

# USB Host DRIVER

## 구조설계서

2022-02-21

홍길동

이 문서는 소프트웨어 구조 설계자 양성/교육 프로그램에서 양성 인력이 작성된 USB Host Driver의 구조 설계서입니다.



(주) 보이는 소프트웨어 연구소  
조용진(drajin.cho@bosornd.com)

**REVISION HISTORY**

Version	Date	Author	Description
0.1	2022-01-12	홍길동	초기 문서 생성
0.2	2022-01-17	홍길동	Interim 리포트
0.3	2022-01-22	홍길동	Interim 리포트 피드백 반영
0.4	2022-02-06	홍길동	Pre-final 리포트
0.5	2022-02-21	홍길동	Final 리포트

**1. 시스템 개요 ..... 5**

1.1. 시스템 동작 환경 .....	5
1.2. 시스템 정의 및 동작 .....	7
1.2.1. 시스템 정의.....	7
1.2.2. 시스템 동작.....	9
1.3. 시스템 제약 사항 .....	10
1.4. 시스템의 목표.....	10

**2. 요구사항 ..... 11**

2.1. 기능적 요구사항 .....	11
2.2. 비기능적 요구사항 .....	19
2.3. 품질 속성 .....	21

**3. 시스템 구조 ..... 25**

3.1. 시스템 전체 구조 .....	25
3.1.1. 디바이스 연결 동작 .....	27
3.1.2. 데이터 전송 동작.....	27
3.1.3. Health Monitoring 동작.....	28
3.1.4. Resource Monitoring 동작.....	29
3.2. 시스템 세부 구조 .....	30
3.2.1. 시스템 내부 Thread간 통신 구조 .....	31
3.2.2. System Core .....	32
3.2.3. Device Connection Manager.....	35

3.2.4. Transfer Dispatcher .....	38
3.2.5. Transfer Manager.....	41
3.2.6. Health Monitor .....	44
3.2.7. Resource Monitor .....	46
3.3. 시스템 구조 특징 분석.....	48
3.3.1. 강점 .....	48
3.3.2. 단점 및 위험 요인 .....	49
<b>4. 모듈 사양 .....</b>	<b>50</b>
4.1. 전체 모듈 구조 .....	50
4.2. 세부 모듈 구조 .....	51
4.2.1. USB Host Driver Interface.....	54
4.2.2. System Core .....	55
4.2.3. Resource Monitor .....	57
4.2.4. Health Monitor .....	58
4.2.5. Common Service.....	58
4.2.6. Device Control .....	59
4.2.7. Core Driver OS Interface .....	62
4.2.8. Host Controller Driver .....	63
4.2.9. Host Controller Driver OS Interface.....	64
4.2.10. OS Abstraction .....	65
4.3. Work Assignment.....	65

<b>부록</b>	<b>67</b>
<b>A. 도메인 모델</b>	<b>72</b>
<b>B. 품질 시나리오</b>	<b>77</b>
<b>C. 품질 시나리오 분석</b>	<b>79</b>
<b>D. 후보 구조</b>	<b>85</b>
<b>E. 후보 구조 평가</b>	<b>130</b>
<b>F. 최종 구조 설계</b>	<b>151</b>

# 1. 시스템 개요

## 1.1. 시스템 동작 환경

본 과제는 모바일 컴퓨팅 장치에서 USB를 지원하기 위한 USB Host Driver를 설계하는 것을 목표로 한다.

USB는 컴퓨팅 장치 본체(호스트)와 키보드, 마우스, 이동형 저장 장치(ex, USB 메모리)와 같은 주변 장치(디바이스)를 연결하는 표준 프로토콜이다. [그림1]은 모바일 장치를 USB를 이용하여 주변 장치와 연결하는 예이다. USB 시스템에는 하나의 호스트 장치가 있으며, 하나의 호스트 장치에 여러 개의 주변 장치가 USB를 통해 연결될 수 있다. 허브는 USB 시스템에 USB 디바이스 연결을 확장하기 위한 장치로 다수의 USB 장치들이 허브를 통해 USB 장치에 연결될 수 있다. 호스트 장치도 USB 장치들을 연결할 수 있는 허브 기능을 포함하고 있으며, 전체 USB 시스템의 루트 허브 역할을 한다. 그림의 경우 모바일 장치(호스트)에 USB 스토리지와 키보드가 직접 연결되고, 키보드는 허브 역할을 하여 마우스를 연결한 예이다.

