, type, member)
```

The `ptr` parameter denotes the pointer whose enclosing structure is to be reconstructed. The `type` parameter specifies the type of the container structure from which the sub-component is to be extracted. Lastly, the `member` parameter indicates the field within the container structure that corresponds to the provided pointer. Upon invocation, this macro facilitates the retrieval of the container structure containing the specified field. [12]

Another integral aspect of the implementation involves the utilization of `struct list_head` [13], [14] and `struct list_element`, which serve as mechanisms for linking components. Rather than directly linking the components themselves, these structures facilitate the linkage process.

The `struct list_head` comprises a single field, `struct list_element* l_e`, representing the first element in the linkage sequence. Conversely, the `struct list_element` solely consists of a `struct list_element* l_e`, indicating the linkage to the subsequent element and thereby forming a chain of elements.

3.2.2 Core System Component Implementation

Upon delving into the internal intricacies of the utilities employed within the system, the focus now shifts to elucidating the process of defining super components within the system itself. To illustrate this, the construction of a super component, employing the `UsbController` as a reference point, will be examined.

```

UsbController.h
1 struct UsbController {
2     // non abstract
3     void (*new_usb_controller)(struct UsbController *usb_controller,
4                               SystemService_C* mem_service, const char *name);
5     int (*insert_callback)(struct UsbController *controller, uint16_t reg_type,
6                           event_callback callback);
7     int (*delete_callback)(struct UsbController *controller, uint16_t reg_type,
8                           callback callback);
9     int (*insert_listener)(struct UsbController *controller,
10                          EventListener *listener);
11     int (*delete_listener)(struct UsbController *controller, int id);
12
13     // abstract
14     void (*poll)(struct UsbController *usb_controller);
15     uint16_t (*reset_port)(struct UsbController* controller, uint8_t port);
16     void (*interrupt_entry_point)(UsbDev* dev, Endpoint* endpoint,
17                                  void* data, unsigned int len, uint8_t prio, uint16_t interval,
18                                  callback_function callback);
19     void (*control_entry_point)(UsbDev* dev, UsbDeviceRequest* device_request,
20                                void* data, uint8_t prio, Endpoint* endpoint,
21                                callback_function callback, uint8_t flags);
22     void (*bulk_entry_point)(UsbDev* dev, Endpoint* endpoint, void* data,
23                              unsigned int len, uint8_t priority,
24                              callback_function callback, uint8_t flags);
25
26     void (*control)(struct UsbController *controller, Interface *interface,
27                    unsigned int pipe, uint8_t priority, void *data,
28                    uint8_t *setup, callback_function callback);
29     void (*interrupt)(struct UsbController *controller, Interface *interface,
30                      unsigned int pipe, uint8_t priority, void *data,
31                      unsigned int len, uint16_t interval,
32                      callback_function callback);
33     void (*bulk)(struct UsbController *controller, Interface *interface,
34                  unsigned int pipe, uint8_t priority, void *data,
35                  unsigned int len, callback_function callback);
36
37     int (*register_driver)(struct UsbController *usb_controller, UsbDriver *driver);
38     int (*remove_driver)(struct UsbController *usb_controller, UsbDriver *driver);
39
40     void (*runnable_function)(struct UsbController* controller);
41     void (*handler_function)(struct UsbController* controller);
42 };

```

As depicted in the illustration above, the `UsbController` component is subdivided into two distinct segments. The initial segment encompasses a collection of non-abstract functions, denoting functions that are directly implemented within the `UsbController` component. In contrast, the subsequent segment comprises abstract functions, which are merely defined within the `UsbController` but necessitate implementation elsewhere. Specifically, the implementation of these abstract functions must be furnished within the context of a specific controller. For instance, upon integrating the `UHCI`, the requisite implementations for these abstract functions must be provided within the corresponding component, as illustrated in the subsequent depiction.

```

UHCI.h
1 int register_driver_uhci(UsbController *controller, struct UsbDriver *driver);
2 int deregister_driver_uhci(UsbController *controller, struct UsbDriver *driver);
3 void init_control_transfer(UsbController *controller, Interface *interface,
4     unsigned int pipe, uint8_t priority, void *data,
5     uint8_t *setup, callback_function callback);
6 void init_interrupt_transfer(UsbController *controller, Interface *interface,
7     unsigned int pipe, uint8_t priority, void *data,
8     unsigned int len, uint16_t interval,
9     callback_function callback);
10 void init_bulk_transfer(UsbController *controller, Interface *interface,
11     unsigned int pipe, uint8_t priority, void *data,
12     unsigned len, callback_function callback);
13 uint16_t uhci_reset_port(UsbController *controller, uint8_t port);
14 void bulk_entry_point_uhci(struct UsbDev *dev, Endpoint *endpoint, void *data,
15     unsigned int len, uint8_t priority,
16     callback_function callback, uint8_t flags);
17 void control_entry_point_uhci(struct UsbDev *dev,
18     struct UsbDeviceRequest *device_request, void *data,
19     uint8_t priority, Endpoint *endpoint,
20     callback_function callback, uint8_t flags);
21 void interrupt_entry_point_uhci(struct UsbDev *dev, Endpoint *endpoint,
22     void *data, unsigned int len, uint8_t priority,
23     uint16_t interval, callback_function callback);
24 void _poll_uhci(UsbController *controller);
25 void handler_function_uhci(UsbController* controller);
26 void runnable_function_uhci(UsbController* controller);

```

With the abstract functions from the `UsbController` component accordingly implemented, invoking the set of functions from the interface facilitates the delegation of these calls to the pertinent controller, where the corresponding functions are executed.

```

UHCI.h
1 struct _UHCI {
2     struct UsbController super;
3     void (*new_UHCI)(struct _UHCI *uhci, PciDevice_Struct* pci_device,
4         SystemService_C* mem_service);
5
6     QH* (*request_frames)(struct _UHCI* uhci);
7     Addr_Region* (*i_o_space_layout_run)(struct _UHCI* uhci);
8     Register** (*request_register)(struct _UHCI* uhci);
9
10    void (*insert_queue)(struct _UHCI *uhci, struct QH *new_qh,
11        uint16_t priority, enum QH_HEADS v);
12    void (*remove_queue)(struct _UHCI *uhci, struct QH *qh);
13
14    unsigned int (*retransmission)(struct _UHCI *uhci, struct QH *process_qh);
15    uint32_t (*get_status)(struct _UHCI *uhci, struct TD *td);
16    uint32_t (*wait_poll)(struct _UHCI *uhci, struct QH *process_qh,
17        uint32_t timeout);
18    void (*traverse_skeleton)(struct _UHCI *uhci, struct QH *entry);
19
20    UsbPacket *(*create_USB_Packet)(struct _UHCI *uhci, UsbDev *dev,
21        UsbPacket *prev, struct TokenValues token,
22        int8_t speed, void *data, int last_packet, uint8_t flags);
23    void (*control_transfer)(struct _UHCI *uhci, UsbDev *dev,
24        struct UsbDeviceRequest *rq, void *data,
25        uint8_t priority, Endpoint *endpoint,
26        UsbTransfer *(*build_control_transfer)(
27            struct _UHCI *uhci, UsbDev *dev,
28            struct UsbDeviceRequest *device_request,
29            void *data, Endpoint *endpoint, uint8_t flags),
30        callback_function callback, uint8_t flags);
31    void (*interrupt_transfer)(struct _UHCI *uhci, UsbDev *dev, void *data,
32        unsigned int len, uint16_t interval,
33        uint8_t priority, Endpoint *e,
34        UsbTransfer *(*build_bulk_or_interrupt_transfer)(
35            struct _UHCI *uhci, UsbDev *dev, void *data,
36            Endpoint *e, unsigned int len,
37            const char *type, uint8_t flags),
38        callback_function callback);
39    void (*bulk_transfer)(struct _UHCI *uhci, UsbDev *dev, void *data,
40        unsigned int len, uint8_t priority, Endpoint *e,
41        UsbTransfer *(*build_bulk_or_interrupt_transfer)(
42            struct _UHCI *uhci, UsbDev *dev, void *data,
43            Endpoint *e, unsigned int len, const char *type, uint8_t flags),
44        callback_function callback, uint8_t flags);
45
46    void (*init_controller_functions)(struct _UHCI *uhci);
47
48    void (*create_dev)(struct _UHCI *uhci, int16_t status, int pn,
49        MemoryService_C *m);
50    void (*controller_port_configuration)(struct _UHCI* uhci);
51    int (*controller_configuration)(struct _UHCI* uhci);
52 };

```

The subsequent noteworthy consideration is the necessity for the super component to be defined as the initial field within the structure, facilitating casting to the super component, namely the `UsbController`. This arrangement enables the invocation of functions and access to fields pertinent to the `UsbController`. The remaining functions within the `UHCI` component are dedicated to internal utilization, offering support in a variety of operational facets.

Additionally, it is imperative to underscore that each function encapsulates the respective component, thereby facilitating manipulation of the component passed into it, as well as enabling invocation of functions associated with that component.

Moving forward, the procedural initialization of these functions will be examined. The `init_controller_functions` function serves as the primary mechanism for initially provisioning the functions corresponding to the abstract function definitions within the `UsbController`. Subsequently, functions pertinent to the function definitions within the `UHCI` component are provided.

```

UHCI.c
1 void init_controller_functions(_UHCI *uhci) {
2     uhci->super.poll = &_poll_uhci_;
3     uhci->super.register_driver = &register_driver_uhci;
4     uhci->super.remove_driver = &deregister_driver_uhci;
5     uhci->super.control = &init_control_transfer;
6     uhci->super.interrupt = &init_interrupt_transfer;
7     uhci->super.bulk = &init_bulk_transfer;
8     uhci->super.reset_port = &uhci_reset_port;
9     uhci->super.interrupt_entry_point = &interrupt_entry_point_uhci;
10    uhci->super.control_entry_point = &control_entry_point_uhci;
11    uhci->super.bulk_entry_point = &bulk_entry_point_uhci;
12    uhci->super.new_usb_controller = &new_super_usb_controller;
13    uhci->super.handler_function = &handler_function_uhci;
14    uhci->super.runnable_function = &runnable_function_uhci;
15    uhci->super.new_usb_controller(&uhci->super, uhci->mem_service, UHCI_name);
16
17    uhci->i_o_space_layout_run = &i_o_space_layout_run;
18    uhci->request_register = &request_register;
19    uhci->request_frames = &request_frames;
20    uhci->insert_queue = &insert_queue;
21    uhci->retransmission = &retransmission;
22    uhci->remove_queue = &remove_queue;
23    uhci->wait_poll = &wait_poll;
24    uhci->traverse_skeleton = &traverse_skeleton;
25    uhci->create_USB_Packet = &create_USB_Packet;
26    uhci->control_transfer = &control_transfer;
27    uhci->interrupt_transfer = &interrupt_transfer;
28    uhci->bulk_transfer = &bulk_transfer;
29    uhci->create_dev = &create_dev;
30    uhci->controller_configuration = &controller_configuration;
31    uhci->controller_port_configuration = &controller_port_configuration;
32 }

```

The initial phase involves the invocation of `uhci->super` coupled with the selection of the suitable function for implementation. For instance, the directive `uhci->super.poll = &_poll_uhci_;` signifies the provisioning of the poll function's implementation, as defined by the `UsbController`. Furthermore, this stage encompasses the assignment of internal function implementations within the `UHCI` structure. For instance, `uhci->i_o_space_layout_run = &i_o_space_layout_run;` delineates the definition of the function intended for a specific function pointer within the `UHCI` structure.

Subsequently, to interface with the UHCI connected to a PCI slot, it becomes necessary to extract information from the configuration space within the device itself and configure the device by writing appropriate values to it. To address this requirement, the ensuing functionality within the UHCI driver is leveraged.

```

UHCI.c
1 Addr_Region* i_o_space_layout_run(UHCI* uhci){
2   PciDevice_Struct* pci_device = uhci->pci_device;
3
4   command = uhci->pci_device->readWord_c(uhci->pci_device, COMMAND);
5
6   command = pci_device->readWord_c(pci_device, COMMAND);
7   command &= (0xFFFF ^ (INTERRUPT_DISABLE | MEMORY_SPACE));
8   command |= BUS_MASTER | IO_SPACE;
9   pci_device->writeWord_c(pci_device, COMMAND, command);
10
11   base_address = pci_device->readDoubleWord_c(pci_device, BASE_ADDRESS_4) &
12   0xFFFFFFFFFC;
13   uhci->irq = pci_device->readByte_c(pci_device, INTERRUPT_LINE);
14
15   pci_device->writeDoubleWord_c(pci_device, CAPABILITIES_POINTER, 0x00000000);
16   pci_device->writeDoubleWord_c(pci_device, 0x38, 0x00000000);
17
18   pci_device->writeWord_c(pci_device, 0xC0, 0x8F00);
19   pci_device->writeWord_c(pci_device, 0xC0, 0x2000);
20
21   io_port->newIO_Port(io_port, base_address);
22   io_region->new_io_region(io_region, io_port, 8);
23
24   return (Addr_Region*)io_region;
25 }

```

As a result, the `struct Addr_Region` is returned, furnishing all functionalities required to access the I/O space of the UHCI.

Subsequently, two crucial structures come into play, significantly influencing the actual transfers.

```

1 struct QH {
2   volatile uint32_t pyhsicalQHLP;
3   volatile uint32_t pyhsicalQHEP;
4   volatile uint32_t parent;
5   volatile uint32_t flags;
6 } __attribute__((packed));

```

```

1 struct TD {
2   volatile uint32_t pyhsicalLinkPointer;
3   volatile uint32_t control_x_status;
4   volatile uint32_t token;
5   volatile uint32_t bufferPointer;
6 } __attribute__((packed));

```

Both instances are characterized by the inclusion of the `volatile` keyword in all internal fields, serving to instruct the compiler to refrain from code optimization, given the requisite accessibility of these structures by the actual UHCI. Moreover, the entirety of both structures is configured as packed, indicative of the stipulation against padding, ensuring the contiguous arrangement of fields within memory.

The ensuing code excerpts delineate the concrete realization of I/O operations within the UHCI I/O space. All such operations are encapsulated as abstract functions within the `struct Addr_Region`, necessitating individual implementation by distinct sub components. The supported sub components encompass `struct Memory_Region` and `struct IO_Region`. However, within this instantiation, solely the latter component is

utilized, thus warranting exclusive depiction herein.

```

UHCIMemory.c
1 size_t io_read(struct Addr_Region* addr_region, uint8_t length,
2               void* r, void* buffer){
3     if(buffer == (void*)0) return 0;
4     enum Register_Type type = *((enum Register_Type*)r);
5
6     IO_Region* i_o_reg = (IO_Region*)container_of(addr_region, IO_Region, super);
7     if(length == 0x01){
8         int8_pvalue = (uint8_t*)buffer;
9         *(int8_pvalue) = i_o_reg->io_port->readByte_IO_off_C(
10            i_o_reg->io_port,type);
11         return length;
12     }
13     else if(length == 0x02){
14         int16_pvalue = (uint16_t*)buffer;
15         *(int16_pvalue) = i_o_reg->io_port->readWord_IO_off_C(
16            i_o_reg->io_port,type);
17         return length;
18     }
19     else if(length == 0x04){
20         int32_pvalue = (uint32_t*)buffer;
21         *(int32_pvalue) = i_o_reg->io_port->readDoubleWord_IO_off_C(
22            i_o_reg->io_port,type);
23         return length;
24     }
25     return 0;
26 }

```

The parameter list of these functions includes `struct Addr_Region* addr_region`, indicating the top component from which the read function is invoked. Subsequently, the length parameter is required to determine the number of bytes to be read, leading to the appropriate `i_o_reg->io_port->read` function call. The pointer `r` is a generic pointer containing the pointer to the `enum Register_Type`. The last parameter is also a generic pointer containing the buffer into which the data should be written.

Similarly, the next code snippet, representing the write operation into the I/O space of the UHCI, operates in a similar manner as explained above. However, unlike the read call, the buffer `b` is utilized to write its content to the I/O space.

```

UHCIMemory.c
1 size_t io_write(struct Addr_Region* addr_region, void* b,
2               uint8_t length, void* r){
3     IO_Region* i_o_reg = container_of(addr_region, IO_Region, super);
4     enum Register_Type type = *((enum Register_Type*)r);
5     if(length == 0x01){
6         int8_tvalue = *((uint8_t*)b);
7         i_o_reg->io_port->writeByte_IO_off_C(
8             i_o_reg->io_port, type, int8_tvalue);
9         return length;
10    }
11    else if(length == 0x02){
12        int16_tvalue = *((uint16_t*)b);
13        i_o_reg->io_port->writeWord_IO_off_C(
14            i_o_reg->io_port, type, int16_tvalue);
15        return length;
16    }
17    else if(length == 0x04){
18        int32_tvalue = *((uint32_t*)b);
19        i_o_reg->io_port->writeDoubleWord_IO_off_C(
20            i_o_reg->io_port, type, int32_tvalue);
21        return length;
22    }
23    return 0;
24 }

```

In order to denote a generic register corresponding to a precise offset within the I/O space, the `struct Register` is employed. This construct encapsulates a variety of abstract functions, notably including `read`, `write`, and `reload`, necessitating implementation for each distinct register within the UHCI. An illustrative instance of such a sub component housed within the `struct Register` is represented by the `struct Command_Register`, emblematic of the command register situated within the I/O space. Analogous implementations are extant for all other registers inherent to the UHCI, each bespoke in its realization of these functions.

The first code snippet demonstrates the process of updating the contents within the `struct Command_Register`.

```

UHCIRRegister.c
1 void command_reg_reload(Register *reg){
2     uint16_t d1 = *((uint16_t *)reg->raw_data);
3     Command_Register *creg = (Command_Register*)container_of(reg,
4         Command_Register, super);
5     creg->max_packet = (d1 & MAXP) >> 7;
6     creg->configure = (d1 & CF) >> 6;
7     creg->software_debug = (d1 & SWDBG) >> 5;
8     creg->global_resume = (d1 & FGR) >> 4;
9     creg->global_suspend = (d1 & EGSM) >> 3;
10    creg->global_reset = (d1 & GRESET) >> 2;
11    creg->host_controller_reset = (d1 & HCRESET) >> 1;
12    creg->run = (d1 & RS);
13 }

```

The subsequent code excerpt pertains to the procedure of reading the current contents within the register.

UHCIRegister.c

C

```

1 size_t command_reg_read(Register *reg, void* buffer){
2     Addr_Region *addr_reg = reg->addr_reg;
3     Register_Type type = reg->type_of(reg);
4     size_t read_count = addr_reg->read(addr_reg, reg->length, &type, buffer);
5     *((uint16_t*)reg->raw_data) = *((uint16_t*)buffer);
6     reg->reload(reg);
7     return read_count;
8 }

```

The final code snippet illustrates the method by which data is written into the register.

UHCIRegister.c

C

```

1 size_t command_reg_write(Register *reg, void* b){
2     size_t word_count;
3     uint16_t word = *((uint16_t*)b);
4     Addr_Region *addr_reg = reg->addr_reg;
5     Register_Type type = reg->type_of(reg);
6     word_count = addr_reg->write(addr_reg, &word, reg->length, &type);
7     *((uint16_t*)(reg->raw_data)) = word;
8     reg->reload(reg);
9     return word_count;
10 }

```

With the configuration of all distinct registers now established, the subsequent step involves the configuration of the UHCI itself. This is facilitated through the manipulation of its registers, achieved by invoking the following functions within the `UHCI`.

UHCI.c

C

```

1 int controller_configuration(_UHCI* uhci){
2     reg = uhci->look_for(uhci, Usb_Command);
3     // 50ms delay
4     for (int i = 0; i < 5; i++){
5         two_byte_command = GRESET;
6         reg->write(reg, &two_byte_command);
7         mdelay(USB_TDRST);
8         two_byte_command = 0;
9         reg->write(reg, &two_byte_command);
10    }
11    mdelay(USB_TRSTRCY);
12
13    reg->read(reg, &two_byte_command);
14    if (two_byte_command != 0x0000)
15        goto fail_label;
16
17    reg = uhci->look_for(uhci, Usb_Status);
18    reg->read(reg, &two_byte_command);
19    // HC Halted
20    if (two_byte_command != HALTED)
21        goto fail_label;
22    // write_clear
23    two_byte_command = 0x00FF;
24    reg->write(reg, &two_byte_command);
25
26    reg = uhci->look_for(uhci, Usb_Command);
27    two_byte_command = HCRESET;
28    reg->write(reg, &two_byte_command);
29
30    mdelay(10);
31
32    reg->read(reg, &two_byte_command);
33    if (two_byte_command & HCRESET)
34        goto fail_label;
35
36    reg = uhci->look_for(uhci, Frame_List_Base_Address);
37    reg->write(reg, &uhci->fba);
38
39    reg = uhci->look_for(uhci, Frame_Number);
40    two_byte_command = 0x0000;
41    reg->write(reg, &two_byte_command);
42
43    reg = uhci->look_for(uhci, Start_of_Frame);
44    one_byte_command = SOF;
45    reg->write(reg, &one_byte_command);
46
47    reg = uhci->look_for(uhci, Usb_Interrupt);
48    two_byte_command = TMOUT_CRC | COMPLETE;
49    reg->write(reg, &two_byte_command);
50
51    reg = uhci->look_for(uhci, Usb_Status);
52    two_byte_command = 0xFFFF;
53    reg->write(reg, &two_byte_command);
54
55    reg = uhci->look_for(uhci, Usb_Command);
56    two_byte_command = MAXP | RS | CF;
57    reg->write(reg, &two_byte_command);
58
59    uhci->controller_port_configuration(uhci);
60    return 1;
61 fail_label:
62     return -1;
63 }

```

Following the successful configuration of the UHCI, the subsequent procedural phase involves the configuration of the downstream ports inherent to the UHCI. Notably, this port configuration presently remains confined to a capacity of 2 ports. Consequently, any port beyond this threshold will remain unconfigured.