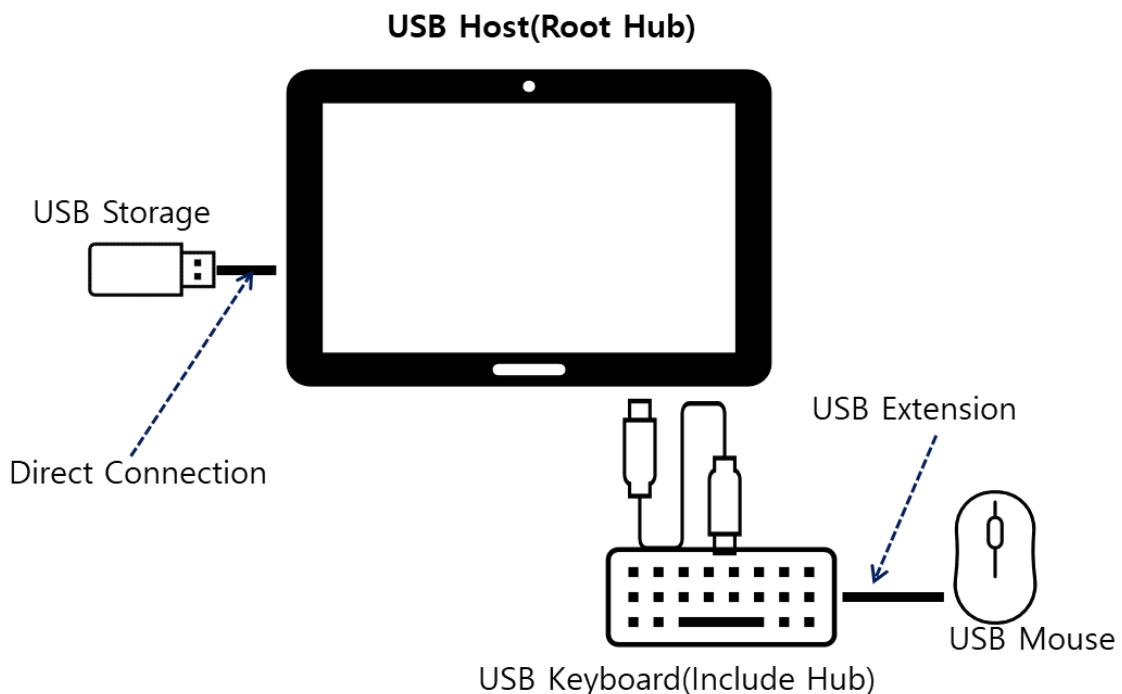


그림 1 USB System 구성 예

[그림2]는 호스트가 USB 디바이스를 사용하기 위한 일반적인 시스템 구성을 나타낸다.



(주) 보이는 소프트웨어 연구소  
조용진(drajin.cho@bosornd.com)

Application(Client SW)은 USB 디바이스의 기능을 사용하는 서비스이며, Host Controller는 USB Protocol을 지원하는 HW 장치이다. USB Host Driver는 Host Controller를 통해 디바이스와 연결을 생성하고 연결된 USB 디바이스와의 통신을 관리하는 기능을 한다. 일반적으로 Application은 USB Host Driver가 제공하는 기능(인터페이스)를 이용하여 디바이스 장치에 요청을 전달/수신하며, USB Host Driver는 Application에서 전달된 요청을 Host Controller를 통해 외부 디바이스 장치로 전달하고, 외부 디바이스 장치에서 수신한 정보를 Application으로 전달하는 역할을 한다. 호스트 시스템에는 하나 이상의 기능을 하는 다양한 형태의 USB 디바이스가 연결될 수 있으며 이를 활용하는 Application도 다양한 형태로 적용될 수 있다. 또한 지원하는 USB Spec에 따라 하나 이상의 Host Controller(H/W)가 적용될 수 있다.