```

UHCI.c
1 void controller_port_configuration(_UHCI* uhci){
2     for(int port = 0; port < 2; port++){
3         uint16_t status;
4         if ((status = ((UsbController*)uhci)->reset_port((UsbController*)uhci,
5             port+1)) > 0) {
6             Register_Type r_type = (port+1 == 1) ? Port1_Status : Port2_Status;
7             Register* port_reg = uhci->look_for(uhci, r_type);
8
9             uint16_t port_bits = *((uint16_t*)port_reg->raw_data);
10            if(port_bits & CONNECT)
11                uhci->create_dev(uhci, status, port+1, m);
12        }
13    }
14 }

```

An integral aspect of the port configuration procedure entails the reset operation specific to each individual port.

```

UHCI.c
1 uint16_t uhci_reset_port(UsbController *controller, uint8_t port) {
2     Register_Type r = (port == 1 ? Port1_Status : Port2_Status);
3
4     reg = uhci->look_for(uhci, r);
5
6     reg->read(reg, &val);
7     lval = val | RESET;
8     reg->write(reg, &lval);
9
10    mdelay(USB_TDRSTR);
11
12    reg->read(reg, &val);
13    lval = val & 0xFCB1;
14    reg->write(reg, &lval);
15
16    udelay(300);
17
18    reg->read(reg, &val);
19    lval = val | CON_CHANGE | CONNECT;
20    reg->write(reg, &lval);
21
22    lval = val | CONNECT | ENA;
23    reg->write(reg, &lval);
24
25    udelay(50);
26
27    reg->read(reg, &val);
28    lval = val | 0x000F;
29    reg->write(reg, &lval);
30
31    mdelay(50);
32
33    reg->read(reg, &val);
34
35    return (val & ENA);
36 }

```

In the process of configuring the UHCI, a crucial requirement involves specifying the start address of the frame list. Consequently, this function is invoked during the initialization

phase of the `UHCI`. By doing so, these essential parameters become readily available upon invocation of the `UHCI` configuration process, facilitating their seamless utilization within the configuration.

UHCI.c

C

```

1 QH *request_frames(_UHCI *uhci) {
2     QH **physical_addresses = (QH **)(m->allocateKernelMemory_c(
3         m, SKELETON_SIZE * sizeof(uint32_t *), 0));
4     uint32_t *frame_list_address =
5         (uint32_t *) (m->mapIO(m, sizeof(uint32_t) * TOTAL_FRAMES, 1));
6     // build bulk qh
7     QH *bulk_qh = (QH *) (map_io_buffer + map_io_offset);
8     bulk_qh->flags =
9         PRIORITY_QH_8 | QH_FLAG_END | QH_FLAG_IS_MQH | QH_FLAG_TYPE_BULK;
10    bulk_qh->pyhsicalQHLP = QH_TERMINATE | QH_SELECT;
11    bulk_qh->pyhsicalQHEP = QH_TERMINATE | TD_SELECT;
12    bulk_qh->parent = 0;
13    // build control qh
14    QH *control_qh = (QH *) (map_io_buffer + map_io_offset);
15    bulk_qh->parent = (uint32_t)(uintptr_t)(m->getPhysicalAddress(m, control_qh));
16
17    control_qh->pyhsicalQHLP =
18        ((uint32_t)(uintptr_t)(m->getPhysicalAddress(m, bulk_qh))) | QH_SELECT;
19    control_qh->pyhsicalQHEP = QH_TERMINATE | TD_SELECT;
20    control_qh->parent = 0;
21    control_qh->flags =
22        PRIORITY_QH_8 | QH_FLAG_END | QH_FLAG_IS_MQH | QH_FLAG_TYPE_CONTROL;
23    // build interrupt qh
24    for (int frame_number = 0; frame_number < TOTAL_FRAMES; frame_number++) {
25        for (int j = FRAME_SCHEDULE.size - 1; j >= 0; j--) {
26            if ((frame_number + 1) % FRAME_SCHEDULE.qh[j] == 0) {
27                if (frame_number + 1 == FRAME_SCHEDULE.qh[j]) {
28                    current = (QH *) (map_io_buffer + map_io_offset);
29                    if (j == 0) {
30                        current->pyhsicalQHLP =
31                            ((uint32_t)(uintptr_t)(m->getPhysicalAddress(m, control_qh))) |
32                            QH_SELECT;
33                        control_qh->parent =
34                            (uint32_t)(uintptr_t)(m->getPhysicalAddress(m, current));
35                        frame_list_address[frame_number] =
36                            (uint32_t)(uintptr_t)(physical_addresses[j]);
37                        frame_list_address[frame_number] |= QH_SELECT;
38                    } else {
39                        current->pyhsicalQHLP =
40                            (uint32_t)(uintptr_t)(physical_addresses[j - 1]);
41                        current->pyhsicalQHLP |= QH_SELECT;
42                        child->parent =
43                            (uint32_t)(uintptr_t)(m->getPhysicalAddress(m, current));
44                        frame_list_address[frame_number] =
45                            (uint32_t)(uintptr_t)(m->getPhysicalAddress(m, current));
46                        frame_list_address[frame_number] |= QH_SELECT;
47                    }
48                    current->pyhsicalQHEP = TD_SELECT | QH_TERMINATE;
49                    current->flags = PRIORITY_QH_8 | QH_FLAG_END | QH_FLAG_IS_MQH |
50                        QH_FLAG_TYPE_INTERRUPT;
51                    child = current;
52                    break;
53                }
54                frame_list_address[frame_number] =
55                    (uint32_t)(uintptr_t)(physical_addresses[j]);
56                frame_list_address[frame_number] |= QH_SELECT;
57                break;
58            }
59        }
60    }
61    uhci->fba =
62        (uint32_t)(uintptr_t)(m->getPhysicalAddress(m, frame_list_address));
63    return current;
64 }

```

The initialization of the `UHCI` necessitates the construction of the start of the frame list, achieved through the allocation of 4 kilobytes (KB) of contiguous memory. This

allocation corresponds to 1024×4 bytes, which collectively accommodate the frame list. Upon construction of the schedule, the physical address denoting the start address of the frame list is stored within the `UHCI`. Consequently, the completion of this initialization process yields the return of the last frame schedule entry, thereby facilitating subsequent traversal of the skeletal structure through the linkage inherent within these QHs.

Preceding functions primarily concentrated on the initialization of the `UHCI` component and the subsequent configuration of the `UHCI` itself. The ensuing functionalities pivot towards elucidating the mechanics underlying data transfers within the UHCI driver.

Subsequently, the forthcoming function within the system is tasked with incorporating transfers, represented as QHs, into the schedule. These transfers are subsequently processed by the UHCI, thereby advancing the data transmission process.

```

UHCI.c C
1 void insert_queue(_UHCI *uhci, QH *new_qh, uint16_t priority, enum QH_HEADS v) {
2     int8_t offset = (int8_t)v;
3     QH *current = uhci->qh_entry;
4
5     while (offset > 0) {
6         if ((current->flags & QH_FLAG_IS_MQH)) {
7             offset--;
8         }
9         physical_addr = (current->pyhsicalQHLP & QH_ADDRESS_MASK);
10        current = m->getVirtualAddress(m, physical_addr);
11    }
12
13    while (((current->flags & QH_FLAG_END_MASK) == QH_FLAG_IN) &&
14           (((current->flags & PRIORITY_QH_MASK) >> 1) >= priority)) {
15        current =
16            (QH *)m->getVirtualAddress(m, current->pyhsicalQHLP & QH_ADDRESS_MASK);
17    }
18
19    if ((current->pyhsicalQHLP & QH_TERMINATE) ==
20        QH_TERMINATE) { // erase bit which got set prior
21        current->pyhsicalQHLP &= 0xFFFFF0;
22        new_qh->pyhsicalQHLP = QH_TERMINATE | QH_SELECT;
23    } else {
24        new_qh->pyhsicalQHLP = current->pyhsicalQHLP;
25    }
26
27    if ((current->flags & QH_FLAG_END_MASK) == QH_FLAG_END) {
28        new_qh->flags |= QH_FLAG_END;
29        current->flags &= 0xFFFFF0;
30    } else { // if not last element in chain update parent pointer
31        QH *c =
32            (QH *)m->getVirtualAddress(m, current->pyhsicalQHLP & QH_ADDRESS_MASK);
33        c->parent = (uint32_t)(uintptr_t)(m->getPhysicalAddress(m, new_qh));
34    }
35    new_qh->parent = (uint32_t)(uintptr_t)(m->getPhysicalAddress(m, current));
36    new_qh->flags |= QH_FLAG_IS_QH | QH_FLAG_IN | priority;
37
38    // adds to skeleton -> gets executed by controller !
39    current->pyhsicalQHLP =
40        ((uint32_t)(uintptr_t)(m->getPhysicalAddress(m, new_qh))) | QH_SELECT;
41 }

```

The function comprises parameters including `QH* new_qh`, denoting the QH intended for insertion into the schedule. The parameter `uint16_t priority` signifies the relative importance of the transfer, influencing its scheduling priority in comparison to other transfers within the same schedule. Lastly, the parameter `enum QH_HEADS v` indicates the target schedule to which the QH is to be appended.

Subsequently, the system integrates a function designed to extract a transfer from the schedule. This function exclusively employs the parameter `QH* qh` to specify the particular QH slated for removal from the schedule.

```

UHCI.c
1 void remove_queue(_UHCI *uhci, QH *qh) {
2     QH *parent =
3         (QH *)memory_service->getVirtualAddress(memory_service, qh->parent);
4     QH *child = (QH *)memory_service->getVirtualAddress(
5         memory_service, qh->pyhsicalQHLP & QH_ADDRESS_MASK);
6
7     parent->pyhsicalQHLP = qh->pyhsicalQHLP;
8
9     QH *mqh = qh;
10
11     while ((mqh->flags & QH_FLAG_IS_MASK) == QH_FLAG_IS_QH) {
12         mqh = (QH *)memory_service->getVirtualAddress(memory_service, mqh->parent);
13     }
14
15     if ((qh->flags & QH_FLAG_END_MASK) == QH_FLAG_END) {
16         parent->flags |= QH_FLAG_END;
17     }
18
19     if ((qh->flags & QH_FLAG_END_MASK) == QH_FLAG_IN) { // if qh is not last in
20         chain update child pointer to
21         // parent
22         child->parent = qh->parent;
23     }
24 }

```

The subsequent function is designed with the objective of generating a valid packet, an essential prerequisite for all individual transfers internally. Within this function, the attributes of TDs are configured in accordance with the specified parameters.

```

UHCI.c
1  UsbPacket *create_USB_Packet(_UHCI *uhci, UsbDev *dev, UsbPacket *prev,
2                               TokenValues token, int8_t speed, void *data,
3                               int last_packet, uint8_t flags) {
4      TD *td = uhci->get_free_td(uhci);
5
6      td->bufferPointer = 0;
7      td->control_x_status = 0;
8      td->pyhsicalLinkPointer = 0;
9      td->token = 0;
10
11     if (prev != (void *)0) {
12         prev->internalTD->pyhsicalLinkPointer |=
13             ((uint32_t)(uintptr_t)(m->getPhysicalAddress(m, td)));
14     }
15
16     td->pyhsicalLinkPointer = DEPTH_BREADTH_SELECT | TD_SELECT;
17
18     td->control_x_status = (speed == LOW_SPEED ? (0x1 << LS) : (0x0 << LS));
19     td->control_x_status |= (0x1 << ACTIVE);
20     td->control_x_status |= (0x3 << C_ERR); // 3 Fehlversuche max
21
22     if (last_packet) {
23         td->pyhsicalLinkPointer |= QH_TERMINATE;
24
25         if((dev->state == CONFIGURED_STATE) && (flags != BULK_INITIAL_STATE) &&
26            (flags != CONTROL_INITIAL_STATE)){
27             td->control_x_status |= (0x1 << IOC); // set interrupt bit only for last td
28             in qh !
29         }
30     }
31
32     td->token = ((token.max_len - 1) & 0x7FF) << TD_MAX_LENGTH;
33     td->token |= (token.toggle << TD_DATA_TOGGLE);
34     td->token |= (token.endpoint << TD_ENDPOINT);
35     td->token |= (token.address << TD_DEVICE_ADDRESS);
36     td->token |= (token.packet_type);
37
38     td->bufferPointer = (uint32_t)(uintptr_t)(m->getPhysicalAddress(m, data));
39
40     UsbPacket *packet =
41         (UsbPacket *)m->allocateKernelMemory_c(m, sizeof(UsbPacket), 0);
42     packet->internalTD = td;
43
44     return packet;
45 }

```

This function encompasses several parameters to facilitate its operation. Firstly, `UsbDev* dev` serves the purpose of retrieving the device's state, thereby enabling verification of the ability to set the `IOC` within the `control_x_status`. Following this, `UsbPacket* prev` signifies the presence of a preceding packet, aiding in the chaining of TDs. The `TokenValues token` field delineates the token values to be populated within the TDs token field. Subsequently, the `int8_t speed` parameter indicates whether the device operates at low speed or high speed, facilitating the appropriate setting of the relevant bit within the TD's control field. The `void * data` parameter functions as a generic pointer, representing the buffer designated for the transfer. The final two parameters, `int last_packet` and `uint8_t flags`, primarily determine whether the `IOC` bit should be set. Additionally, `int last_packet` plays a pivotal role in terminating the chaining of TDs. Upon execution, the function yields a packet as its outcome.

In the process of constructing the requisite transfers within the `UHCI`, dedicated build functions are provided, tailored to accommodate specific transfer types. Notably, given

the analogous nature of bulk and interrupt transfers, the build function designed for one transfer type is repurposed to suit the requirements of both transfer types.

```

UHCI.c
1  UsbTransfer *build_interrupt_or_bulk(_UHCI *uhci, UsbDev *dev, void *data,
2                                     Endpoint *e, unsigned int len,
3                                     const char *type, uint8_t flags) {
4      UsbTransfer *usb_transfer =
5          (UsbTransfer*)m->allocateKernelMemory_c(m, sizeof(UsbTransfer), 0);
6
7      uint8_t toggle = 0;
8      uint16_t max_len = e->endpoint_desc.wMaxPacketSize & WMAX_PACKET_SIZE_MASK;
9      uint8_t endpoint = e->endpoint_desc.bEndpointAddress & ENDPOINT_MASK;
10
11     // data transaction
12     uint8_t *start = (uint8_t *)data;
13     uint8_t *end = start + len;
14     uint8_t packet_type =
15         (e->endpoint_desc.bEndpointAddress & DIRECTION_IN) ? IN : OUT;
16     int last_packet = 0;
17     while (start < end) { // run through data and send max payload for endpoint
18         last_packet = (start + max_len) >= end ? 1 : 0;
19         if (start + max_len > end) { // send less than maximum payload
20             max_len = end - start;
21         }
22         token = (TokenValues){.max_len = max_len,
23                               .toggle = toggle,
24                               .endpoint = endpoint,
25                               .address = dev->address,
26                               .packet_type = packet_type};
27         prev = uhci->create_USB_Packet(uhci, dev, prev, token, dev->speed, start,
28                                       last_packet, flags);
29         toggle ^= 1;
30         count++;
31         start += max_len;
32
33         data_transaction->entry_packet = prev;
34         data_transaction->next = 0;
35
36         if (prev_transaction == (void *)0) {
37             usb_transfer->entry_transaction = data_transaction;
38         } else {
39             prev_transaction->next = data_transaction;
40         }
41         prev_transaction = data_transaction;
42     }
43
44     usb_transfer->transaction_count = count;
45     usb_transfer->transfer_type = type;
46     return usb_transfer;
47 }

```

Internally, the `create_USB_Packet` function is invoked for each transaction initiated. In terms of its parameters, this build function commences with the `UsbDev* dev`, furnishing pertinent details regarding internal properties. Subsequently, the `void *data` parameter, constituting a generic pointer, encapsulates the buffer from which data is either read or written. The `Endpoint* e` parameter designates the specific endpoint facilitating communication. Notably, a null pointer within this parameter signifies the utilization of endpoint zero. Lastly, the `unsigned int len` field specifies the quantity of bytes earmarked for transfer. Upon the culmination of transaction assembly, the constructed transfer encompassing all transactions is returned.

In order to facilitate the assembly of control transfers, the `UHCI` incorporates support

for control transfer construction. Diverging from the interrupt/bulk build methodology, the control transfer build process encompasses additional stages, notably including setup, data, and status. These distinct stages are meticulously delineated within the implementation to avert any potential loss or oversight.

```

UHCI.c
1  UsbTransfer *build_control(_UHCI *uhci, UsbDev *dev,
2                               UsbDeviceRequest *device_request, void *data,
3                               Endpoint *e, uint8_t flags) {
4      unsigned int endpoint =
5          (e == ((void *)0) ? 0
6           : e->endpoint_desc.bEndpointAddress & ENDPOINT_MASK);
7
8      UsbTransfer *usb_transfer =
9          (UsbTransfer*)m->allocateKernelMemory_c(m, sizeof(UsbTransfer), 0);
10
11      uint8_t toggle = 0;
12      uint16_t total_bytes_to_transfer = device_request->wLength;
13      uint16_t max_len = 8; // max payload of setup transaction
14
15      // setup transaction
16      UsbTransaction *setup_transaction =
17          (UsbTransaction *)m->allocateKernelMemory_c(m, sizeof(UsbTransaction), 0);
18      usb_transfer->entry_transaction = setup_transaction;
19      setup_transaction->transaction_type = SETUP_TRANSACTION;
20      token = (TokenValues){.max_len = max_len,
21                           .toggle = toggle,
22                           .endpoint = endpoint,
23                           .address = dev->address,
24                           .packet_type = SETUP};
25      head = uhci->create_USB_Packet(uhci, dev, 0, token, dev->speed,
26                                   device_request, 0, flags);
27      setup_transaction->entry_packet = head;
28      count++;
29      prev = head;
30      prev_transaction = setup_transaction;
31
32      max_len = dev->max_packet_size;
33      // data transaction
34      uint8_t *start = (uint8_t *)data;
35      uint8_t *end = start + total_bytes_to_transfer;
36      uint8_t packet_type =
37          (device_request->bmRequestType & DEVICE_TO_HOST) ? IN : OUT;
38      while (start < end) { // run through data and send max payload for endpoint
39          data_transaction = (UsbTransaction *)m->allocateKernelMemory_c(
40              m, sizeof(UsbTransaction), 0);
41          data_transaction->transaction_type =
42              (packet_type == OUT) ? DATA_OUT_TRANSACTION : DATA_IN_TRANSACTION;
43          toggle ^= 1;
44          if (start + max_len > end) { // send less than maximum payload
45              max_len = end - start;
46          }
47          token = (TokenValues){.max_len = max_len,
48                               .toggle = toggle,
49                               .endpoint = endpoint,
50                               .address = dev->address,
51                               .packet_type = packet_type};
52          prev =
53              uhci->create_USB_Packet(uhci, dev, prev, token, dev->speed, start, 0, 0);
54          count++;
55          start += max_len;
56          data_transaction->entry_packet = prev;
57          prev_transaction->next = data_transaction;
58          prev_transaction = data_transaction;
59      }

```

The singular parameter unique to this build function, distinct from others, is the

`UsbDeviceRequest* device_request`. This structure encapsulates the actual request pertinent to the transfer. The details of this structure will be expounded upon during the discussion concerning the implementation of the USB device. Notably, since the maximum data to be transferred is contained within this structure, the inclusion of a `len` field becomes redundant and thus is omitted.

The subsequent utilization of the `X_transfer` functions serves the purpose of establishing an intermediary mechanism that systematically orchestrates the control flow of the corresponding transfer. For each such transfer, the associated build function is invoked, thereby formulating the transfer in question. Subsequently, the constructed transfer undergoes insertion into the schedule, thus effectuating its integration into the workflow.

The ensuing section of the implementation will now elucidate the bulk transfer.

```

UHCI.c
1 void bulk_transfer(_UHCI *uhci, UsbDev *dev, void *data, unsigned int len,
2                     uint8_t priority, Endpoint *e,
3                     build_bulk_or_interrupt_transfer build_function,
4                     callback_function callback, uint8_t flags) {
5     QH *qh = uhci->get_free_qh(uhci);
6
7     UsbTransfer *transfer =
8         build_function(uhci, dev, data, e, len, BULK_TRANSFER, flags);
9
10    TD *td = transfer->entry_transaction->entry_packet->internalTD;
11
12    qh->flags = (transfer->transaction_count << QH_FLAG_DEVICE_COUNT_SHIFT) |
13               QH_FLAG_TYPE_BULK;
14
15    qh->physicalQHEP = (uint32_t)(uintptr_t)(m->getPhysicalAddress(m, td));
16
17    if(flags != BULK_INITIAL_STATE){
18        uhci->qh_to_td_map->put_c(uhci->qh_to_td_map, qh, td);
19        uhci->qh_data_map->put_c(uhci->qh_data_map, qh, data);
20        uhci->qh_dev_map->put_c(uhci->qh_dev_map, qh, dev);
21        uhci->callback_map->put_c(uhci->callback_map, qh, callback);
22    }
23
24    uhci->insert_queue(uhci, qh, priority, QH_BULK);
25
26    if(flags == BULK_INITIAL_STATE){
27        uint32_t status = uhci->wait_poll(uhci, qh, UPPER_BOUND_TIME_OUT_MILLIS_BULK);
28
29        callback(dev, status, data);
30
31        uhci->remove_queue(uhci, qh);
32    }
33 }

```

An implementation intricacy within the bulk transfer entails the existence of two distinct states within the function's execution. The initial state is characterized by polling, wherein the function awaits the completion of the transfer. Conversely, the non-initial state denotes the storage of requisite data, crucial for navigating the skeletal structure.

In terms of parameters unique to this function, focus shall be directed solely on those not previously elucidated. Primarily, the `build_bulk_or_interrupt_transfer` parameter specifies the build function employed within this function. Lastly, the `callback_function callback` parameter designates the callback function of the pertinent driver to be invoked upon completion of the transfer.

The subsequent segment of the implementation will now demonstrate the control transfer.

```

UHCI.c
1 void control_transfer(_UHCI *uhci, UsbDev *dev, UsbDeviceRequest *rq,
2                       void *data, uint8_t priority, Endpoint *endpoint,
3                       build_control_transfer build_function,
4                       callback_function callback, uint8_t flags) {
5     QH *qh = uhci->get_free_qh(uhci);
6
7     UsbTransfer *transfer = build_function(uhci, dev, rq, data, endpoint, flags);
8
9     // we already linked them up while creating the packets
10    TD *internalTD = transfer->entry_transaction->entry_packet->internalTD;
11    qh->flags = (transfer->transaction_count << QH_FLAG_DEVICE_COUNT_SHIFT) |
12               QH_FLAG_TYPE_CONTROL;
13    qh->pyhsicalQHEP =
14        ((uint32_t)(uintptr_t)(m->getPhysicalAddress(m, internalTD)));
15
16    if(dev->state == CONFIGURED_STATE && (flags != CONTROL_INITIAL_STATE)){
17        uhci->qh_to_td_map->put_c(uhci->qh_to_td_map, qh, internalTD);
18        uhci->qh_data_map->put_c(uhci->qh_data_map, qh, data);
19        uhci->qh_dev_map->put_c(uhci->qh_dev_map, qh, dev);
20        uhci->callback_map->put_c(uhci->callback_map, qh, callback);
21        uhci->qh_device_request_map->put_c(uhci->qh_device_request_map, qh, rq);
22    }
23
24    uhci->insert_queue(uhci, qh, priority, QH_CTL);
25
26    if (dev->state != CONFIGURED_STATE || flags == CONTROL_INITIAL_STATE) {
27        status = uhci->wait_poll(uhci, qh, UPPER_BOUND_TIME_OUT_MILLIS_CONTROL);
28        callback(dev, status, data);
29        uhci->remove_queue(uhci, qh);
30    }
31 }

```

The principal disparity between this function and the bulk transfer function lies in the utilization of the `build_control_transfer build_function`. Apart from this distinction, it also encompasses two distinct operational states.

Lastly, the subsequent implementation will address the interrupt transfer.

```

UHCI.c
1 void interrupt_transfer(_UHCI *uhci, UsbDev *dev, void *data, unsigned int len,
2                        uint16_t interval, uint8_t priority, Endpoint *e,
3                        build_bulk_or_interrupt_transfer build_function,
4                        callback_function callback) {
5     QH *qh = uhci->get_free_qh(uhci);
6
7     UsbTransfer *transfer =
8         build_function(uhci, dev, data, e, len, INTERRUPT_TRANSFER, 0);
9
10    TD *td = transfer->entry_transaction->entry_packet->internalTD;
11
12    qh->flags = (transfer->transaction_count << QH_FLAG_DEVICE_COUNT_SHIFT) |
13               QH_FLAG_TYPE_INTERRUPT;
14    qh->pyhsicalQHEP = (uint32_t)(uintptr_t)(m->getPhysicalAddress(m, td));
15
16    uhci->qh_to_td_map->put_c(uhci->qh_to_td_map, qh, td);
17    uhci->qh_data_map->put_c(uhci->qh_data_map, qh, data);
18    uhci->qh_dev_map->put_c(uhci->qh_dev_map, qh, dev);
19    uhci->callback_map->put_c(uhci->callback_map, qh, callback);
20
21    uhci->insert_queue(uhci, qh, priority, (enum QH_HEADS)interval);
22 }

```

The interrupt transfer function operates exclusively in a singular state, thereby precluding the necessity for polling. Within this function, the

`build_control_transfer build_function` is employed to facilitate transfer construction. Additionally, the function incorporates a parameter denoted as `interval`, representing the specific sub schedule within the periodic schedule.

To ascertain whether a transfer necessitates retransmission, the subsequent function is implemented. Within this function, the TDs associated with the QH are recycled for repetitive utilization, thus promoting efficient resource management.

```

UHCI.c
1 unsigned int retransmission(_UHCI *uhci, QH *process_qh) {
2     TD *saved_td = 0;
3     TD *head = 0;
4
5     unsigned int retransmission_occured = 0;
6     uint8_t transfer_type = process_qh->flags & QH_FLAG_TYPE_MASK;
7     if (transfer_type == QH_FLAG_TYPE_INTERRUPT) {
8         saved_td = (TD *)uhci->qh_to_td_map->get_c(uhci->qh_to_td_map, process_qh);
9         head = saved_td;
10        while (saved_td != (void *)0) {
11            saved_td->control_x_status = ((0x1 << LS) & saved_td->control_x_status) |
12                                         ((0x1 << IOC) & saved_td->control_x_status) |
13                                         (0x3 << C_ERR) | (0x1 << ACTIVE);
14            saved_td = (TD *)m->getVirtualAddressTD(m, saved_td->pyhsicalLinkPointer &
15                                                    QH_ADDRESS_MASK);
16        }
17        process_qh->pyhsicalQHEP =
18            (uint32_t)(uintptr_t)(m->getPhysicalAddress(m, head));
19        retransmission_occured = 1;
20    }
21
22    return retransmission_occured;
23 }

```

This function encapsulates the core logic by which the UHCI driver determines the status of ongoing transfers and detects the completion of transfers.

```

UHCI.c
1 void traverse_skeleton(_UHCI *uhci, QH *entry) {
2     if(entry == (void*)0) return;
3     if ((entry->flags & QH_FLAG_IS_MASK) == QH_FLAG_IS_MQH) {
4         return;
5     }
6
7     TD *td = (TD *)mem_service->getVirtualAddressTD(
8         mem_service, entry->pyhsicalQHEP & QH_ADDRESS_MASK);
9
10    callback_function callback =
11        (callback_function)uhci->callback_map->get_c(uhci->callback_map, entry);
12    void *data = (void *)uhci->qh_data_map->get_c(uhci->qh_data_map, entry);
13    UsbDev *dev = (UsbDev *)uhci->qh_dev_map->get_c(uhci->qh_dev_map, entry);
14
15    if (td == (void *)0) { // if null -> transmission was successful
16        callback(dev, S_TRANSFER, data);
17        retransmission_occured = uhci->retransmission(uhci, entry);
18        if (!retransmission_occured) {
19            TD* rcvry = (TD*)uhci->qh_to_td_map->get_c(uhci->qh_to_td_map, entry);
20            uhci->remove_queue(uhci, entry);
21            uhci->remove_td_linkage(uhci, rcvry);
22        }
23    }
24
25    else if (((td->control_x_status >> ACTIVE) & 0x01) == 0x00) {
26        error_mask = uhci->get_status(uhci, td);
27
28        TD* rcvry = (TD*)uhci->qh_to_td_map->get_c(uhci->qh_to_td_map, entry);
29        uhci->remove_queue(uhci, entry);
30        uhci->remove_td_linkage(uhci, rcvry);
31
32        callback(dev, E_TRANSFER | error_mask, data);
33    }
34 }

```

The function is invoked from the `runnable_function_uhci`, iteratively for each QH within the schedule denoted by `QH* entry`. It subsequently triggers the execution of the corresponding callback function registered with the QH, signifying the success or failure of the transfer upon completion.

The final implementation within the `UHCI` pertains to the registration/deregistration of drivers, serving to enlist or withdraw a driver, and to ascertain whether devices conform to the specifications outlined by the driver.

To comprehend this concept, the definitions of the `UsbDriver` and the `UsbDevice_ID` are initially introduced.

The `UsbDriver` component incorporates two abstract functions, mandating implementation by subordinate components. The probing function assesses the compatibility of the driver with the interface, determining potential matches.

```

UsbDriver.h
1 struct UsbDriver{
2     void (*new_usb_driver)(struct UsbDriver* usb_driver, char* name,
3                           struct UsbDevice_ID* entry);
4     int16_t (*probe)(UsbDev* dev, Interface* interface);
5     void (*disconnect)(UsbDev* dev, Interface* interface);
6 };

```

The `UsbDevice_ID` component serves as a repository of specific values utilized in the quest for particular devices. These values are directly accessible to the driver, enabling

the construction of custom search queries for devices. However, an alternative, more streamlined approach involves leveraging one of the predefined macros also accessible to the driver.

```

1  struct UsbDevice_ID{
2      uint16_t idVendor;
3      uint16_t idProduct;
4      uint16_t bcdDevice_low; // minor+subminor part 0x00MN
5      uint16_t bcdDevice_high; // major part 0xJJ00 // format of bcd 0xJJMN , JJ major
6      // low -> minor + sub minor , high major
7
8      uint8_t bDeviceClass;
9      uint8_t bDeviceSubClass;
10     uint8_t bDeviceProtocol;
11
12     uint8_t bInterfaceClass;
13     uint8_t bInterfaceSubClass;
14     uint8_t bInterfaceProtocol;
15 };
16
17 #define USB_DEVICE(vendor, product) \
18 (struct UsbDevice_ID){.idVendor = vendor, .idProduct = product, \
19 .bcdDevice_low = 0xFF, .bcdDevice_high = 0xFF, .bDeviceClass = 0xFF, \
20 .bDeviceSubClass = 0xFF, .bDeviceProtocol = 0xFF, \
21 .bInterfaceClass = 0xFF, .bInterfaceSubClass = 0xFF, .bInterfaceProtocol = 0xFF}
22
23 #define USB_DEVICE_VER(vendor, product, low, high) \
24 (struct UsbDevice_ID){.idVendor = vendor, .idProduct = product, \
25 .bcdDevice_low = low, .bcdDevice_high = high, .bDeviceClass = 0xFF, \
26 .bDeviceSubClass = 0xFF, .bDeviceProtocol = 0xFF, \
27 .bInterfaceClass = 0xFF, .bInterfaceSubClass = 0xFF, .bInterfaceProtocol = 0xFF}
28
29 #define USB_DEVICE_INFO(class_d, subclass, protocol) \
30 (struct UsbDevice_ID){.idVendor = 0xFF, .idProduct = 0xFF, \
31 .bcdDevice_low = 0xFF, .bcdDevice_high = 0xFF, .bDeviceClass = class_d, \
32 .bDeviceSubClass = subclass, .bDeviceProtocol = protocol, \
33 .bInterfaceClass = 0xFF, .bInterfaceSubClass = 0xFF, .bInterfaceProtocol = 0xFF}
34
35 #define USB_INTERFACE_INFO(class_i, subclass, protocol) \
36 (struct UsbDevice_ID){.idVendor = 0xFF, .idProduct = 0xFF, \
37 .bcdDevice_low = 0xFF, .bcdDevice_high = 0xFF, .bDeviceClass = 0xFF, \
38 .bDeviceSubClass = 0xFF, .bDeviceProtocol = 0xFF, \
39 .bInterfaceClass = class_i, .bInterfaceSubClass = subclass, .bInterfaceProtocol = \
40 protocol}
41
42 #define USB_MAJOR_VERSION(bcd) ((bcd & 0xFF00) >> 8)
43 #define USB_MINOR_VERSION(bcd) ((bcd & 0x00FF))
44
45 #define USB_MINOR_PART_VERSION(bcd) ((bcd & 0x00F0) >> 4)
46 #define USB_SUB_MINOR_PART_VERSION(bcd) ((bcd & 0x000F))

```

The initial macro serves the purpose of querying devices possessing a distinct vendor and product identifier.

The subsequent macro extends this functionality by enabling the search for devices with a specified vendor and product identifier, in addition to designated major and minor version specifications.

The third macro facilitates the identification of devices characterized by specific device information, encompassing designated device class, device subclass, and device protocol attributes.

Lastly, the fourth macro facilitates the identification of devices featuring particular interfaces, discerned by designated interface class, interface subclass, and interface protocol specifications.

The registration components' implementation is presented herein, divided into three distinct sections. The initial section delineates the overarching control flow within the implementation. Subsequently, the second section focuses on device verification, while the third section elucidates interface validation.

Commencing with the actual function implementation:

```

UHCI.c
1 int register_driver_uhci(UsbController *controller, UsbDriver *driver) {
2     if (driver->probe == (void *)0) {
3         return -1;
4     };
5     ((UsbController*)uhci)->link_driver_to_controller((UsbController*)uhci, driver);
6     for (int i = 0;
7         device_id_table[i].idVendor != 0 || device_id_table[i].idProduct != 0 ||
8         device_id_table[i].bcdDevice_low != 0 ||
9         device_id_table[i].bcdDevice_high != 0 ||
10        device_id_table[i].bDeviceClass != 0 ||
11        device_id_table[i].bDeviceSubClass != 0 ||
12        device_id_table[i].bDeviceProtocol != 0 ||
13        device_id_table[i].bInterfaceClass != 0 ||
14        device_id_table[i].bInterfaceSubClass != 0 ||
15        device_id_table[i].bInterfaceProtocol != 0;
16        i++) {
17         UsbDevice_ID device_id = device_id_table[i];
18         for (list_element *copy_devs = first_dev; copy_devs != (void *)0;
19             copy_devs = copy_devs->l_e) {
20             :refer to device checks:
21
22             Configuration *config = dev->active_config;
23             int interface_num = config->config_desc.bNumInterfaces;
24
25             for (int k = 0; k < interface_num; k++) {
26                 :refer to interface checks:
27             }
28         }
29     }
30     return (driver_device_match_count == 0
31           ? -1
32           : driver_device_match_count); // no device found if 0
33 }

```


This excerpt entails the verification of devices within the registration process.

```
UHCI.c C
1  UsbDevice_ID device_id = device_id_table[i];
2  UsbDev *dev = (UsbDev *)container_of(copy_devs, UsbDev, l_e);
3  DeviceDescriptor device_desc = dev->device_desc;
4  // 0xFF is default value -> if change check
5  if (device_id.idVendor != 0xFF &&
6      device_id.idVendor != device_desc.idVendor) {
7      continue;
8  }
9  if (device_id.idProduct != 0xFF &&
10     device_id.idProduct != device_desc.idProduct) {
11     continue;
12 }
13 if (device_id.bcdDevice_high != 0xFF &&
14     device_id.bcdDevice_high !=
15         USB_MAJOR_VERSION(device_desc.bcdDevice) &&
16     device_id.bcdDevice_low != USB_MINOR_VERSION(device_desc.bcdDevice)) {
17     continue;
18 }
19 if (device_id.bDeviceClass != 0xFF &&
20     device_id.bDeviceClass != device_desc.bDeviceClass) {
21     continue;
22 }
23 if (device_id.bDeviceSubClass != 0xFF &&
24     device_id.bDeviceSubClass != device_desc.bDeviceSubClass) {
25     continue;
26 }
27 if (device_id.bDeviceProtocol != 0xFF &&
28     device_id.bDeviceProtocol != device_desc.bDeviceProtocol) {
29     continue;
30 }
```

This snippet pertains to the validation of interfaces within the registration procedure.

```

UHCI.c
1 Interface *interface = config->interfaces[k];
2 Alternate_Interface *alt_interface = interface->active_interface;
3 if (device_id.bInterfaceClass != 0xFF &&
4     device_id.bInterfaceClass !=
5     alt_interface->alternate_interface_desc.bInterfaceClass) {
6     continue;
7 }
8 if (device_id.bInterfaceSubClass != 0xFF &&
9     device_id.bInterfaceSubClass !=
10    alt_interface->alternate_interface_desc.bInterfaceSubClass) {
11    continue;
12 }
13 if (device_id.bInterfaceProtocol != 0xFF &&
14     device_id.bInterfaceProtocol !=
15     alt_interface->alternate_interface_desc.bInterfaceProtocol) {
16    continue;
17 }
18
19 int status = dev->usb_dev_interface_lock(dev, interface, driver);
20
21 if ((status == E_INTERFACE_IN_USE) || (status == E_INTERFACE_INV)) {
22    continue;
23 }
24
25 if (driver->probe(dev, interface) < 0) {
26    dev->usb_dev_free_interface(dev, interface);
27 } else {
28    driver_device_match_count++;
29    ((UsbController *)uhci)
30    ->link_device_to_driver((UsbController *)uhci, dev, driver);
31    ((UsbController *)uhci)
32    ->link_driver_to_interface((UsbController *)uhci, driver,
33                              interface);
34    ((UsbController *)uhci)
35    ->interface_dev_map->put_c(
36        ((UsbController *)uhci)->interface_dev_map, interface, dev);
37    driver->dispatcher = uhci->super.dispatcher;
38 }

```

To deregister a driver from the system, the following has to be invoked :

```

UHCI.c
1 int deregister_driver_uhci(UsbController *controller, UsbDriver *driver) {
2     list_element *prev;
3     list_element *l_e = uhci->super.head_driver.l_e;
4
5     UsbDriver *d = (UsbDriver *)container_of(l_e, UsbDriver, l_e);
6
7     if (l_e == (void *)0)
8         return -1;
9
10    if (d == driver) {
11        uhci->super.head_driver.l_e = uhci->super.head_driver.l_e->l_e;
12    } else {
13        prev = l_e;
14        l_e = l_e->l_e;
15        // remove controller linkage
16        while (l_e != (void *)0) {
17            d = (UsbDriver *)container_of(l_e, UsbDriver, l_e);
18            if (driver == d) {
19                prev->l_e = l_e->l_e;
20                break;
21            }
22            l_e = l_e->l_e;
23        }
24    }
25
26    // remove device linkage
27    list_element *l_dev = uhci->super.head_dev.l_e;
28    while (l_dev != (void *)0) {
29        UsbDev *dev = (UsbDev *)container_of(l_dev, UsbDev, l_e);
30        int interface_num = dev->active_config->config_desc.bNumInterfaces;
31        Interface **interfaces = dev->active_config->interfaces;
32        for (int i = 0; i < interface_num; i++) {
33            if (((UsbDriver *)interfaces[i]->driver) == driver) {
34                dev->usb_dev_free_interface(dev, interfaces[i]);
35                return 1;
36            }
37        }
38        l_dev = l_dev->l_e;
39    }
40    return -1;
41 }

```

Having examined the function implementations within the `UHCI`, the focus now shifts to the implementation of the `UsbDevice` component.

To commence, the definitions of all descriptors acquired when requesting the specific descriptor shall be delineated.

```

1 struct DeviceDescriptor {
2     uint8_t bLength;
3     uint8_t bDescriptorType;
4     uint16_t bcdUSB; // Version der Usb-Spezifikation
5     uint8_t bDeviceClass;
6     uint8_t bDeviceSubClass;
7     uint8_t bDeviceProtocol;
8     uint8_t bMaxPacketSize0;
9     uint16_t idVendor;
10    uint16_t idProduct;
11    uint16_t bcdDevice; // gerätedefinierte Version
12    uint8_t iManufacturer;
13    uint8_t iProduct;
14    uint8_t iSerialNumber;
15    uint8_t bNumConfigurations;
16 } __attribute__((packed));
17
18 struct ConfigurationDescriptor {
19     uint8_t bLength;
20     uint8_t bDescriptorType;
21     uint16_t wTotalLength;
22     uint8_t bNumInterfaces;
23     uint8_t bConfigurationValue;
24     uint8_t iConfiguration;
25     uint8_t bmAttributes;
26     uint8_t bMaxPower;
27 } __attribute__((packed));
28
29 struct InterfaceDescriptor {
30     uint8_t bLength;
31     uint8_t bDescriptorType;
32     uint8_t bInterfaceNumber;
33     uint8_t bAlternateSetting;
34     uint8_t bNumEndpoints;
35     uint8_t bInterfaceClass;
36     uint8_t bInterfaceSubClass;
37     uint8_t bInterfaceProtocol;
38     uint8_t iInterface;
39 } __attribute__((packed));
40
41 struct EndpointDescriptor {
42     uint8_t bLength;
43     uint8_t bDescriptorType;
44     uint8_t bEndpointAddress;
45     uint8_t bmAttributes;
46     uint16_t wMaxPacketSize;
47     uint8_t bInterval;
48 } __attribute__((packed));

```

The subsequent definition pertains to the request structure, essential for acquiring specific data from the device when a request is made.