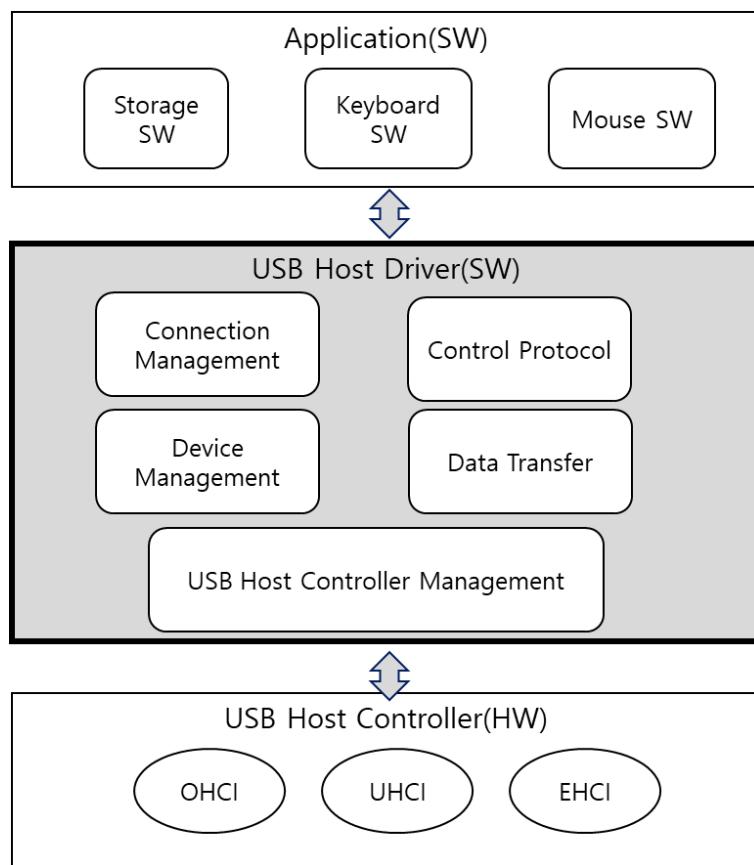


그림 2 USB Host 시스템 구성

본 과제에서는 USB를 통해 USB 저장 장치와 키보드/마우스와 같은 Human Interface 장치를 지원하는 호스트에서 동작하는 USB Host Driver를 설계하는 것을 목표로 한다. USB Host Driver는 [그림2]와 같이 USB를 이용하는 다양한 Application을 지원할 수 있도록 고려되어야 하며, H/W 사양 변경(ex, Host Controller 기능추가)에 대한 확장성을 지원해야 한다. 또한, USB 장치 인식 속도나 USB 장치와

의 통신 속도 등의 성능 측면의 효율 성도 고려하여야 한다.

## 1.2. 시스템 정의 및 동작

### 1.2.1. 시스템 정의

본 과제는 호스트에서 동작하는 USB Host Driver를 대상으로 하며 대상 시스템의 정의 및 경계는 다음 그림과 같다.