```

1 struct UsbDeviceRequest {
2     uint8_t bmRequestType;
3     uint8_t bRequest;
4     uint16_t wValue;
5     uint16_t wIndex;
6     uint16_t wLength;
7 } __attribute__((packed));

```

To enumerate a device connected to a downstream port, the `new_usb_device` function within the `UsbDev` component is implemented. This function is responsible for acquiring all specific descriptors from the device. Failure to retrieve these descriptors results in the

device not being listed for the `UHCI`.

```

1 void new_usb_device(struct UsbDev *dev, uint8_t speed, uint8_t port,
2                   uint8_t level, uint8_t removable, uint8_t root_port,
3                   uint8_t dev_num, SystemService_C *m, void *controller) {
4     dev->speed = speed;
5     dev->port = port;
6     dev->address = 0;
7     dev->max_packet_size = 8; // default
8     dev->level = level;
9     dev->removable = removable;
10    dev->rootport = root_port;
11    dev->dev_num = dev_num;
12    dev->controller = controller;
13    dev->state = DEFAULT_STATE;
14    dev->init_device_functions(dev);
15
16    if(dev->process_device_descriptor(dev, device_descriptor, 8) == -1){
17        return;
18    }
19    dev->max_packet_size = device_descriptor->bMaxPacketSize0;
20
21    if(dev->set_address(dev, address) == -1){
22        return;
23    }
24    dev->address = address;
25    address++;
26    dev->state = ADDRESS_STATE;
27
28    if(dev->handle_lang(dev, string_buffer) == -1){
29        return;
30    }
31    if(dev->handle_dev(dev, string_buffer, device_descriptor) == -1){
32        return;
33    }
34
35    if(dev->handle_configuration(dev, string_buffer, config_buffer,
36                               config_descriptor, device_descriptor->bNumConfigurations) == -1){
37        return;
38    }
39
40    if(dev->set_configuration(dev,
41                             dev->active_config->config_desc.bConfigurationValue) == -1){
42        return;
43    }
44
45    dev->state = CONFIGURED_STATE;
46
47    ((UsbController*)dev->controller)->add_device((UsbController *)dev->controller,
48    dev);
49 }

```

This function is equipped with several parameters of type `uint8_t`, intended for configuring device properties within the broader USB infrastructure. The parameter `speed` delineates the device's operating speed, distinguishing between low or full speed. The `port` parameter specifies the port, whether situated within a hub or the downstream port of the UHCI. The parameter `level` indicates the tier at which the device resides within the USB hierarchy. Notably, the UHCI itself and its connected devices constitute tier 0, with subsequent tiers incrementing accordingly for downstream devices. The `removable` parameter signifies whether the device attached to the port is removable. For the two root ports, this value defaults to 0xFF. The parameter `root_port` identifies the root port to which the current device is linked, while `dev_num` denotes the device

number, a unique identifier corresponding to the device within the USB system. The parameter `SystemService_C *m` denotes a dependency required for accessing memory operations. Lastly, the parameter `void* controller` serves as a generic pointer to the actual `UsbController`, facilitating controller invocation from the `UsbDev`.

The enumeration process unfolds through the following steps:

- Processing the device descriptor (initial 8 bytes).
- Assigning an address to the device, leading to the transition to the address state.
- Managing device operations.
- Managing device configuration.
- Configuring the device by selecting a specific configuration, transitioning to the configuration state.
- Establishing the link between the device and the controller.

The objective of the subsequent function is to gather comprehensive device related data and verify error free retrieval to facilitate further processing within the call chain within the `new_usb_device` function.

The handling of the device is executed as follows:

```

UsbDevice.c
1 int handle_dev(UsbDev* dev, uint8_t* string_buffer,
2               DeviceDescriptor* device_descriptor){
3     if(dev->process_device_descriptor(dev, device_descriptor,
4                                     sizeof(DeviceDescriptor)) == -1)
5         return -1;
6
7     dev->device_desc = *device_descriptor;
8
9     if (device_descriptor->iManufacturer != 0) {
10        if(dev->process_string_descriptor(
11            dev, string_buffer, device_descriptor->iManufacturer, dev->lang_id,
12            "iManufacturer", 1) == -1)
13            return -1;
14
15        s_len = string_buffer[0];
16
17        if(dev->process_string_descriptor(
18            dev, string_buffer, device_descriptor->iManufacturer, dev->lang_id,
19            "iManufacturer", s_len) == -1)
20            return -1;
21
22        ascii_string = dev->build_string(dev, s_len, string_buffer);
23        dev->manufacturer = ascii_string;
24    }
25
26    if (device_descriptor->iProduct != 0) {
27        if(dev->process_string_descriptor(dev, string_buffer,
28            device_descriptor->iProduct, dev->lang_id, "iProduct", 1) == -1)
29            return -1;
30        s_len = string_buffer[0];
31
32        if(dev->process_string_descriptor(
33            dev, string_buffer, device_descriptor->iProduct, dev->lang_id,
34            "iProduct", s_len) == -1)
35            return -1;
36        ascii_string = dev->build_string(dev, s_len, string_buffer);
37        dev->product = ascii_string;
38    }
39
40    if (device_descriptor->iSerialNumber != 0) {
41        if(dev->process_string_descriptor(
42            dev, string_buffer, device_descriptor->iSerialNumber, dev->lang_id,
43            "iSerialNumber", 1) == -1)
44            return -1;
45        s_len = string_buffer[0];
46
47        if(dev->process_string_descriptor(
48            dev, string_buffer, device_descriptor->iSerialNumber, dev->lang_id,
49            "iSerialNumber", s_len) == -1)
50            return -1;
51        ascii_string = dev->build_string(dev, s_len, string_buffer);
52        dev->serial_number = ascii_string;
53    }
54    return 1;
55 }

```

The objective of the subsequent function is to sequentially traverse through all extant configurations, extracting configuration specific particulars, and invoking the subsequent handling of the interfaces contained within these configurations.

The handling of the configuration unfolds as follows:

```

UsbDevice.c
1 int handle_configuration(UsbDev* dev, uint8_t* string_buffer, uint8_t*
  config_buffer, ConfigurationDescriptor* config_descriptor, uint8_t
  num_configurations){
2   uint8_t config_sel = 0, current_entry = 0;
3   for (int i = 0; i < num_configurations; i++) {
4       // request first 4 bytes of each config desc (wTotalLength)
5       if(dev->process_configuration_descriptor(dev, config_descriptor, i, 4) == -1)
6           return -1;
7       total_len = config_descriptor->wTotalLength;
8       desc_len = config_descriptor->bLength;
9
10      if (total_len > CONFIG_BUFFER_SIZE) {
11          continue;
12      }
13      if(dev->process_whole_configuration(dev, config_buffer, i, total_len) == -1)
14          return -1;
15      ConfigurationDescriptor *config_desc =
16          (ConfigurationDescriptor *)config_buffer;
17      if (config_desc->iConfiguration != 0) {
18          if(dev->process_string_descriptor(dev, string_buffer,
19              config_desc->iConfiguration, dev->lang_id, "iConfiguration", 1) == -1)
20              return -1;
21          s_len = string_buffer[0];
22
23          if(dev->process_string_descriptor(
24              dev, string_buffer, config_desc->iConfiguration, dev->lang_id,
25              "iConfiguration", s_len) == -1)
26              return -1;
27          ascii_string = dev->build_string(dev, s_len, string_buffer);
28          configuration->config_description = ascii_string;
29      }
30      if (!config_sel) {
31          dev->active_config = configuration;
32          config_sel = 1;
33      }
34      start = config_buffer + desc_len;
35      end = config_buffer + total_len;
36
37      configuration->config_desc = *config_desc;
38
39      dev->handle_interface(dev, configuration, string_buffer, start, end,
40          config_desc->bNumInterfaces);
41
42      configurations[current_entry++] = configuration;
43  }
44
45  return 1;
46 }

```

The aim of the final handling function is to systematically iterate through all potential interfaces within the designated configuration and assemble these interfaces.

The handling of the interfaces is executed as follows:

```

UsbDevice.c
1 int handle_interface(UsbDev* dev, Configuration* configuration,
2                     uint8_t* string_buffer, uint8_t* start,
3                     uint8_t* end, uint8_t num_interfaces){
4     while (start < end) {
5         if (*(start + 1) == INTERFACE) {
6             AlternateInterface *alt_interface =
7                 (AlternateInterface *)mem_service->allocateKernelMemory_c(
8                     mem_service, sizeof(AlternateInterface), 0);
9             alt_interface->next = 0;
10
11             InterfaceDescriptor *interface_desc = (InterfaceDescriptor *)start;
12             if (interface_desc->iInterface != 0) {
13                 if(dev->process_string_descriptor(dev, string_buffer,
14                     interface_desc->iInterface, dev->lang_id, "iInterface", 1) == -1)
15                     return -1;
16                 s_len = string_buffer[0];
17
18                 if(dev->process_string_descriptor(
19                     dev, string_buffer, interface_desc->iInterface, dev->lang_id,
20                     "iInterface", s_len) == -1)
21                     return -1;
22                 ascii_string = dev->build_string(dev, s_len, string_buffer);
23             }
24             alt_interface->alternate_interface_desc = *interface_desc;
25
26             if (interface_desc->bInterfaceNumber == prev_interface_number) {
27                 prev->next = alt_interface;
28                 prev = prev->next;
29             }
30             else {
31                 Interface *interface =
32                     (Interface *)mem_service->allocateKernelMemory_c(
33                         mem_service, sizeof(Interface), 0);
34                 interface->active = 0;
35                 interface->driver = 0;
36                 interface->active_interface = alt_interface;
37                 interface->alternate_interfaces = alt_interface;
38                 interface->interface_description = ascii_string;
39                 prev = alt_interface;
40                 interfaces[interface_num++] = interface;
41             }
42             endpoint_num = 0;
43             prev_interface_number = interface_desc->bInterfaceNumber;
44             alt_interface->endpoints =
45                 (Endpoint **)mem_service->allocateKernelMemory_c(mem_service,
46                     sizeof(struct Endpoint *) * interface_desc->bNumEndpoints, 0);
47         }
48         else if (*(start + 1) == ENDPOINT) {
49             EndpointDescriptor *endpoint_desc = (EndpointDescriptor *)start;
50             Endpoint *endpoint = (Endpoint *)mem_service->allocateKernelMemory_c(
51                 mem_service, sizeof(Endpoint), 0);
52             endpoint->endpoint_desc = *endpoint_desc;
53             prev->endpoints[endpoint_num++] = endpoint;
54         }
55         start += *(start);
56     }
57     return 1;
58 }

```

The foundational level accessed from the handling functions within the device component encompasses the actual processing. Internally, these functions primarily rely on two key functions, enumerated as follows:

```

1 void request_build(UsbDev *dev, UsbDeviceRequest *device_request,
2                   int8_t rq_type, int8_t rq, int16_t value_high,
3                   int16_t value_low, int16_t shift, int16_t index,
4                   int16_t len) {
5     device_request->bmRequestType = rq_type;
6     device_request->bRequest = rq;
7     device_request->wValue = value_high << shift | value_low;
8     device_request->wIndex = index;
9     device_request->wLength = len;
10 }

```

When executing a particular request, the `request_build` function is initially invoked to populate the structure with pertinent values tailored to the request at hand.

Following the construction of this specialized request, the subsequent step involves submitting the request. This is achieved by calling the `request` function, which subsequently calls upon the controller to execute the requisite transfer, typically a control transfer.

```

1 void request(UsbDev *dev, UsbDeviceRequest *device_request, void *data,
2             uint8_t priority, Endpoint *endpoint, callback_function callback,
3             uint8_t flags) {
4     ((UsbController*)dev->controller)->control_entry_point(dev, device_request,
5                                                             data, priority, endpoint, callback, flags);
6 }

```

Now that the fundamentals of constructing and submitting requests to the controller have been established, the focus shifts to examining the processing mechanism.

Commencing with the processing of the device descriptor, the procedure unfolds as follows:

```

1 int process_device_descriptor(UsbDev *dev, DeviceDescriptor *device_descriptor,
2                               unsigned int len) {
3     UsbDeviceRequest *request = dev->get_free_device_request(dev);
4
5     if (request == (void *)0)
6         return -1;
7
8     dev->request_build(dev, request, DEVICE_TO_HOST, GET_DESCRIPTOR, DEVICE, 0, 8,
9                       0, len);
10    dev->request(dev, request, device_descriptor, PRIORITY_QH_8, 0,
11               &request_callback, CONTROL_INITIAL_STATE);
12
13    if (dev->error_while_transferring) {
14        return -1;
15    }
16
17    return 1;
18 }

```

Next in line is the processing of the configuration descriptor, which is outlined as follows:

```

UsbDevice.c
1 int process_configuration_descriptor(UsbDev *dev,
2                                     ConfigurationDescriptor *config_descriptor,
3                                     uint16_t configuration_index,
4                                     unsigned int len) {
5     DeviceDescriptor device_descriptor = dev->device_desc;
6     if (configuration_index > device_descriptor.bNumConfigurations)
7         return -1;
8     if (dev->state == DEFAULT_STATE)
9         return -1;
10
11     UsbDeviceRequest *request = dev->get_free_device_request(dev);
12
13     if (request == (void *)0)
14         return -1;
15
16     dev->request_build(dev, request, DEVICE_TO_HOST, GET_DESCRIPTOR,
17                       CONFIGURATION, configuration_index, 8, 0, len);
18     dev->request(dev, request, config_descriptor, PRIORITY_QH_8, 0,
19                &request_callback, CONTROL_INITIAL_STATE);
20
21     if (dev->error_while_transferring) {
22         return -1;
23     }
24
25     return 1;
26 }

```

Subsequently, the processing of the string descriptor is as follows:

```

UsbDevice.c
1 int process_string_descriptor(UsbDev *dev, uint8_t *string_buffer,
2                               uint16_t index, uint16_t lang_id, char *s,
3                               unsigned int len) {
4     if (lang_id != 0 && lang_id != dev->lang_id)
5         return -1;
6     if (s == (void *)0)
7         return -1;
8
9     UsbDeviceRequest *request = dev->get_free_device_request(dev);
10
11     if (request == (void *)0)
12         return -1;
13
14     dev->request_build(dev, request, DEVICE_TO_HOST, GET_DESCRIPTOR, STRING,
15                       index, 8, lang_id, len);
16     dev->request(dev, request, string_buffer, PRIORITY_QH_8, 0,
17                &request_callback, CONTROL_INITIAL_STATE);
18
19     if (dev->error_while_transferring) {
20         return -1;
21     }
22
23     return 1;
24 }

```

Following the initialization process, provided no errors were encountered, the device becomes operational and available for use. Drivers gain access to the device by sending specific requests to a designated endpoint of an interface.

To facilitate this functionality, the `UsbDev` component incorporates a function to verify whether a pipe can be established using the provided `Endpoint`.

```
UsbDevice.c C
1 int is_pipe_buildable(Endpoint *endpoint, unsigned int pipe) {
2     unsigned int type;
3     unsigned int end_point;
4     unsigned int direction;
5     unsigned int reserved;
6
7     unsigned int endpoint_dev;
8     unsigned int type_dev;
9     unsigned int direction_dev;
10
11     type = (pipe & CONTROL_PIPE_MASK) >> 5;
12     end_point = pipe & ENDPOINT_MASK;
13     reserved = (pipe & 0x10) >> 4;
14     direction = pipe & DIRECTION_MASK;
15
16     endpoint_dev = endpoint->endpoint_desc.bEndpointAddress & ENDPOINT_MASK;
17     direction_dev = endpoint->endpoint_desc.bEndpointAddress & DIRECTION_MASK;
18     type_dev = endpoint->endpoint_desc.bmAttributes & TRANSFER_TYPE_MASK;
19
20     return ((type == type_dev) && (end_point == endpoint_dev) &&
21            (direction == direction_dev) && (reserved == 0));
22 }
```

If a pipe cannot be established for all endpoints within the device, it signifies that the endpoint within the interface is unsupported, thereby impeding communication.

3.2.3 Driver System Component Implementation

Concluding the implementation section is the implementation of the actual drivers. To enable registration within the **UHCI** component, a driver must furnish a probing function. Through this function, the driver furnishes information regarding the interfaces it supports and those it does not.

```

KeyBoardDriver.c
1 int16_t probe_key_board(UsbDev *dev, Interface *interface) {
2     KeyBoardDev* kbd_dev = internal_k_driver->get_free_kbd_dev(internal_k_driver);
3
4     if(kbd_dev == 0)
5         return -1;
6
7     Endpoint **endpoints = interface->active_interface->endpoints;
8     InterfaceDescriptor interface_desc =
9         interface->active_interface->alternate_interface_desc;
10    int e = interface_desc.bNumEndpoints;
11    // select only 1 endpoint
12    for (int i = 0; i < e; i++) {
13        if (!(endpoints[i]->endpoint_desc.bEndpointAddress & DIRECTION_IN)) {
14            internal_k_driver->free_kbd_dev(internal_k_driver, kbd_dev);
15            continue;
16        }
17        if (!(endpoints[i]->endpoint_desc.bmAttributes & TRANSFER_TYPE_INTERRUPT)) {
18            internal_k_driver->free_kbd_dev(internal_k_driver, kbd_dev);
19            continue;
20        }
21        if (!kbd_dev->endpoint_addr) {
22            kbd_dev->endpoint_addr =
23                endpoints[i]->endpoint_desc.bEndpointAddress & ENDPOINT_MASK;
24            kbd_dev->usb_dev = dev;
25            kbd_dev->buffer = key_board_buffer;
26            kbd_dev->buffer_size = KEYBOARD_BUFFER_SIZE;
27            kbd_dev->priority = PRIORITY_8;
28            kbd_dev->interface = interface;
29            kbd_dev->interval = endpoints[i]->endpoint_desc.bInterval;
30
31            return 1;
32        }
33    }
34    return -1;
35 }

```

In order to access information upon the completion of a transfer, the driver is required to furnish a callback function upon the initial submission of a particular request. Upon completion of the transfer, this callback function is invoked, enabling the driver's logic to be accessed.