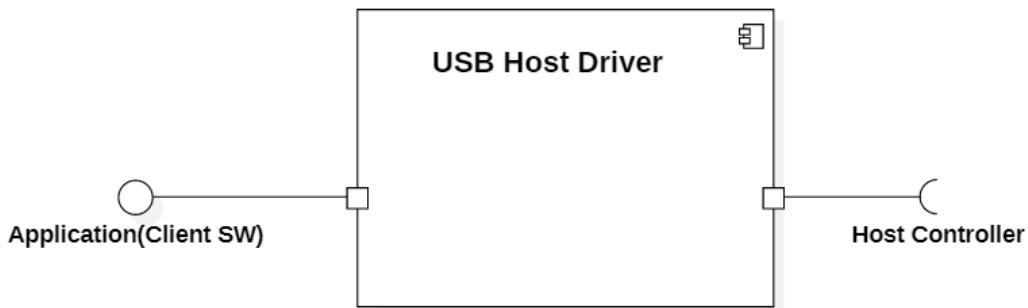


그림 3 시스템 정의

- **Application(Client SW):** Application은 연결된 USB 디바이스를 이용하는 SW로 연결된 USB 장치에 따라 다양한 서비스가 동작할 수 있다. 본 과제에서 대상으로 하는 시스템은 USB 메모리와 같이 USB Mass Storage 기능과 USB 키보드 마우스와 같은 Human Interface 장치를 구현하는 Application을 기본적으로 지원한다.
- **USB Host Driver:** 본 과제에서 대상으로 하는 시스템으로, USB 디바이스를 관리하고 Application으로부터 전달되는 요청을 외부의 USB 디바이스로 전달하는 기능을 한다. USB Host Driver는 아래와 같은 기능을 지원해야 한다.
  - **Device Connection Management:** USB 디바이스 연결(해제)에 따른 USB 자원 (Bandwidth, Power 등)을 할당(해제)하고, 연결된 디바이스와 관련된 Application에 디바이스 연결(해제)을 통보한다.
  - **Data Transfer Management:** Application에서 전달한 IRP(I/O Request Packet)를 Host Controller가 처리할 수 있는 Transaction 단위로 분할하고, Transaction을 우선 순위에 따라 Scheduling하여 Host Controller가 처리할 수 있는 Frame에 추가한다.
  - **Device Configuration Management:** USB 디바이스가 지원하는 다양한 Configuration 중 USB Host Driver

현재 Application이 사용하는 Configuration에 따라 Interface 혹은 endpoint에 대한 설정(ex, max packet size)을 수행한다.

- **Host Controller:** USB 전송 Protocol을 지원하는 HW로 USB Spec에 정의된 Host Controller 기능을 지원한다. 본 과제에서 대상으로 하는 USB2.0의 경우 EHCI를 기본적으로 지원하며, 상황에 따라 OHCI와 UHCI도 함께 지원한다. 다음은 Host Controller가 지원하는 세부 HW 기능들이다.
  - HW 상태 Reporting: Host Controller의 상태 정보를 Host Controller Driver에 통보한다.
  - Serializer/Deserializer: USB Protocol과 data 전송 message를 bit stream으로 변환하여 Physical한 Link를 통해 전송 한다.
  - Interrupt: USB 디바이스 상태 변화 및 전송 과정의 다양한 상황에 대한 interrupt 발생 시켜 Host Controller Driver에 통보한다.
  - Frame데이터 처리: Frame에 포함된 Transaction들을 전송 type에 따라 token을 생성하고 디바이스와 통신한다.

본 과제에서 주 대상으로 하고 있는 USB2.0에서 기본적으로 사용하는 EHCI의 경우 앞서 설명된 기능을 지원하기 위해 다음 그림과 같은 형태의 HW Interface를 제공한다. Host Controller는 Memory based I/O Register 형태로 Interface를 제공한다. 제공되는 주요 기능은 Host Controller의 상태 정보 확인, HW에 대한 Control, Interrupt 처리 등이다. 특히, Host Controller의 가장 중요한 기능인 데이터 전송을 위한 Interface는, 전송 데이터 저장을 위한 Shared Memory와 데이터가 저장된 위치를 나타내는 Register의 조합으로 제공된다. 데이터가 저장된 Shared Memory는 Isochronous와 Interrupt type과 같은 주기적 전송 방식과, Bulk와 Control type과 같은 비주기적 전송을 위한 transaction을 별도의 List로 관리된다.

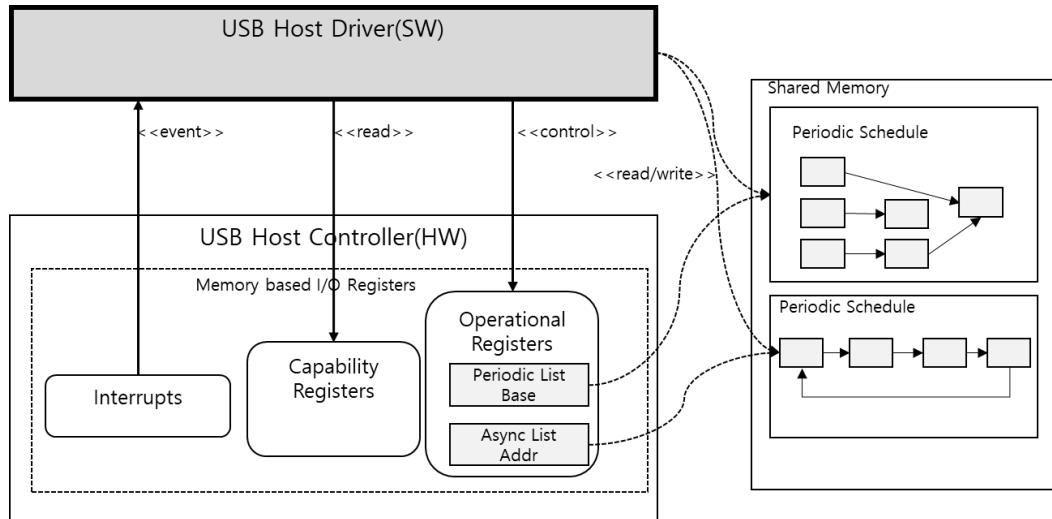


그림 4 EHCI HW interface 구조

### 1.2.2. 시스템 동작

USB Host Driver가 동작하는 기본적인 개념은 아래와 같다.

- 호스트에 USB 디바이스가 연결되는 경우 Host Controller는 새로운 장치가 연결되었음을 USB Host Driver에 알린다.
- USB Host Driver은 새로운 장치가 연결된 경우 장치의 종류와 Configuration 등을 확인하는 Enumeration 과정을 수행하여 호스트에서 해당 디바이스를 사용할 수 있도록 준비한다.
- USB Host Driver에서 Enumeration 과정이 완료된 경우 연결된 USB 디바이스의 종류에 따라 해당 디바이스를 사용할 수 있는 Application에게 디바이스가 연결되었음을 알린다.
- Application은 USB Host Driver를 통해 연결된 USB 디바이스로 요청을 전달(ex, Data read, write, state 확인 등) 한다.
- USB Host Driver는 Application이 전달한 요청을 USB Spec에서 정의한 메시지 형태로 변환하여 Host Controller로 전달한다.
- Host Controller는 Frame 형태로 변환된 데이터를 디바이스로 전달하고, 그 결과를 수신하여 USB Host Driver로 반환한다.

- USB Host Driver는 Host Controller가 전달한 디바이스의 응답을 Application에 전달한다.

### 1.3. 시스템 제약 사항

본 시스템이 동작하는 모바일 장치는 다음과 같은 제약 사항을 가지는 것을 가정한다.

- 본 시스템이 적용되는 모바일 장치는 USB2.0 Spec을 지원하는 것을 가정하고 있다.
- Host Controller는 OHCI, UHCI, EHCI가 지원되는 H/W를 가정한다.

### 1.4. 시스템의 목표

본 시스템이 적용되는 모바일 장치의 사용자는 임의의 시점에 USB 메모리나 키보드/마우스 등을 시스템에 연결하거나 분리할 수 있으며, 다수의 USB 디바이스 장치를 연결하여 사용할 수 있다. 따라서 시스템은 USB 디바이스의 다양한 오류 상황에서도 USB장치를 사용할 수 있도록 **가용성을 보장해야** 하며, 다수의 디바이스가 연결된 상태에서도 안정적인 데이터 전송 및 **전송 성능을 지원해야 한다**.

본 시스템을 이용하여 Application을 개발하는 개발자 측면에서는 USB 디바이스의 서비스를 구현하기 위해 충분한 Interface가 지원되어야 하며, 시스템이 다양한 Application의 요구 사항을 반영할 수 있도록 하는 **확장성** 측면을 고려해야 한다.