```

KeyBoardDriver.c
1 void callback_key_board(UsbDev* dev, uint32_t status, void *data) {
2     if (status & E_TRANSFER)
3         return;
4     KeyBoardDev* kbd_dev = internal_k_driver->match_kbd_dev(internal_k_driver, dev);
5     if(kbd_dev == 0)
6         return;
7     uint8_t prev_state = kbd_dev->current_led_state;
8
9     uint8_t *buffer = (uint8_t *)data;
10    uint8_t modifiers = *buffer;
11    uint8_t key_code_1 = *(buffer + 2);
12    uint8_t key_code_2 = *(buffer + 3);
13    uint8_t key_code_3 = *(buffer + 4);
14    uint8_t key_code_4 = *(buffer + 5);
15    uint8_t key_code_5 = *(buffer + 6);
16    uint8_t key_code_6 = *(buffer + 7);
17
18    internal_k_driver->look_for_events(internal_k_driver, kbd_dev, &key_code_1,
19                                     &modifiers);
20    internal_k_driver->look_for_events(internal_k_driver, kbd_dev, &key_code_2,
21                                     &modifiers);
22    internal_k_driver->look_for_events(internal_k_driver, kbd_dev, &key_code_3,
23                                     &modifiers);
24    internal_k_driver->look_for_events(internal_k_driver, kbd_dev, &key_code_4,
25                                     &modifiers);
26    internal_k_driver->look_for_events(internal_k_driver, kbd_dev, &key_code_5,
27                                     &modifiers);
28    internal_k_driver->look_for_events(internal_k_driver, kbd_dev, &key_code_6,
29                                     &modifiers);
30    internal_k_driver->look_for_released(internal_k_driver, kbd_dev, buffer + 2,
31                                       &modifiers);
32    if (prev_state != kbd_dev->current_led_state) {
33        internal_k_driver->trigger_led_report(internal_k_driver, kbd_dev);
34    }
35    if (kbd_dev->current_modifier_count == 5) {
36        kbd_dev->current_modifier_state = 0;
37        kbd_dev->current_modifier_count = 0;
38    }
39    kbd_dev->current_modifier_count++;
40
41    kbd_dev->look_up_buffer[0] = key_code_1;
42    kbd_dev->look_up_buffer[1] = key_code_2;
43    kbd_dev->look_up_buffer[2] = key_code_3;
44    kbd_dev->look_up_buffer[3] = key_code_4;
45    kbd_dev->look_up_buffer[4] = key_code_5;
46    kbd_dev->look_up_buffer[5] = key_code_6;
47 }

```

The function parameters include a `UsbDev* dev`, representing the device involved in the communication. The `uint32_t status` parameter encapsulates the status of the transfer. In case of an unsuccessful transfer, this parameter incorporates additional status bits indicative of the specific error encountered. Lastly, the `void* data` parameter serves as a generic pointer denoting the buffer designated for reading from or writing to. Consequently, the buffer encapsulates the actual data in the event of a write operation.

Following are two definitions utilized within the `MassStorageDriver` component. These definitions are employed to initiate the commencement of a bulk transfer and to denote the conclusion of a bulk transfer.

MassStorageDriver.c

C

```
1 struct CommandBlockWrapper {
2     uint32_t signature;
3     uint32_t tag;
4     uint32_t transfer_length;
5     uint8_t flags;
6     uint8_t lun;
7     uint8_t command_len;
8     uint8_t command[COMMAND_LEN]; // command data block consist of 16 bytes
9 } __attribute__((packed));
```

MassStorageDriver.c

C

```
1 struct CommandStatusWrapper {
2     uint32_t signature;
3     uint32_t tag;
4     uint32_t data_residue; // diff in data transfered -> success = 0
5     uint8_t status;
6 } __attribute__((packed));
```

Each driver operates uniquely, and the `MassStorageDriver` is no exception in this regard. Unlike the `KeyBoardDriver`, for instance, the `MassStorageDriver` features a configuration implementation that outlines the necessary steps for seamless device access. This configuration is implemented as follows:

MassStorageDriver.c

C

```

1 int configure_device(MassStorageDriver *driver) {
2     for(int j = 0; j < MAX_DEVICES_PER_USB_DRIVER; j++){
3         if(driver->msd_map[j] == 0) continue;
4         UsbDev *dev = driver->dev[j].usb_dev;
5         Interface *itf = driver->dev[j].interface;
6         MassStorageDev* msd_dev = driver->dev + j;
7
8         if(dev->get_max_logic_unit_numbers(dev, itf, command, &callback_config) == -1)
9             return -1;
10        if(!msd_dev->success_transfer)
11            return -1;
12        unsigned int volumes = command[0];
13        msd_dev->volumes = volumes;
14        msd_dev->mass_storage_volumes =
15            (MassStorageVolume*)mem_service->allocateKernelMemory_c(mem_service,
16                sizeof(MassStorageVolume) * (command[0] + 1), 0);
17        for(int i = 0; i < volumes + 1; i++){
18            msd_dev->mass_storage_volumes[i].new_storage_volume = &new_storage_volume;
19            msd_dev->mass_storage_volumes[i].new_storage_volume(
20                msd_dev->mass_storage_volumes + i, i);
21        }
22
23        for(int i = 0; i < volumes + 1; i++){
24            InquiryCommandData* inquiry_data =
25                (InquiryCommandData*)(command + add_offset);
26            if(driver->send_inquiry(driver, msd_dev, cbw, csw,
27                inquiry_data, i, rs) == -1)
28                return -1;
29            CapacityListHeader* clh =
30                (CapacityListHeader*)(command + add_offset);
31            if(driver->send_read_format_capacities(driver, msd_dev, cbw, csw,
32                clh, i, rs) == -1)
33                return -1;
34            ReadCapacity__32_Bit* rc_32_bit =
35                (ReadCapacity__32_Bit*)(command + add_offset);
36            if(driver->send_read_capacity__32_bit(driver, msd_dev, cbw, csw,
37                rc_32_bit, i, rs) == -1)
38                return -1;
39            if(rc_32_bit->logical_block_address == 0xFFFFFFFF){
40                ReadCapacity__64_Bit* rc_64_bit =
41                    (ReadCapacity__64_Bit*)(command + add_offset);
42                if(driver->send_read_capacity__64_bit(driver, msd_dev, cbw, csw,
43                    rc_64_bit, i, rs) == -1)
44                    return -1;
45                msd_dev->mass_storage_volumes[i].version = READ_CAPACITY_16;
46                msd_dev->mass_storage_volumes[i].rc_64_bit = *rc_64_bit;
47            }
48            else{
49                msd_dev->mass_storage_volumes[i].version = READ_CAPACITY_10;
50                msd_dev->mass_storage_volumes[i].rc_32_bit = *rc_32_bit;
51            }
52            // set volume size
53            msd_dev->mass_storage_volumes[i].volume_size = driver->get_drive_size(
54                driver, msd_dev, i);
55            msd_dev->mass_storage_volumes[i].block_size = driver->get_block_size(
56                driver, msd_dev, i);
57            msd_dev->mass_storage_volumes[i].block_num = driver->get_block_num(
58                driver, msd_dev, i);
59            if(driver->send_read(driver, msd_dev, cbw, csw,
60                msd_dev->buffer, 64, i, 0, 0, &callback_config,
61                BULK_INITIAL_STATE, rs) == -1)
62                return -1;
63        }
64    }
65    return 1;
66 }

```


Another notable divergence from other probing functions lies in the requirement to examine both directions. This necessity arises from the need to commence a transfer using one direction and terminate it using the other. This probing process is elucidated as follows:

```

MassStorageDriver.c
1 int16_t probe_mass_storage(UsbDev *dev, Interface *interface) {
2     Endpoint **endpoints = interface->active_interface->endpoints;
3     InterfaceDescriptor interface_desc =
4         interface->active_interface->alternate_interface_desc;
5     int e = interface_desc.bNumEndpoints;
6     uint8_t* msd_buffer;
7     uint8_t select = 0;
8     MassStorageDev* msd_dev =
9         internal_msdriver->get_free_msdriver(internal_msdriver);
10    if(msd_dev == (void*)0) return -1;
11    for (int i = 0; i < e; i++) {
12        Endpoint* endpoint = endpoints[i];
13        if (!(endpoint->endpoint_desc.bmAttributes & TRANSFER_TYPE_BULK)) {
14            internal_msdriver->free_msdriver(internal_msdriver, msd_dev);
15            continue;
16        }
17        uint8_t direction =
18            endpoint->endpoint_desc.bEndpointAddress & DIRECTION_MASK;
19        if (!msd_dev->bulk_in_endpoint_addr &&
20            (direction == DIRECTION_IN)) {
21            if(msd_dev->buffer == (void*)0){
22                msd_dev->buffer = msd_buffer;
23            }
24            select = 1;
25            msd_dev->bulk_in_endpoint_addr =
26                endpoint->endpoint_desc.bEndpointAddress & ENDPOINT_MASK;
27        } else if (!msd_dev->bulk_out_endpoint_addr &&
28                    (direction == DIRECTION_OUT)) {
29            if(msd_dev->buffer == (void*)0){
30                msd_dev->buffer = msd_buffer;
31            }
32            select = 1;
33            msd_dev->bulk_out_endpoint_addr =
34                endpoint->endpoint_desc.bEndpointAddress & ENDPOINT_MASK;
35        }
36    }
37    if(select){
38        msd_dev->usb_dev = dev;
39        msd_dev->buffer_size = MAX_TRANSFER_BYTES;
40        msd_dev->priority = PRIORITY_8;
41        msd_dev->interface = interface;
42        return 1;
43    }
44    return -1;
45 }
46
47
48 }

```

To send or receive the actual data to or from the device and to verify the success of the corresponding transfer, the integration of callback chaining is employed. This mechanism facilitates the continuation of the transfer by chaining the next callback, or alternatively, aborting the process if an error is encountered. Given that the bulk transfer comprises three sub transfers, a callback is implemented for each transfer to ascertain the success or failure of the specific request. The initial callback in this sequence verifies the successful transfer of the `CommandBlockWrapper` and initiates the subsequent transfer.

```

MassStorageDriver.c
1 void callback_cbw_data_read(UsbDev *dev, uint32_t status, void *data) {
2     if (status & E_TRANSFER)
3         return;
4     uint32_t* id = (uint32_t*)internal_msd_driver->cbw_map->get_c(
5         internal_msd_driver->cbw_map, (CommandBlockWrapper*)data);
6     uint32_t* stored_len = internal_msd_driver->stored_len_map->get_c(
7         internal_msd_driver->stored_len_map, id);
8     uint8_t* target = internal_msd_driver->stored_target_map->get_c(
9         internal_msd_driver->stored_target_map, id);
10    msd_callback callback = internal_msd_driver->callback_map->get_c(
11        internal_msd_driver->callback_map, id);
12    uint8_t* mm = (uint8_t*)m->mapIO(m, sizeof(uint8_t) * *stored_len, 0);
13    MassStorageDev* msd_dev = internal_msd_driver->match_msd_dev(
14        internal_msd_driver, dev);
15    if(mm == (void*)0 || msd_dev == (void*)0){
16        callback(target, 0);
17        return;
18    }
19    if(internal_msd_driver->get_data(internal_msd_driver, msd_dev, mm,
20        *stored_len, 0, &callback_mass_storage) == -1){
21        callback(target, 0);
22    }
23 }

```

The subsequent integrated callback serves to verify the successful transmission of the actual data to the device and triggers the commencement of the subsequent transfer, wherein the callback for the `CommandStatusWrapper` is chained.

```

MassStorageDriver.c
1 void callback_mass_storage(UsbDev *dev, uint32_t status, void *data) {
2     if (status & E_TRANSFER)
3         return;
4     CommandStatusWrapper* csw =
5         internal_msd_driver->get_free_csw(internal_msd_driver);
6     uint32_t* id = (uint32_t*)internal_msd_driver->data_map->get_c(
7         internal_msd_driver->data_map, (uint8_t*)data);
8     uint8_t* target = internal_msd_driver->stored_target_map->get_c(
9         internal_msd_driver->stored_target_map, id);
10    msd_callback callback = internal_msd_driver->callback_map->get_c(
11        internal_msd_driver->callback_map, id);
12    MassStorageDev* msd_dev = internal_msd_driver->match_msd_dev(
13        internal_msd_driver, dev);
14    if(csw == (void*)0 || msd_dev == (void*)0){
15        callback(target, 0);
16        return;
17    }
18    if(internal_msd_driver->retrieve_status(internal_msd_driver, msd_dev, csw,
19        0, &callback_csw) == -1){
20        callback(target, 0);
21        return;
22    }
23 }

```

The final callback is integrated to ascertain whether the `CommandStatusWrapper` was successfully transferred and to determine any overarching errors in the transmission process. Upon successful transfer and absence of errors, the data is written to the buffer.

```

MassStorageDriver.c
1 void callback_csw(UsbDev *dev, uint32_t status, void *data) {
2     if (status & E_TRANSFER)
3         return;
4     uint32_t* id = (uint32_t*)internal_msd_driver->csw_map->get_c(
5         internal_msd_driver->csw_map, (CommandStatusWrapper*)data);
6     uint8_t* target = internal_msd_driver->stored_target_map->get_c(
7         internal_msd_driver->stored_target_map, id);
8     msd_callback callback = internal_msd_driver->callback_map->get_c(
9         internal_msd_driver->callback_map, id);
10    uint32_t* len = internal_msd_driver->stored_len_map->get_c(
11        internal_msd_driver->stored_len_map, id);
12    uint8_t* mm = internal_msd_driver->stored_mem_buffer_map->get_c(
13        internal_msd_driver->stored_mem_buffer_map, id);
14    // if csw->status is erroneous or if data residue is greater than 0 we should
    not fill target buffer
15    if(((CommandStatusWrapper*)data)->status != 0 ||
16        ((CommandStatusWrapper*)data)->data_residue > 0){
17        callback(target, 0);
18    }
19    copy_to_user(target, mm, *len);
20    callback(target, *len);
21 }

```

Another vital configuration implementation entails the actual configuration of the **HubDriver** component, as it necessitates preliminary handling to ensure proper functionality. Its primary objective is to inspect downstream devices connected to its downstream ports and subsequently enumerate them, thereby facilitating the utilization of a wider array of USB devices in general.

The configuration is implemented as follows:

```

HubDriver.c
1 int configure_hub(HubDriver* driver){
2     for(int i = 0; i < MAX_DEVICES_PER_USB_DRIVER; i++){
3         if(driver->hub_map[i] == 0) continue;
4         UsbDev* dev = driver->dev[i].usb_dev;
5         Interface* itf = driver->dev[i].interface;
6         HubDev* hub_dev = driver->dev + i;
7         if(driver->read_hub_descriptor(driver, hub_dev, itf, data) == -1)
8             return -1;
9         if(driver->read_hub_status(driver, hub_dev, itf, data, 0x02) == -1)
10            return -1;
11        uint8_t wait_time = driver->dev[i].hub_desc.potpgt;
12        uint8_t multiplicator_wait_time_ms = 2;
13        uint8_t num_ports = driver->dev[i].hub_desc.num_ports;
14
15        dev->add_downstream(dev, num_ports);
16        for(uint8_t start_port = 0x01; start_port <= num_ports; start_port++){
17            if(driver->set_hub_feature(driver, hub_dev, itf, start_port,
18                PORT_POWER) == -1) return -1;
19            mdelay(wait_time * multiplicator_wait_time_ms);
20
21            if(driver->clear_hub_feature(driver, hub_dev, itf, start_port,
22                C_PORT_OVER_CURRENT) == -1) return -1;
23            if(driver->clear_hub_feature(driver, hub_dev, itf, start_port,
24                C_PORT_CONNECTION) == -1) return -1;
25            if(driver->read_hub_status(driver, hub_dev, itf, data,
26                0x04) == -1) return -1;
27            port_status_field = *((uint16_t*)data);
28            if(port_status_field & device_attached_mask){ // device attached
29                if(driver->set_hub_feature(driver, hub_dev, itf, start_port,
30                    PORT_RESET) == -1) return -1;
31                if(driver->clear_hub_feature(driver, hub_dev, itf, start_port,
32                    C_PORT_RESET) == -1) return -1;
33                uint8_t speed = ((port_status_field & 0x200) >> 9)
34                    == 1 ? LOW_SPEED : FULL_SPEED;
35                uint8_t removable = driver->is_device_removable(
36                    driver, hub_dev, start_port);
37                uint8_t level = dev->level + 1;
38
39                UsbDev* new_dev = m->allocateKernelMemory_c(m, sizeof(UsbDev), 0);
40                new_dev->new_usb_device(new_dev, speed, start_port, level, removable,
41                    dev->rootport, start_device_num, dev->mem_service,
42                    dev->controller);
43                if(new_dev->error_while_transferring){
44                    new_dev->delete_usb_dev(new_dev);
45                }
46                else {
47                    dev->add_downstream_device(dev, new_dev);
48                    start_device_num++;
49                }
50            }
51        }
52    }
53    return 1;
54 }

```

The instantiation of the 'Low Level Keyboard Driver' and the configuration of its requisite parameters are encapsulated within the `initialize` function within the `KernelKbdDriver`. This function acts as an intermediary, orchestrating the invocation of specific logic blocks within the 'Low Level Keyboard Driver' and serving as the entry point for external components seeking to interact with it.

The initialization is implemented here :

```

KernelKbdDriver.cpp
1 int Kernel::Usb::Driver::KernelKbdDriver::initialize(){
2     int k_id = 0, dev_found = 0;
3     Kernel::MemoryService& m =
4     Kernel::System::getService<Kernel::MemoryService>();
5     Kernel::UsbService& u = Kernel::System::getService<Kernel::UsbService>();
6     UsbDevice_ID usbDevs[] = {
7         USB_INTERFACE_INFO(HID_INTERFACE, 0xFF, INTERFACE_PROTOCOL_KDB),
8     };
9     };
10    KeyBoardDriver* kbd_driver =
11    (KeyBoardDriver*)m.allocateKernelMemory(sizeof(KeyBoardDriver), 0);
12    kbd_driver->new_key_board_driver = &new_key_board_driver;
13    kbd_driver->new_key_board_driver(kbd_driver, this->getName(), usbDevs);
14
15    this->driver = kbd_driver;
16
17    dev_found = u.add_driver((UsbDriver*)driver);
18    if(dev_found == -1) return -1;
19
20    KeyBoardListener* key_board_listener =
21    (KeyBoardListener*)m.allocateKernelMemory(sizeof(KeyBoardListener), 0);
22    key_board_listener->new_listener(key_board_listener);
23    k_id = u.register_listener((EventListener*)key_board_listener);
24
25    if(k_id < 0) return -1;
26
27    kbd_driver->super.listener_id = k_id;
28    return 1;
29 }

```

The instantiation of the 'Low Level Mass Storage Driver' and the configuration of its parameters are embedded within the `initialize` function of the `KernelMassStorageDriver`. This function serves as an intermediary, coordinating the invocation of specific logic blocks within the 'Low Level Mass Storage Driver' and acting as the entry point for external components seeking interaction. Furthermore, the `KernelMassStorageDriver` contains its own configuration, wherein the actual configuration of the 'Low Level Mass Storage Driver' occurs.

```

KernelMassStorageDriver.cpp
C++
1 int Kernel::Usb::Driver::KernelMassStorageDriver::initialize(){
2     int dev_found = 0;
3     Kernel::MemoryService& m =
4         Kernel::System::getService<Kernel::MemoryService>();
5     Kernel::UsbService& u = Kernel::System::getService<Kernel::UsbService>();
6     UsbDevice_ID usbDevs[] = {
7         USB_INTERFACE_INFO(MASS_STORAGE_INTERFACE, 0xFF, 0xFF),
8     };
9 };
10 MassStorageDriver* msd_driver =
11     (MassStorageDriver*)m.allocateKernelMemory(sizeof(MassStorageDriver), 0);
12 msd_driver->new_mass_storage_driver(msd_driver, this->getName(), usbDevs);
13 this->driver = msd_driver;
14
15 dev_found = u.add_driver((UsbDriver*)msd_driver);
16 if(dev_found == -1) return -1;
17 if(msd_driver->configure_device(msd_driver) == -1)
18     return -1;
19 return 1;
20 }

```

To acquire specific properties of the device or to furnish settings to the driver, these tasks are accomplished through control I/O.

```

KernelMassStorageDriver.cpp
C++
1 bool Kernel::Usb::Driver::KernelMassStorageDriver::control(uint32_t request,
2     const Util::Array<uint32_t>& parameters, uint8_t minor){
3     MassStorageDev* msd_dev = driver->get_msdev_by_minor(driver, minor);
4     if(msd_dev == (void*)0) return false;
5     switch(request){
6     case GET_SIZE : {
7         if(parameters.length() != 2)
8             Util::Exception::throwException(Util::Exception::INVALID_ARGUMENT
9             uint32_t* user_address = (uint32_t*)(uintptr_t)parameters[0];
10            uint32_t volume = parameters[1];
11            if(!driver->is_valid_volume(driver, msd_dev, volume)){
12                Util::Exception::throwException(Util::Exception::INVALID_ARGUMENT
13                *user_address = driver->get_drive_size(driver, msd_dev, volume);
14                return true;
15            };
16        case SET_CALLBACK : { // 0:15 = magic number ; 16:23 = u_tag
17            if(parameters.length() != 2){
18                Util::Exception::throwException(Util::Exception::INVALID_ARGUMENT
19                msd_callback callback = (msd_callback)(uintptr_t)parameters[0];
20                uint32_t param = parameters[1];
21                uint16_t magic_number = param & 0xFFFF;
22                uint8_t u_tag = (param & 0xFF0000) >> 16;
23                if(driver->set_callback_msdev(driver, callback, magic_number, u_tag) == -1)
24                    return false;
25                return true;
26            };
27        }
28        return false;
29    }

```

As a parameter, an `uint32_t request` is required, specifying the particular request to be handled within the control function. The `const Util::Array<uint32_t>& parameters` delineates all the parameters pertinent to the specific request. Lastly, the `uint8_t minor` is associated with a specific device utilized by the driver.

Examining the `GET_SIZE` request within the implementation, the parameters are extracted as follows:

- `uint32_t* user_address = (uint32_t)(uintptr_t)parameters[0];`
- `uint32_t volume = parameters[1];`

Here, the `user_address` signifies the buffer in user space where the data should be written, while `volume` denotes the specific volume within the device to be accessed.

Regarding the `SET_CALLBACK` request in the implementation, the parameters are extracted as follows:

- `msd_callback callback = (msd_callback)(uintptr_t)parameters[0];`
- `uint32_t param = parameters[1];`
- `uint16_t magic_number = param & 0xFFFF;`
- `uint8_t u_tag = (param & 0xFF0000) >> 16;`

Here, the `callback` indicates the callback in user space to be registered when reading or writing. The `param` serves as a container holding both the `magic_number` and `u_tag`, which should correspond to a unique entry.

The intermediary component responsible for reading data from the device is implemented to facilitate the extraction of parameters into specific values requisite for the MassStorageDriver component.

```

KernelMassStorageDriver.cpp
1  uint64_t Kernel::Usb::Driver::KernelMassStorageDriver::readData(uint8_t
   *targetBuffer, uint64_t start_lba, uint64_t msd_data, uint8_t minor){
2      uint32_t blocks = (msd_data & 0xFFFFFFFF00000000) >> 32;
3      uint8_t volume = msd_data & 0xFF;
4      uint16_t magic = (msd_data & 0xFFFF00) >> 8;
5      uint8_t u_tag = (msd_data & 0xFF000000) >> 24;
6
7      return driver->read_msd(driver, targetBuffer, start_lba, blocks,
8                             magic, u_tag, volume, minor);
9  }

```

The parameters pertinent to elucidation here include the `uint64_t start_lba` and `uint64_t msd_data`. The former parameter denotes the starting logical block address from which reading is initiated. However, the key parameter is the `msd_data`, encapsulating specific properties requisite for the correct functioning of the MassStorageDriver. Within this parameter, various attributes are encoded to convey essential information. Specifically:

- The `blocks` attribute, representing the number of blocks intended for reading, is encoded within the higher 32 bits.
- The `volume` attribute, signifying the volume from which to read, is encoded within the first 8 bits.
- The `magic` attribute, mirroring the identical magic number entered in the `SET_CALLBACK`, is encoded within the 16 bits, shifted 8 bits to the left.

- Lastly, the `u_tag` attribute, corresponding to the same tag entered in the `SET_CALLBACK`, is encoded within the upper 8 bits of the first 32 bits.

In the process of writing data from the device, the intermediary components function similarly to the read operation in terms of parameter extraction.

```
KernelMassStorageDriver.cpp C++
1  uint64_t Kernel::Usb::Driver::KernelMassStorageDriver::writeData(const uint8_t
   *sourceBuffer, uint64_t start_lba, uint64_t msd_data, uint8_t minor){
2  uint32_t blocks = (msd_data & 0xFFFFFFFF00000000) >> 32;
3  uint8_t volume = msd_data & 0xFF;
4  uint16_t magic = (msd_data & 0xFFFF00) >> 8;
5  uint8_t u_tag = (msd_data & 0xFF000000) >> 24;
6
7  return driver->write_msd(driver, (uint8_t*)sourceBuffer, start_lba,
8                          blocks, magic, u_tag, volume, minor);
9 }
```


To visualize the entire bus architecture, the `create_usb_fs` function was incorporated, reconstructing the bus utilizing a virtual file system.

```

1 void Kernel::UsbService::create_usb_fs(){
2     Kernel::FileSystemService &fs =
3         Kernel::System::getService<Kernel::FileSystemService>();
4     list_element *l_e_controller = usb_service_c->head.l_e;
5
6     Filesystem::Memory::MemoryDriver* m_driver = new
7         Filesystem::Memory::MemoryDriver();
8
9     Util::String temp = "/system/usbfs";
10    unsigned int con_n = 0;
11    fs.createDirectory(temp);
12    fs.getFilesystem().mountVirtualDriver(temp, m_driver);
13
14    while(l_e_controller != (void*)0){
15        unsigned int dev_n = 0;
16        unsigned int driver_n = 0;
17        UsbController* controller = usb_service_c->get_controller(
18            usb_service_c, l_e_controller);
19
20        temp = Util::String::format("controller%d", con_n++);
21        Filesystem::Memory::MemoryDirectoryNode* controller_node =
22            new Filesystem::Memory::MemoryDirectoryNode(temp);
23        m_driver->addNode("/", controller_node);
24        Filesystem::Memory::MemoryDirectoryNode* driver_node =
25            new Filesystem::Memory::MemoryDirectoryNode("drivers");
26        Filesystem::Memory::MemoryDirectoryNode* device_node =
27            new Filesystem::Memory::MemoryDirectoryNode("devices");
28        controller_node->addChild(driver_node);
29        controller_node->addChild(device_node);
30
31        list_element* l_e_dev = controller->head_dev.l_e;
32        list_element* l_e_driver = controller->head_driver.l_e;
33
34        while(l_e_dev != (void*)0){
35            UsbDev* dev = usb_service_c->get_dev(usb_service_c, l_e_dev);
36            handle_fs_device(dev, dev_n, device_node);
37
38            l_e_dev = l_e_dev->l_e;
39            dev_n++;
40        }
41
42        while(l_e_driver != (void*)0){
43            UsbDriver* driver = usb_service_c->get_driver(usb_service_c, l_e_driver);
44            handle_fs_driver(driver, driver_n, driver_node, controller);
45
46            l_e_driver = l_e_driver->l_e;
47            driver_n++;
48        }
49        l_e_controller = l_e_controller->l_e;
50    }
51 }
52 }

```

The virtual file system's entry points are the controllers, each of which comprises two sub-directories: one for the connected devices and another for the available drivers compatible with those devices on the bus.

The devices root entry encompasses all devices linked to the controller. Although this approach may not precisely mirror the physical bus topology, it offers the convenience of directly inspecting all connected devices under that controller without navigating through

the actual device hierarchy. Each device node includes its distinct attributes. Additionally, the hub device node within the device root directory comprises further nodes representing its downstream ports.

Similarly, the drivers root entry encompasses all available drivers for the system, along with specific details pertaining to each driver.

3.2.4 Changes in hhuOS

To enable the integration of the USB system into hhuOS, an entry point was established, serving as the foundation from which the entire USB system is constructed within the hhuOS environment.

```

GatesOfHell.cpp C++
1 void GatesOfHell::initializeUsb(){
2     int kbd_status = 0, mouse_status = 0, msd_status = 0;
3     Kernel::System::registerService(
4         Kernel::UsbService::SERVICE_ID, new Kernel::UsbService());
5     Kernel::UsbService& usb_service =
6         Kernel::System::getService<Kernel::UsbService>();
7     Kernel::Usb::Driver::KernelUsbDriver* k_driver =
8         new Kernel::Usb::Driver::KernelKbdDriver("keyboard");
9     Kernel::Usb::Driver::KernelUsbDriver* msd_driver =
10        new Kernel::Usb::Driver::KernelMassStorageDriver("msd");
11     Kernel::Usb::Driver::KernelUsbDriver* m_driver =
12        new Kernel::Usb::Driver::KernelMouseDriver("mouse");
13     Kernel::Usb::Driver::KernelUsbDriver* hub_driver =
14        new Kernel::Usb::Driver::KernelHubDriver("hub");
15
16     hub_driver->initialize();
17     kbd_status = k_driver->initialize();
18     mouse_status = m_driver->initialize();
19     msd_status = msd_driver->initialize();
20
21     usb_service.create_usb_fs();
22
23     if(kbd_status != -1)
24         k_driver->create_usb_dev_node();
25     if(mouse_status != -1)
26         m_driver->create_usb_dev_node();
27     if(msd_status != -1)
28         msd_driver->create_usb_dev_node();
29 }

```

In the system architecture, support for various peripherals has been seamlessly integrated. This includes facilitation for the keyboard through the utilization of the driver entry point denoted as `KernelKbdDriver`, accommodation for the mass storage device via the driver entry point designated as `KernelMassStorageDriver`, provision for the mouse utilizing the driver entry point labeled as `KernelMouseDriver`, and finally, incorporation of support for the hub via the driver entry point termed `KernelHubDriver`.

Another requisite adjustment necessitated for functionality involved the parsing of specific USB codes.

```

KeyDecoder.cpp
1 bool Util::Io::KeyDecoder::parseUsbCodes(uint8_t code, uint8_t type){
2     if(type == KEY_MODIFIER){
3         if(code & LEFT_CONTROL_MASK)
4             currentKey.setCtrlLeft(true);
5         else if((code & LEFT_SHIFT_MASK) || (code & RIGHT_SHIFT_MASK))
6             currentKey.setShift(true);
7         else if(code & LEFT_ALT_MASK)
8             currentKey.setAltLeft(true);
9         else if(code & RIGHT_CONTROL_MASK)
10            currentKey.setCtrlRight(true);
11        else if(code & RIGHT_ALT_MASK)
12            currentKey.setAltRight(true);
13    }
14    else if(type == KEY_TYPE){
15        if(code == KEY_PRESSED)
16            currentKey.setPressed(true);
17        else if(code == KEY_RELEASED){
18            currentKey.setPressed(false);
19        }
20    }
21    else if(type == KEY_CODE){
22        parseAsciiUsb(code);
23        currentKey.setScanCode(code);
24    }
25    return true;
26 }

```

In total, three distinct types characterize USB codes, attributed to the manner in which key events are generated and stored within their respective local buffers. Leveraging these varied types, key properties are configured utilizing internal codes to discern their specific attributes.

An additional modification was implemented within the **Terminal** component, originally designed to interface with the PS2 keyboard, characterized by its distinct scan codes and operational methodology differing from that of the USB keyboard. To address this discrepancy, the following implementation was integrated:

```

Terminal.cpp C++
1 void Terminal::KeyboardRunnable::run() {
2     // just pick first kbd for now
3     auto keyboardStream = Io::FileInputStream("/device/keyboard0");
4     auto keyDecoder = Io::KeyDecoder();
5
6     int16_t modifiers = keyboardStream.read();
7     int16_t keytype    = keyboardStream.read();
8     int16_t keycode    = keyboardStream.read();
9     for(;;){
10        keyDecoder.parseUsbCodes(modifiers, (uint8_t)Io::KeyDecoder::KEY_MODIFIER);
11        keyDecoder.parseUsbCodes(keytype,   (uint8_t)Io::KeyDecoder::KEY_TYPE);
12        keyDecoder.parseUsbCodes(keycode,   (uint8_t)Io::KeyDecoder::KEY_CODE);
13
14        auto key = keyDecoder.getCurrentKey();
15        if(key.isPressed()){
16            auto c = key.getAscii();
17            if (c == 0) {
18                auto scan_code = key.getScanCode();
19                if(scan_code == KEY_UP_ARROW) :out:
20                else if(scan_code == KEY_DOWN_ARROW) :out:
21                else if(scan_code == KEY_RIGHT_ARROW) :out:
22                else if(scan_code == KEY_LEFT_ARROW) :out:
23            }
24            else :out:
25                keyDecoder.defaulting();
26                modifiers = keyboardStream.read();
27                keytype    = keyboardStream.read();
28                keycode    = keyboardStream.read();
29        }
30    }

```

Herein lies the illustration of the necessity to invoke the `read` function thrice to retrieve the complete keyboard event, as each event comprises three essential properties: `modifiers`, `event_value`, and `event_code`.

The final adaptation incorporated into the operational script pertains to the integration of USB functionality into the system, achieved through the utilization of the `-u | -usb` argument during script invocation.

The prescribed format for employing this option is as follows:

```
--usb "type=[x0],vendorid=[x1],productid=[x2],port=[a.b];
```

```
vendorid=[],productid=[],port=[]"
```

Wherein, `x0` designates the type of controller to be supported, with the 'uhci' option being requisite for system utilization. Subsequently, `x1` represents the specific vendor ID of a physical USB device, while `x2` denotes the corresponding product ID. Furthermore, the `[a.b]` specification for the port delineates the hub connection and downstream port allocation. Specifically, 'a' indicates the hub number (1 or 2), while 'b' denotes the downstream port (1-8). This format accommodates the inclusion of multiple devices connected to the controller, facilitating their incorporation and utilization within the system.

The parsing of the provided information is enacted through the following implementation:

```

parse_usb() {
    local UHCI_TYPE="uhci"
    local OHCI_TYPE="ohci"
    local XHCI_TYPE="xhci"
    local EHCI_TYPE="ehci"
    local TYPE_KEY="type"
    local VENDOR_KEY="vendorid"
    local PRODUCT_KEY="productid"
    local HOST_BUS_KEY="hostbus"
    local HOST_ADDRESS_KEY="hostaddr"
    local HOST_PORT_KEY="hostport"
    local PORT_KEY="port"

    local device_msg="-device usb-host"
    local bus_msg="bus="
    local uhci_bus="usb-bus.0"
    local ehci_bus="ehci.0"

    #reset default
    QEMU_USB_ARGS=""

    OLD_IFS=${IFS}
    IFS=":"
    for y in $1; do
        IFS=";"
        read -ra controller_entry <<< ${y}
        for z in "${controller_entry[@]}"; do
            IFS=","
            read -ra device_entry <<< ${z}
            :processing:
            :build:
        done
    done
}

```

The processing of the provided information is encapsulated within this code snippet. Within this segment, a series of checks are implemented to discern the categorization of each entry and extract the pertinent information from it.

```

for x in "${device_entry[@]"; do
    type=$(expr ${x} : "${TYPE_KEY}=\([a-z]\{4\}\)")
    if [[ -n "${type}" && -z "${_type}" ]]; then _type=${type};
    case ${type} in
        ${UHCI_TYPE})
            _type="-usb -device usb-hub,bus=usb-bus.0,port=1"
                    "-device usb-hub,bus=usb-bus.0,port=2";;
        ${EHCI_TYPE})
            _type="-device usb-ehci,id=ehci ";;
        ${XHCI_TYPE})
            _type="-device qemu-xhci ";;
        *)
            printf "Unknown controller type '%s'\n" ${type};exit 1;;
    esac
    QEMU_USB_ARGS="${QEMU_USB_ARGS}${_type}"
fi
vendor=$(expr ${x} : "${VENDOR_KEY}=\(0x[0-9a-fA-F]\{1,4\}\)")
if [[ -n "${vendor}" && -z "${_vendor}" ]]; then _vendor=${vendor}; fi

product=$(expr ${x} : "${PRODUCT_KEY}=\(0x[0-9a-fA-F]\{1,4\}\)")
if [[ -n "${product}" && -z "${_product}" ]]; then _product=${product}; fi

host_bus=$(expr ${x} : "${HOST_BUS_KEY}=\([0-9]\{1,4\}\)")
if [[ -n "${host_bus}" && -z "${_host_bus}" ]]; then _host_bus=${host_bus}; fi

host_address=$(expr ${x} : "${HOST_ADDRESS_KEY}=\([0-9]\{1,4\}\)")
if [[ -n "${host_address}" && -z "${_host_addr}" ]]; then
    _host_addr=${host_address}; fi

host_port=$(expr ${x} : "${HOST_PORT_KEY}=\([a-zA-Z0-9]\{1,4\}\)")
if [[ -n "${host_port}" && -z "${_host_port}" ]]; then _host_port=${host_port}; fi

port=$(expr ${x} : "${PORT_KEY}=\([0-9]\{1\}\.[0,1]\{0,1\}\{0,1\}\)")
if [[ -n "${port}" && -z "${_port}" ]]; then _port=${port}; fi
done

```

By default, the implementation for the 'uhci' type involves the utilization of two hubs interconnected to the downstream ports of the UHCI.

To compile the comprehensive argument list essential for system functionality, the subsequent build snippet is furnished.

```

if [ -z "${_type_}" ]; then printf "no controller type inserted ... exiting.\n";exit
1; fi
QEMU_USB_ARGS="${QEMU_USB_ARGS}${device_msg}"
t=$(grep -e "usb" <<< "${_type_}"); tu=$(grep -e "usb-ehci" <<< "${_type_}")

if [ -n "${t}" ]; then QEMU_USB_ARGS="${QEMU_USB_ARGS},${bus_msg}${uhci_bus}"; fi
if [ -n "${tu}" ]; then QEMU_USB_ARGS="${QEMU_USB_ARGS},${bus_msg}${ehci_bus}"; fi
if [ -n "${_port_}" ]; then QEMU_USB_ARGS="${QEMU_USB_ARGS},${PORT_KEY}=${_port_}";
fi
if [ -n "${_vendor_}" ]; then
    QEMU_USB_ARGS="${QEMU_USB_ARGS},${VENDOR_KEY}=${_vendor_}"; fi
if [ -n "${_product_}" ]; then
    QEMU_USB_ARGS="${QEMU_USB_ARGS},${PRODUCT_KEY}=${_product_}"; fi
if [ -n "${_host_bus_}" ]; then
    QEMU_USB_ARGS="${QEMU_USB_ARGS},${HOST_BUS_KEY}=${_host_bus_}"; fi
if [ -n "${_host_addr_}" ]; then
    QEMU_USB_ARGS="${QEMU_USB_ARGS},${HOST_ADDRESS_KEY}=${_host_addr_}"; fi
if [ -n "${_host_port_}" ]; then
    QEMU_USB_ARGS="${QEMU_USB_ARGS},${HOST_PORT_KEY}=${_host_port_}"; fi

QEMU_USB_ARGS="${QEMU_USB_ARGS} "

```

Within this section, the validation of the configured options is conducted, followed by their integration into the argument list.

Chapter 4

Evaluation

The evaluation section will serve as a critical analysis of the integrated USB system within hhuOS.

4.1 Experimental Setup

The experimental investigation entailed the development of the USB system tailored specifically for integration into hhuOS. The experimental setup was orchestrated within the host environment, Ubuntu 22.04.4 LTS, serving as the foundational platform for experimentation. Within this controlled environment, three distinct USB devices were seamlessly integrated: a Logitech, Inc. M105 Optical Mouse, a LogiLink UDisk flash drive, and a SteelSeries Apex SteelSeries Apex Pro TKL keyboard.

To facilitate virtualization and emulation, QEMU [15], a virtualization software, was employed. Within the QEMU environment, the UHCI was emulated alongside two hubs, leveraging QEMU's USB emulation capabilities [16]. Noteworthy is the integration of host devices into the virtual environment, a feat accomplished through QEMU's pass-through mechanism [16], underscoring the indispensable role played by the Linux host system in ensuring seamless operability.

Throughout the experimental phase, rigorous testing and monitoring protocols were meticulously observed. The QEMU monitor emerged as a crucial tool for real-time observation and evaluation of system operations, providing invaluable insights into the behavior and performance of the integrated USB system.

The initialization of the experimental system is governed by a run script tailored to orchestrate the emulation of the UHCI and two hubs, alongside the seamless integration of host devices through pass-through mechanisms. This intricate orchestration is achieved by configuring the script with the requisite options.

```
--usb type=uhci,vendorid=0x1038,productid=0x1614,port=1.1;
```

```
vendorid=0xabcd,productid=0x1234,port=1.2;
```

```
vendorid=0x046d,productid=0xc077,port=1.3
```


To instigate the experimental environment, the aforementioned script must be invoked with the specified options.