또한, 본 시스템은 현재는 USB 디바이스로 Mass Storage와 Human Interface 장치를 기본적으로 지원하는 것을 가정하지만, USB System은 새로운 Spec이 계속해서 추가되고 있고, 이를 이용하는 새로운 USB Device들도 계속해서 추가될 수 있기 때문에, 향후 새로운 Spec과 서비스 지원이 필요한 경우를 고려가 필요하다. 따라서, USB Spec의 추가와 새로운 디바이스 클래스가 추가되는 경우에 대한 시스템 확장성과 변경 용이성을 고려해야 한다.

## 2. 요구사항

### 2.1. 기능적 요구사항

본 시스템의 구조에 영향을 줄 수 있는 주요 Use case는 다음 그림과 같다. Application은 USB 저장장치 서비스와 같은 USB Host Driver를 이용하여 서비스를 구현하는 SW를 의미하며, Host Controller는 USB Protocol을 지원하는 HW를 의미한다. USB host Driver는 본 시스템을 의미한다.

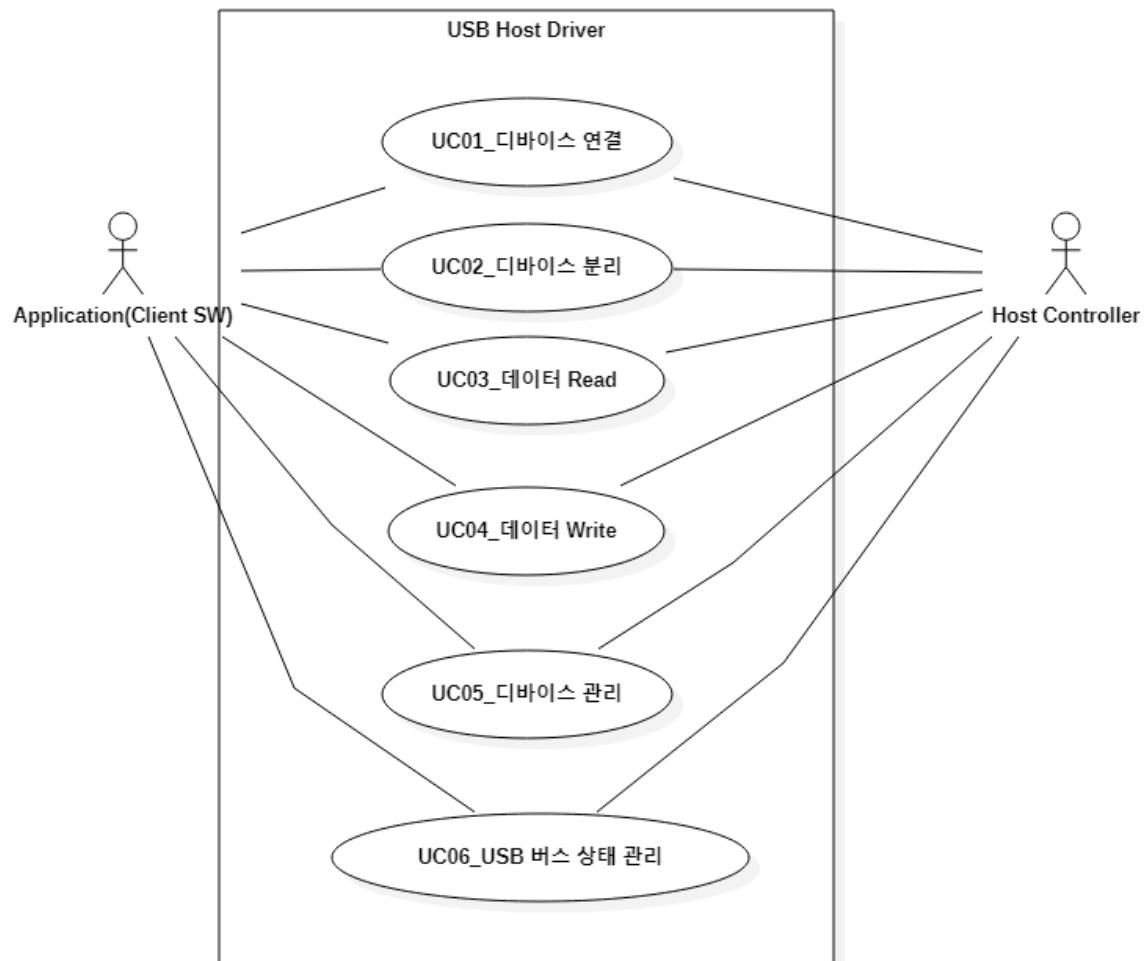


그림 5 USB Host Driver의 Use Case Diagram

UC_01	디바이스 연결
설명	시스템은 USB 디바이스가 호스트에 연결되는 경우 이를 인지하고 디바이스 사용에 필요한 초기 설정 과정(Enumeration)을 수행한다. 설정이 완료된 경우 디바이스를 사용할 Application에게 디바이스의 연결 상태를 통보한다.
행위자	Host Controller, Application
선행조건	시스템이 실행되어 USB 디바이스의 연결을 대기하고 있는 상태 Application에서 서비스할 USB 디바이스 종류를 등록해 둔 상태
후행조건	USB 디바이스 연결 상태
기본 동작	<p>1. (A) Host Controller 가 USB 디바이스의 연결 상태 변화를 시스템에 통보한다.</p> <p>2. (S) 시스템은 Host Controller 의 정보를 확인하여 새로운 디바이스가 연결된 것을 확인한다.</p> <p>3. (S) 시스템은 Host Controller 로 디바이스에 대한 Reset 명령을 전달한다.</p> <p>4. (S) 시스템은 디바이스의 Description 요청을 Host Controller 로 전달하고 Host Controller 는 그 결과를 통보한다(<b>AF1</b>).</p> <p>5. (S) 시스템은 디바이스 정보를 관리하는 저장소를 확인하여 가용한 Address 를 Address 할당 순서에 따라 선택한다.