```
./run.sh -bios true -file hhuOS-grub.iso -machine pc
--usb type=uhci,vendorid=0x1038,productid=0x1614,port=1.1;
vendorid=0xabcd,productid=0x1234,port=1.2;
vendorid=0x046d,productid=0xc077,port=1.3
```

4.2 Evaluation of Components

PCI Device Enumeration

The inaugural stage of scrutiny involves the validation of the UHCI's initialization and enumeration as a PCI device. The depicted imagery attests to the successful registration and identification of the UHCI within the PCI device hierarchy, emblematic of proficient hardware initialization.

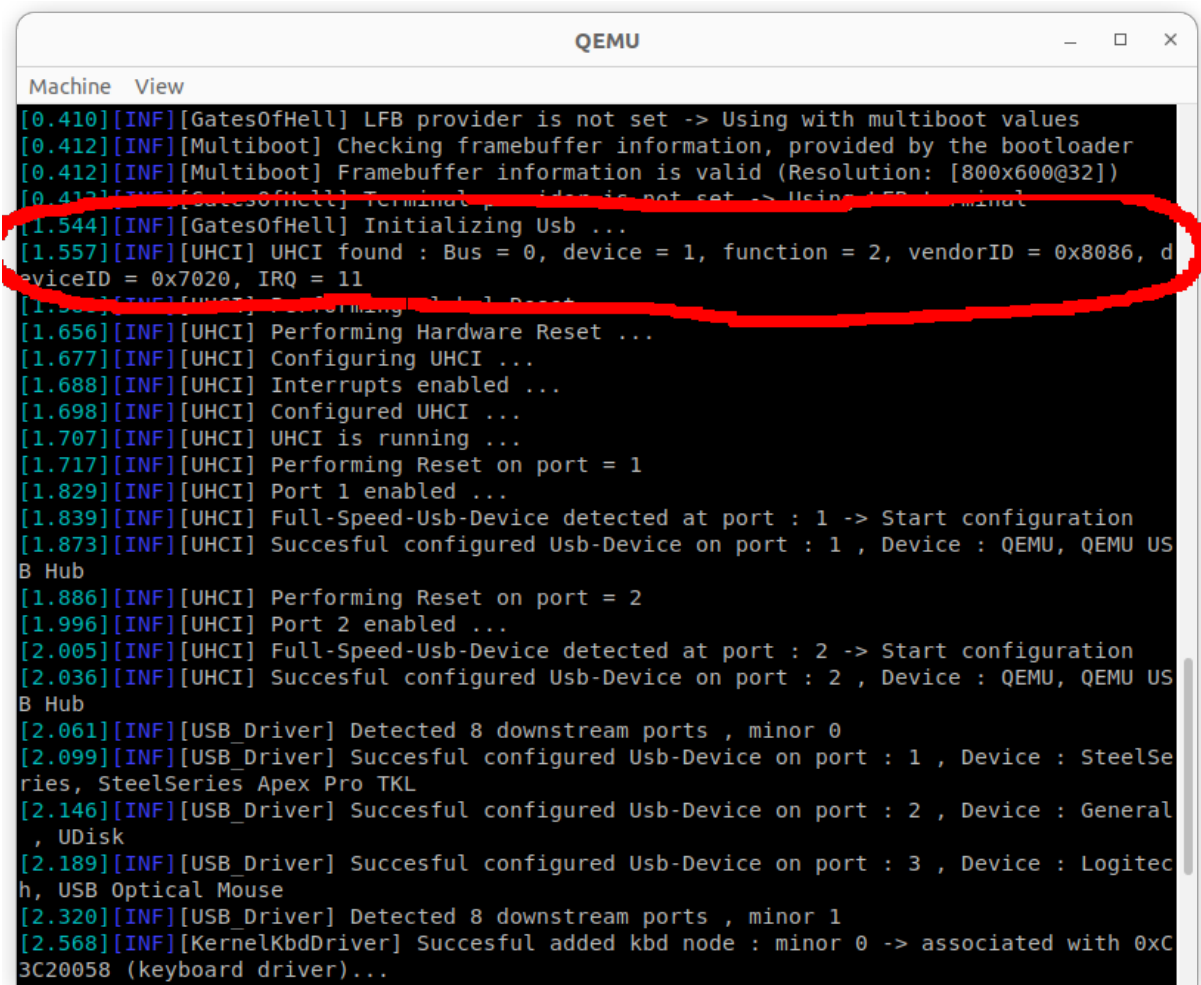


Figure 4.1: UHCI Enumeration as PCI Device

Configuration Steps and Downstream Port Enumeration

The UHCI's configuration, downstream port enumeration, and hub management were evaluated. Operational imagery indicates successful UHCI configuration and downstream device enumeration, notably exemplified by the enumeration of two hub devices. This underscores the driver's adeptness in orchestrating downstream port enumeration routines, facilitating device connectivity and interaction.

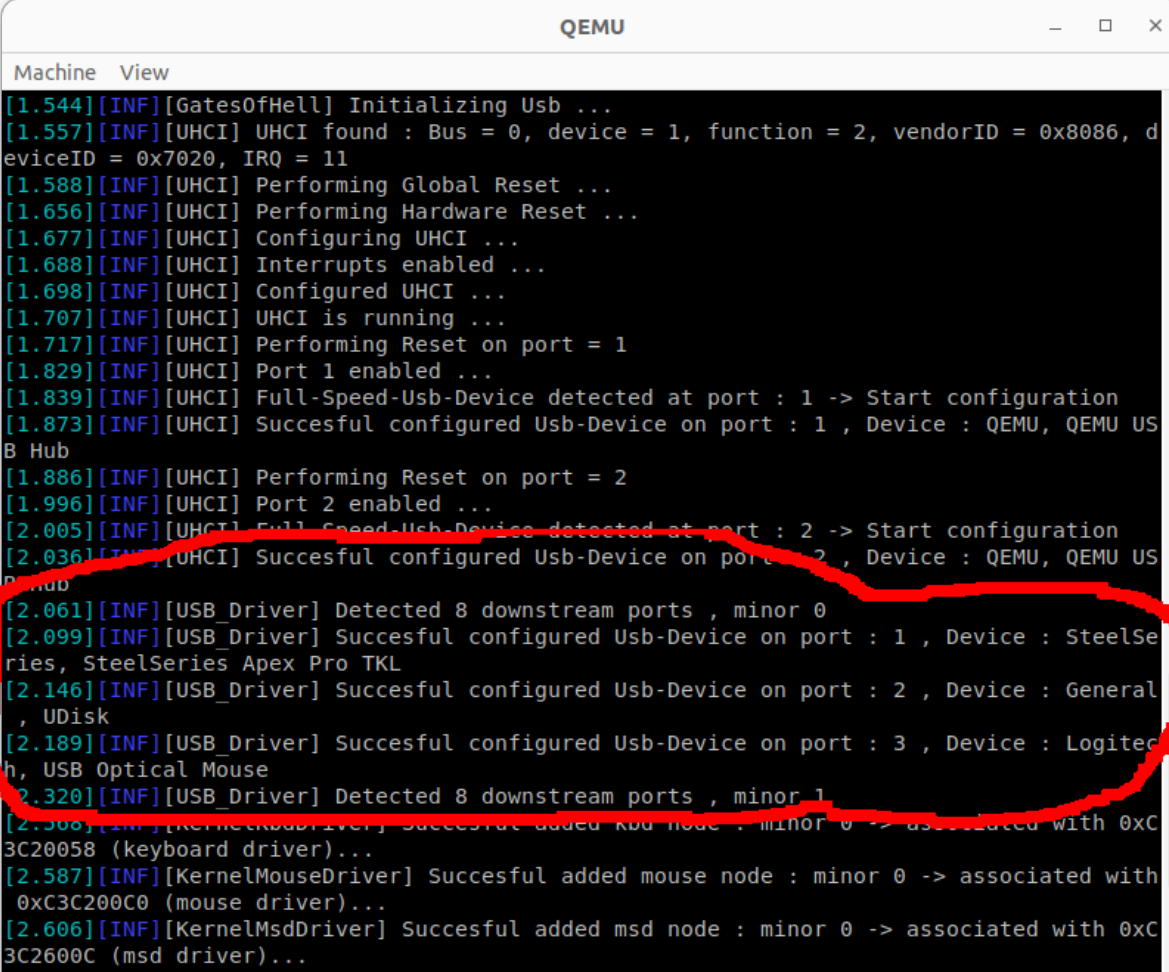
```

Machine View
[0.410][INF][GatesOfHell] LFB provider is not set -> Using with multiboot values
[0.412][INF][Multiboot] Checking framebuffer information, provided by the bootloader
[0.412][INF][Multiboot] Framebuffer information is valid (Resolution: [800x600@32])
[0.413][INF][GatesOfHell] Terminal provider is not set -> Using LFB terminal
[1.544][INF][GatesOfHell] Initializing Usb ...
[1.557][INF][UHCI] UHCI found : Bus = 0, device = 1, function = 2, vendorID = 0x8086, d
iceID = 0x7020, irq = 11
[1.588][INF][UHCI] Performing Global Reset ...
[1.656][INF][UHCI] Performing Hardware Reset ...
[1.677][INF][UHCI] Configuring UHCI ...
[1.688][INF][UHCI] Interrupts enabled ...
[1.698][INF][UHCI] Configured UHCI ...
[1.707][INF][UHCI] UHCI is running ...
[1.717][INF][UHCI] Performing Reset on port = 1
[1.829][INF][UHCI] Port 1 enabled ...
[1.839][INF][UHCI] Full-Speed-Usb-Device detected at port : 1 -> Start configuration
[1.873][INF][UHCI] Successful configured Usb-Device on port : 1 , Device : QEMU, QEMU U
B Hub
[1.886][INF][UHCI] Performing Reset on port = 2
[1.996][INF][UHCI] Port 2 enabled ...
[2.005][INF][UHCI] Full-Speed-Usb-Device detected at port : 2 -> Start configuration
[2.036][INF][UHCI] Successful configured Usb-Device on port : 2 , Device : QEMU, QEMU U
B Hub
[2.001][INF][USB_Driver] Detected 8 downstream ports , minor 0
[2.099][INF][USB_Driver] Successful configured Usb-Device on port : 1 , Device : SteelSe
ries, SteelSeries Apex Pro TKL
[2.146][INF][USB_Driver] Successful configured Usb-Device on port : 2 , Device : General
, UDisk
[2.189][INF][USB_Driver] Successful configured Usb-Device on port : 3 , Device : Logitec
h, USB Optical Mouse
[2.320][INF][USB_Driver] Detected 8 downstream ports , minor 1
[2.568][INF][KernelKbdDriver] Successful added kbd node : minor 0 -> associated with 0xC
3C20058 (keyboard driver)...
```

Figure 4.2: UHCI Configuration & Downstream Port Enumeration

Hub Device Enumeration

Hub device enumeration, crucial for peripheral connectivity, was examined. The imagery provides evidence of the hub's proficiency in enumerating connected devices, including keyboards, mass storage, and mice. This underscores the UHCI driver's capacity to mediate hub device enumeration protocols expediently, enhancing peripheral device interoperability.



```

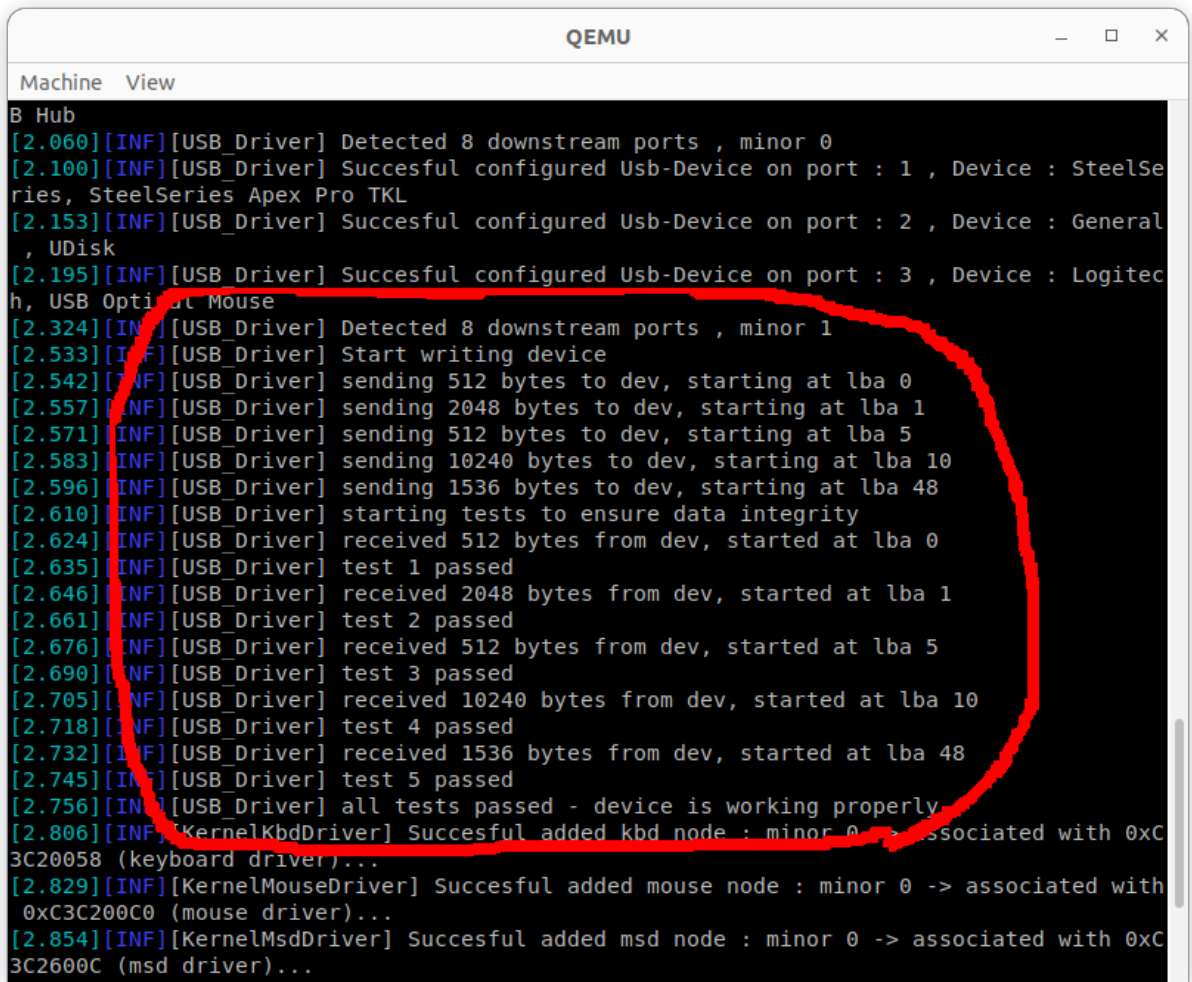
Machine View
[1.544][INF][GatesOfHell] Initializing Usb ...
[1.557][INF][UHCI] UHCI found : Bus = 0, device = 1, function = 2, vendorID = 0x8086, deviceID = 0x7020, IRQ = 11
[1.588][INF][UHCI] Performing Global Reset ...
[1.656][INF][UHCI] Performing Hardware Reset ...
[1.677][INF][UHCI] Configuring UHCI ...
[1.688][INF][UHCI] Interrupts enabled ...
[1.698][INF][UHCI] Configured UHCI ...
[1.707][INF][UHCI] UHCI is running ...
[1.717][INF][UHCI] Performing Reset on port = 1
[1.829][INF][UHCI] Port 1 enabled ...
[1.839][INF][UHCI] Full-Speed-Usb-Device detected at port : 1 -> Start configuration
[1.873][INF][UHCI] Successful configured Usb-Device on port : 1 , Device : QEMU, QEMU USB Hub
[1.886][INF][UHCI] Performing Reset on port = 2
[1.996][INF][UHCI] Port 2 enabled ...
[2.005][INF][UHCI] Full-Speed-Usb-Device detected at port : 2 -> Start configuration
[2.036][INF][UHCI] Successful configured Usb-Device on port : 2 , Device : QEMU, QEMU USB Hub
[2.061][INF][USB_Driver] Detected 8 downstream ports , minor 0
[2.099][INF][USB_Driver] Successful configured Usb-Device on port : 1 , Device : SteelSeries, SteelSeries Apex Pro TKL
[2.146][INF][USB_Driver] Successful configured Usb-Device on port : 2 , Device : General, UDisk
[2.189][INF][USB_Driver] Successful configured Usb-Device on port : 3 , Device : Logitech, USB Optical Mouse
[2.320][INF][USB_Driver] Detected 8 downstream ports , minor 1
[2.508][INF][KernelKeyboardDriver] Successful added keyboard node : minor 0 -> associated with 0xC3C20058 (keyboard driver)...
[2.587][INF][KernelMouseDriver] Successful added mouse node : minor 0 -> associated with 0xC3C200C0 (mouse driver)...
[2.606][INF][KernelMsDriver] Successful added msd node : minor 0 -> associated with 0xC3C2600C (msd driver)...

```

Figure 4.3: Enumeration of Devices Connected to Hubs

Mass Storage Device Test

The functionality of data transfer operations with mass storage devices interfaced with the UHCI was validated. Test sequences demonstrated seamless read/write operations on a mass storage device, affirming the driver's proficiency in facilitating bidirectional data exchanges.

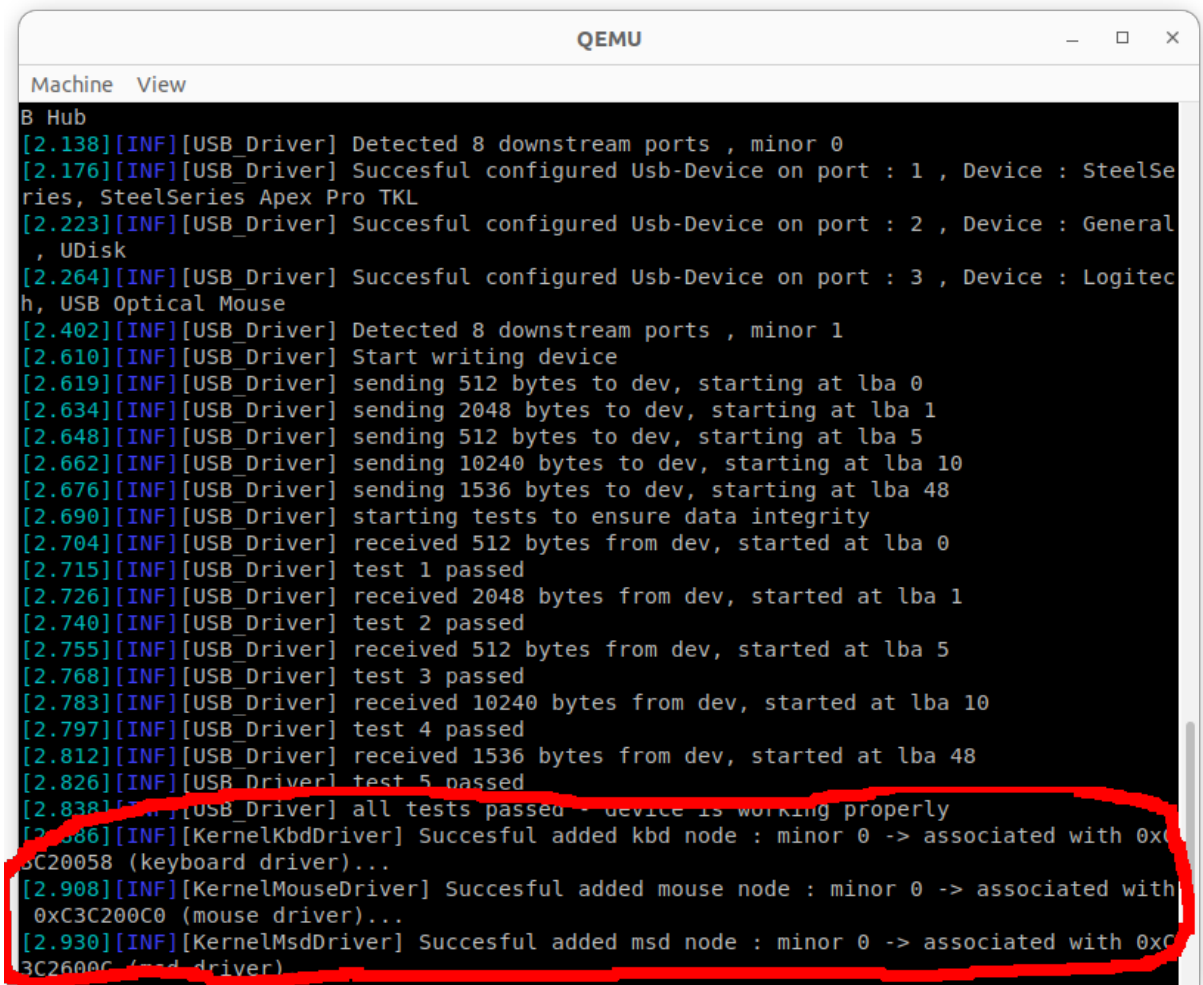


```
Machine View
B Hub
[2.060][INF][USB_Driver] Detected 8 downstream ports , minor 0
[2.100][INF][USB_Driver] Successful configured Usb-Device on port : 1 , Device : SteelSeries, SteelSeries Apex Pro TKL
[2.153][INF][USB_Driver] Successful configured Usb-Device on port : 2 , Device : General, UDisk
[2.195][INF][USB_Driver] Successful configured Usb-Device on port : 3 , Device : Logitech, USB Optical Mouse
[2.324][INF][USB_Driver] Detected 8 downstream ports , minor 1
[2.533][INF][USB_Driver] Start writing device
[2.542][INF][USB_Driver] sending 512 bytes to dev, starting at lba 0
[2.557][INF][USB_Driver] sending 2048 bytes to dev, starting at lba 1
[2.571][INF][USB_Driver] sending 512 bytes to dev, starting at lba 5
[2.583][INF][USB_Driver] sending 10240 bytes to dev, starting at lba 10
[2.596][INF][USB_Driver] sending 1536 bytes to dev, starting at lba 48
[2.610][INF][USB_Driver] starting tests to ensure data integrity
[2.624][INF][USB_Driver] received 512 bytes from dev, started at lba 0
[2.635][INF][USB_Driver] test 1 passed
[2.646][INF][USB_Driver] received 2048 bytes from dev, started at lba 1
[2.661][INF][USB_Driver] test 2 passed
[2.676][INF][USB_Driver] received 512 bytes from dev, started at lba 5
[2.690][INF][USB_Driver] test 3 passed
[2.705][INF][USB_Driver] received 10240 bytes from dev, started at lba 10
[2.718][INF][USB_Driver] test 4 passed
[2.732][INF][USB_Driver] received 1536 bytes from dev, started at lba 48
[2.745][INF][USB_Driver] test 5 passed
[2.756][INF][USB_Driver] all tests passed - device is working properly
[2.806][INF][KernelKbdDriver] Successful added kbd node : minor 0 -> associated with 0xC3C20058 (keyboard driver)...
[2.829][INF][KernelMouseDriver] Successful added mouse node : minor 0 -> associated with 0xC3C200C0 (mouse driver)...
[2.854][INF][KernelMsDriver] Successful added msd node : minor 0 -> associated with 0xC3C2600C (msd driver)...
```

Figure 4.4: Mass Storage Device Test

Driver Binding and Device Recognition

The seamless integration and binding of specific drivers with corresponding peripheral devices were evaluated. Evidentiary vignettes underscore the driver's capacity to discern and associate disparate device types with respective driver modules.



```

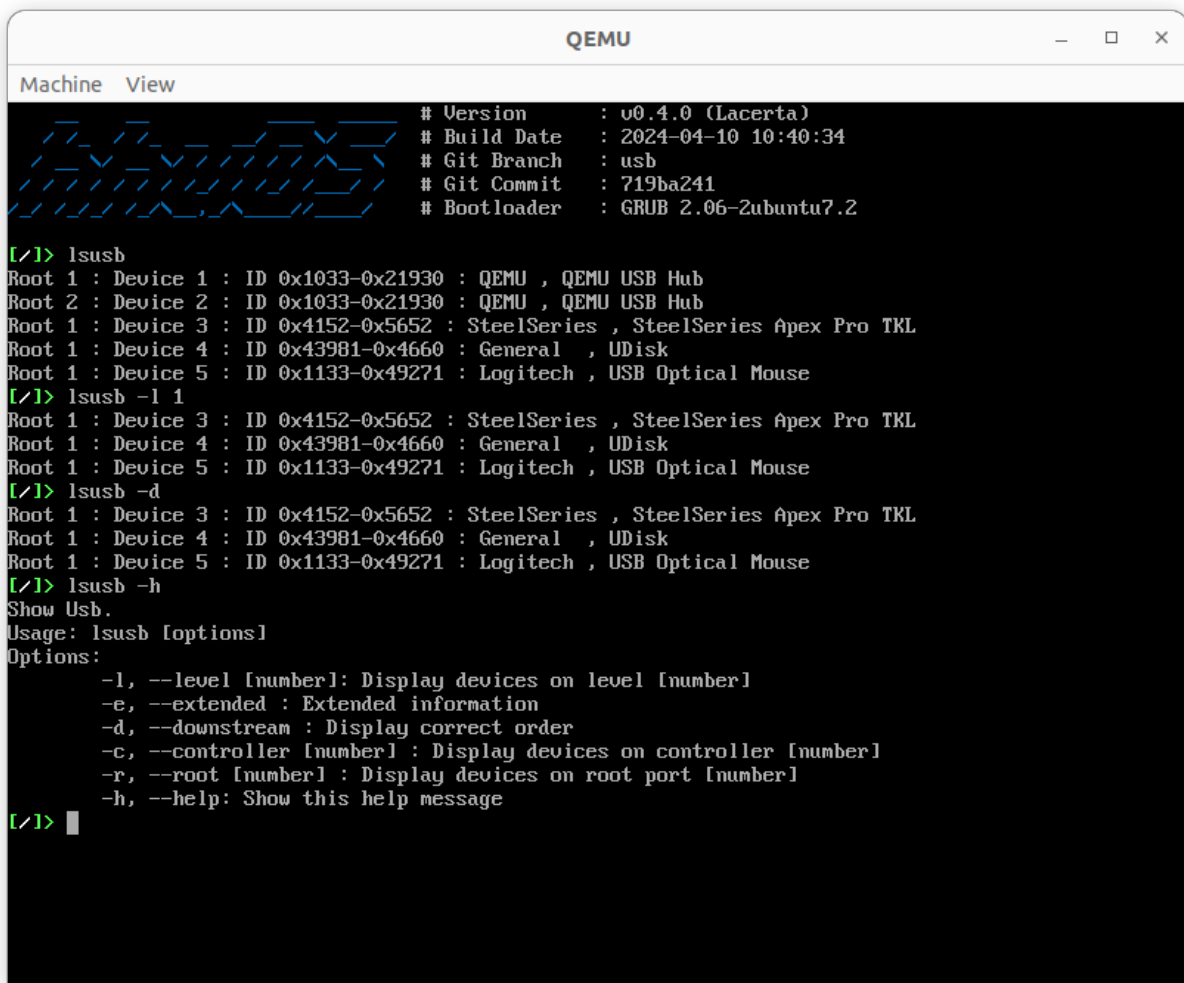
Machine  View
B Hub
[2.138][INF][USB_Driver] Detected 8 downstream ports , minor 0
[2.176][INF][USB_Driver] Successful configured Usb-Device on port : 1 , Device : SteelSeries, SteelSeries Apex Pro TKL
[2.223][INF][USB_Driver] Successful configured Usb-Device on port : 2 , Device : General, UDisk
[2.264][INF][USB_Driver] Successful configured Usb-Device on port : 3 , Device : Logitech, USB Optical Mouse
[2.402][INF][USB_Driver] Detected 8 downstream ports , minor 1
[2.610][INF][USB_Driver] Start writing device
[2.619][INF][USB_Driver] sending 512 bytes to dev, starting at lba 0
[2.634][INF][USB_Driver] sending 2048 bytes to dev, starting at lba 1
[2.648][INF][USB_Driver] sending 512 bytes to dev, starting at lba 5
[2.662][INF][USB_Driver] sending 10240 bytes to dev, starting at lba 10
[2.676][INF][USB_Driver] sending 1536 bytes to dev, starting at lba 48
[2.690][INF][USB_Driver] starting tests to ensure data integrity
[2.704][INF][USB_Driver] received 512 bytes from dev, started at lba 0
[2.715][INF][USB_Driver] test 1 passed
[2.726][INF][USB_Driver] received 2048 bytes from dev, started at lba 1
[2.740][INF][USB_Driver] test 2 passed
[2.755][INF][USB_Driver] received 512 bytes from dev, started at lba 5
[2.768][INF][USB_Driver] test 3 passed
[2.783][INF][USB_Driver] received 10240 bytes from dev, started at lba 10
[2.797][INF][USB_Driver] test 4 passed
[2.812][INF][USB_Driver] received 1536 bytes from dev, started at lba 48
[2.826][INF][USB_Driver] test 5 passed
[2.838][INF][USB_Driver] all tests passed - device is working properly
[2.886][INF][KernelKbdDriver] Successful added kbd node : minor 0 -> associated with 0xC3C20058 (keyboard driver)...
[2.908][INF][KernelMouseDriver] Successful added mouse node : minor 0 -> associated with 0xC3C200C0 (mouse driver)...
[2.930][INF][KernelMsDriver] Successful added msd node : minor 0 -> associated with 0xC3C26000 (msd driver)

```

Figure 4.5: Device x Driver Binding

USB System Information

The USB system information serves as a crucial avenue for gaining insights into the interconnected devices and their respective properties. At the core of this exploration lies the `lsusb` command, an indispensable tool enabling the systematic enumeration and comprehensive display of USB devices currently interfaced with the system. Each device entry produced by `lsusb` encapsulates a wealth of indispensable attributes, encompassing not only the associated identification parameters but also vendor-specific properties and other pertinent information.



```

QEMU
Machine View
# Version      : v0.4.0 (Lacerta)
# Build Date   : 2024-04-10 10:40:34
# Git Branch    : usb
# Git Commit    : 719ba241
# Bootloader    : GRUB 2.06-Zubuntu7.2

[/]> lsusb
Root 1 : Device 1 : ID 0x1033-0x21930 : QEMU , QEMU USB Hub
Root 2 : Device 2 : ID 0x1033-0x21930 : QEMU , QEMU USB Hub
Root 1 : Device 3 : ID 0x4152-0x5652 : SteelSeries , SteelSeries Apex Pro TKL
Root 1 : Device 4 : ID 0x43981-0x4660 : General , UDisk
Root 1 : Device 5 : ID 0x1133-0x49271 : Logitech , USB Optical Mouse
[/]> lsusb -l 1
Root 1 : Device 3 : ID 0x4152-0x5652 : SteelSeries , SteelSeries Apex Pro TKL
Root 1 : Device 4 : ID 0x43981-0x4660 : General , UDisk
Root 1 : Device 5 : ID 0x1133-0x49271 : Logitech , USB Optical Mouse
[/]> lsusb -d
Root 1 : Device 3 : ID 0x4152-0x5652 : SteelSeries , SteelSeries Apex Pro TKL
Root 1 : Device 4 : ID 0x43981-0x4660 : General , UDisk
Root 1 : Device 5 : ID 0x1133-0x49271 : Logitech , USB Optical Mouse
[/]> lsusb -h
Show Usb.
Usage: lsusb [options]
Options:
  -l, --level [number] : Display devices on level [number]
  -e, --extended : Extended information
  -d, --downstream : Display correct order
  -c, --controller [number] : Display devices on controller [number]
  -r, --root [number] : Display devices on root port [number]
  -h, --help : Show this help message
[/]>

```

Figure 4.6: USB Device Listing

Moreover, to delve deeper into the USB topology and extract specific attributes of connected devices, interfacing with `usbfs` emerges as a pivotal mechanism. Through this interface, individuals can navigate the hierarchical structure of USB connections with ease, gaining access to intricate device attributes.

```
QEMU
```

```
Machine View  
# Bootloader : GRUB 2.06-Zubuntu7.2  
  
[/]> cd system/usbfs  
[usbfs]> cd controller0  
[controller0]> ls  
drivers/ devices/  
[controller0]> ls drivers  
driver0/ driver1/ driver2/ driver3/  
[controller0]> tree drivers/driver0  
|--driver0/  
|--name  
|--device0  
|--device1  
[controller0]> cd devices  
[devices]> ls  
dev0/ dev1/ dev2/ dev3/ dev4/  
[devices]> ls dev0  
level root dev_num removable speed port max-packet-size lang_id manufacturer product serial-number bcdUSB bDeviceClass bDeviceSubClass bDeviceProtocol idVendor idProduct bcdDevice bNumConfigurations usb_port0/ usb_port1/ usb_port2/ usb_port3/ usb_port4/ usb_port5/ usb_port6/ usb_port7/ 1.0/  
[devices]> ls dev2  
level root dev_num removable speed port max-packet-size lang_id manufacturer product serial-number bcdUSB bDeviceClass bDeviceSubClass bDeviceProtocol idVendor idProduct bcdDevice bNumConfigurations 1.0/ 1.1/ 1.2/ 1.3/ 1.4/  
[devices]> ls dev4  
level root dev_num removable speed port max-packet-size lang_id manufacturer product serial-number bcdUSB bDeviceClass bDeviceSubClass bDeviceProtocol idVendor idProduct bcdDevice bNumConfigurations 1.0/  
[devices]> ls dev3  
level root dev_num removable speed port max-packet-size lang_id manufacturer product serial-number bcdUSB bDeviceClass bDeviceSubClass bDeviceProtocol idVendor idProduct bcdDevice bNumConfigurations 1.0/  
[devices]> ls dev1  
level root dev_num removable speed port max-packet-size lang_id manufacturer product serial-number bcdUSB bDeviceClass bDeviceSubClass bDeviceProtocol idVendor idProduct bcdDevice bNumConfigurations usb_port0/ usb_port1/ usb_port2/ usb_port3/ usb_port4/ usb_port5/ usb_port6/ usb_port7/ 1.0/  
[devices]>
```

Figure 4.7: USB Filesystem

The evaluation of the USB system showcases its functional efficacy, procedural robustness, and technical proficiency within the operating system.

Chapter 5

Conclusion

The thesis undertakes a thorough exploration of USB system development, with a specific focus on integrating an UHCI driver tailored for hhuOS. The investigation commences with an exhaustive examination of USB architecture, elucidating fundamental principles governing communication flow, notably encompassing control, bulk, and interrupt transfers. While the primary emphasis lies on these transfer modalities within the implementation, the theoretical framework extends to include isochronous transfers, thus enriching the understanding of USB communication dynamics.

Following the initial investigation, the exploration extends into the intricate realm of the USB protocol, delving into the detailed analysis of packet structures. This further inquiry places particular emphasis on dissecting fundamental components such as the packet identifier, address field, endpoint field, and data synchronization mechanisms. These components play pivotal roles in facilitating communication within the USB protocol, elucidating fundamental aspects essential for driver development.

Subsequently, the thesis navigates through the multifaceted terrain of USB device architecture, delineating the structural components of USB devices, device request protocols, and standard descriptors crucial for device interaction. Particular attention is devoted to standard requests such as set address, get descriptor, get configuration, and set configuration, fostering a holistic comprehension of device configuration and management protocols.

Within the scope of UHCI, meticulous attention is directed towards elucidating the initialization process, configuration procedures, port management strategies, interrupt handling mechanisms, and frame scheduling. This exhaustive exploration establishes a robust foundation for subsequent driver implementation endeavors.

In the realm of implementation, strategic design decisions are articulated to modularize the USB system, thereby facilitating the independent integration of USB device drivers without necessitating substantial system modifications. The UHCI driver implementation seamlessly integrates functionalities such as configuration handling, port management, interrupt handling, transfer processing, and device interaction, leveraging insights gleaned from protocol and architecture exploration.

The objective of developing the UHCI driver, as articulated in this thesis, is effectively fulfilled as evidenced by the results of rigorous testing. Initial tests focusing on the UHCI

driver demonstrate successful enumeration as a PCI device, correct configuration, and robust interaction with connected devices. These tests serve as tangible validation of the driver's reliability, affirming the achievement of the stated objective.

Furthermore, comprehensive evaluation, encompassing scenarios such as mass storage device testing, driver bindings, and system information verification, further solidifies the efficacy of the developed USB system within hhuOS. The successful integration of hub, mass storage, mouse, and keyboard drivers underscores the versatility and integration capabilities of the USB system.

In delineating areas for improvement, a pivotal consideration pertains to the reconfiguration of core logic into a modular architecture, potentially manifesting in the instantiation of a discrete 'UsbCore' component, stacked atop the UHCI driver. Additionally, the adoption of a dynamic port detection mechanism to replace the static counterpart is proposed, thereby ensuring scalability.

For future iterations of the USB system, potential extensions encompass the incorporation of support for isochronous transfers, enabling communication with USB devices utilizing isochronous endpoints. Furthermore, the expansion of the driver repertoire to encompass a wider array of devices and the integration of USB hot-plug functionality are identified as avenues for enriching system capabilities.

In summation, this thesis represents a comprehensive investigation into USB architecture and culminates in the delivery of a robust UHCI driver implementation tailored for hhuOS, encapsulating theoretical insights and practical applications.

Bibliography

The following resources were consulted and utilized throughout the preparation of this thesis. These sources provided valuable insights and information contributing to the research and analysis conducted:

- [1] B. Akguel, C. Gesse, F. Ruhland, F. Krakowski, M. Schoettner, *et al.* “Hhuos”. (), [Online]. Available: <https://github.com/hhuOS/hhuOS> (visited on 04/15/2024).
- [2] “Universal serial bus”. (), [Online]. Available: <https://wiki.osdev.org/USB> (visited on 04/15/2024).
- [3] B. D. Lunt, *USB: The Universal Serial Bus*, 3rd. CreateSpace Independent Publishing Platform, 2018, ISBN: 978-1717425362.
- [4] *Universal serial bus specification*, 1998. [Online]. Available: <https://fabiensanglard.net/usbcheat/usb1.1.pdf>.
- [5] J. Corbet, A. Rubini, and G. Kroah-Hartman, “Usb drivers”, in ch. 13.
- [6] B. D. Lunt, “Device enumeration with the uhci”, in ch. 11.
- [7] “Pci”. (), [Online]. Available: <https://wiki.osdev.org/PCI> (visited on 04/15/2024).
- [8] B. D. Lunt, “The pci hardware”, in ch. 2.
- [9] “Universal host controller interface”. (), [Online]. Available: https://www.lowlevel.eu/wiki/Universal_Host_Controller_Interface (visited on 04/15/2024).
- [10] B. D. Lunt, “The uhci hardware”, in ch. 3.
- [11] B. D. Lunt, “The uhci stack”, in ch. 4.
- [12] torvalds *et al.* “Linux”. (), [Online]. Available: <https://github.com/torvalds/linux/blob/master/tools/include/linux/kernel.h> (visited on 04/15/2024).
- [13] torvalds *et al.* “Linux”. (), [Online]. Available: <https://github.com/torvalds/linux/blob/master/include/linux/list.h> (visited on 04/15/2024).
- [14] J. Corbet, A. Rubini, and G. Kroah-Hartman, “Usb drivers”, in ch. 11.
- [15] “About qemu”. (), [Online]. Available: <https://qemu-project.gitlab.io/qemu/about/index.html> (visited on 04/15/2024).
- [16] “Usb emulation”. (), [Online]. Available: <https://qemu-project.gitlab.io/qemu/system/devices/usb.html> (visited on 04/15/2024).

- [17] R. J, *Scsi commands reference manual*, Seagate, 2016. [Online]. Available: <https://www.seagate.com/files/staticfiles/support/docs/manual/Interface%20manuals/100293068j.pdf>.
- [18] B. D. Lunt, “Using the usb hid mouse”, in ch. 15.
- [19] B. D. Lunt, “Using the usb hid keyboard”, in ch. 16.
- [20] B. D. Lunt, “Using the usb mass storage device”, in ch. 17.
- [21] B. D. Lunt, “Using the usb external hub”, in ch. 19.
- [22] “Usb class codes”. (), [Online]. Available: <https://www.usb.org/defined-class-codes> (visited on 04/15/2024).
- [23] “Hid usage tables for usb”. (), [Online]. Available: https://www.usb.org/sites/default/files/hut1_22.pdf (visited on 04/15/2024).
- [24] “Usb hubs”. (), [Online]. Available: https://wiki.osdev.org/USB_Hubs (visited on 04/15/2024).
- [25] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*, 3rd, A. Oram, Ed. O’Reilly Media, 2005.

List of Figures

2.1	USB System Illustration [2]	3
2.2	USB Topology [4, sec. 4.1.1]	4
2.3	USB Communication Flow [4, sec. 5.3]	4
2.4	Successful Data Transmission [2]	10
2.5	Failed Data Transmission [2]	11
2.6	Control Transfer [4, sec. 8.5.2]	12
2.7	Bulk and Interrupt Transfer [4, sec. 8.5.1]	13
2.8	USB Device Structure [5]	14
2.9	UHCI Frame List Structure [11]	35
2.10	UHCI Frame Handling	36
2.11	Example Schedule Overview	41
2.12	Example Schedule Interrupt Transfer Insertion	42
2.13	Example Schedule Control Transfer Insertion	43
2.14	QH Visualization	44
2.15	Schedule Example;Depth First Execution [11]	45
2.16	UHCI Process [11]	46
3.1	Transfer Schedule	50
3.2	System Overview	51
3.3	Interaction Subsystems	52
3.4	Core System Components 1	53
3.5	Core System Components 2	56
3.6	Driver System Components 1	59
3.7	Driver System Components 2	62
3.8	UHCI Initialization Process	65
3.9	Interrupt Transfer Initialization Process	67
3.10	Bulk Transfer Initialization Process	69
3.11	Control Transfer Initialization Process	71
3.12	Driver Registration Process	72
3.13	Event Registration Process	74
3.14	Event Handling	76
3.15	Interrupt Handling	77
3.16	Runnable Workflow	78
3.17	Probe Mechanism	79
3.18	Callback Mechanism	80
3.19	Mass Storage Device Read/Write	82
3.20	Keyboard Read	84
3.21	Hub Driver Initialization	86
3.22	Keyboard Driver Initialization	87

3.23	Mass Storage Driver Initialization	88
3.24	USB Device Node Creation	89
3.25	Controller Configuration	90
3.26	Controller Port Reset	92
3.27	Schedule Construction	93
3.28	Queue Insertion	94
3.29	Queue Removal	95
3.30	Packet Creation	96
3.31	Control Transfer Build	97
3.32	Bulk and Interrupt Transfer Build	98
3.33	Schedule Traversal	99
3.34	Transfer Retransmission	100
3.35	Device Handling	101
3.36	Configuration Handling	102
3.37	Interface Handling	103
3.38	Configuration Descriptor Processing	104
3.39	String Descriptor Processing	105
3.40	Driver Registration	106
3.41	Hub Driver Configuration	108
3.42	Mass Storage Driver Configuration	110
3.43	Keyboard Driver Probing	111
3.44	Keyboard Driver Callback	112
3.45	Mass Storage Driver Callback Chaining	114
3.46	Mass Storage Driver Control I/O	116
4.1	UHCI Enumeration as PCI Device	174
4.2	UHCI Configuration & Downstream Port Enumeration	175
4.3	Enumeration of Devices Connected to Hubs	176
4.4	Mass Storage Device Test	177
4.5	Device x Driver Binding	178
4.6	USB Device Listing	179
4.7	USB Filesystem	180

List of Tables

2.1	Address Field [4, sec. 8.3.2.1]	8
2.2	Endpoint Field [4, sec. 8.3.2.2]	8
2.3	Packet Identifier Types [4, sec. 8.3.1]	9
2.4	USB Device Request [4, sec. 9.3]	15
2.5	Standard Request Codes [4, sec. 9.4]	16
2.6	SET-ADDRESS Request [4, sec. 9.4.6]	17
2.7	Descriptor Types [4, sec. 9.4]	17
2.8	GET-DESCRIPTOR Request [4, sec. 9.4.3]	18
2.9	GET-CONFIGURATION Request [4, sec. 9.4.2]	18
2.10	SET-CONFIGURATION Request [4, sec. 9.4.7]	19
2.11	Device Descriptor [4, sec. 9.6.1]	20
2.12	Configuration Descriptor [4, sec. 9.6.2]	21
2.13	Interface Descriptor [4, sec. 9.6.3]	22
2.14	Endpoint Descriptor [4, sec. 9.6.4]	24
2.15	String Descriptor v1 [4, sec. 9.6.5]	25
2.16	String Descriptor v2 [4, sec. 9.6.5]	26
2.17	GET-DESCRIPTOR(String v1) Request	26
2.18	GET-DESCRIPTOR(String v2) Request	27
2.19	I/O Registers [9]	28
2.20	Command Register [9]	29
2.21	Status Register [9]	30
2.22	Interrupt Register [9]	30
2.23	Frame Number Register [9]	31
2.24	Frame Base Address Register [9]	31
2.25	Start of Frame Register [9]	32
2.26	Port Register [9]	32
2.27	Frame Entry [9]	35
2.28	Queue Head [9]	37
2.29	Queue Head Link Pointer [9]	37
2.30	Queue Element Link Pointer [9]	37
2.31	Transfer Descriptor [9]	38
2.32	Link Pointer [9]	38
2.33	Control and Status [9]	39
2.34	Token [9]	40
3.1	QH Flags	49

Statutory Declaration

I hereby state that I have written this Bachelor's Thesis independently and that I have not used any sources or aids other than those declared. All passages taken from the literature have been marked as such. This thesis has not yet been submitted to any examination authority in the same or a similar form.

Düsseldorf, 22. April 2024

Sandro Jose Leal Miudo