</p> <p>6. (S) 시스템은 선택된 Address 를 디바이스로 전달하기 위한 요청을 생성하여 Host Controller 로 전달하고, Host Controller 는 그 결과를 통보한다(<b>AF1</b>)</p> <p>7. (S) 시스템은 디바이스의 Configuration 정보 전달 요청을 Host Controller 로 전달하고 Host Controller 는 그 결과를 통보한다(<b>AF1</b>).</p> <p>8. (S) 시스템은 Configuration 에 필요한 리소스(bandwidth, Power 등)를 계산한다.</p> <p>9. (S) 시스템은 현재 리소스 상태 저장소를 확인하여 할당 가능한 리소스가 있는지 확인한다(<b>AF2</b>).</p> <ul style="list-style-type: none"> <li>- 리소스 상태 저장소는 전체 USB System 에서 가용한 자원의 양, 각 디바이스 및 endpoint 별로 할당된 자원 정보 등을 관리한다.</li> </ul> <p>10. (S) 시스템은 가용한 리소스 범위에 해당하는 디바이스의 Configuration 정보를 선택한다.</p> <p>11. (S) 시스템은 선택된 Configuration 설정 요청을 Host Controller 로 전달한다(<b>AF1</b>).</p> <p>12. (S) 시스템은 추가된 디바이스의 정보(Address, Description, Configuration 정보 등)를 시스템 내부적으로 관리하는 디바이스 정보 저장소에 등록한다.</p> <p>13. (S) 시스템은 추가된 디바이스를 USB System 의 BUS topology 정보 저장소에 추가한다.</p> <p>14. (S) 시스템은 Application 이 사용할 수 있는 Pipe 정보를 생성하여 Pipe 정보 저장소에 추가한다.</p>

	<ul style="list-style-type: none"> <li>- Pipe 정보 저장소에서는 &lt;Application, Pipe, 디바이스 Address, endpoint&gt;에 대한 매핑 정보를 포함하고 있어야 한다.</li> </ul> <p>15. (S) 시스템은 디바이스 종류와 Application 간의 매핑 정보를 검색(<b>AF3</b>)하여 디바이스를 사용할 Application 을 확인한다.</p> <p>16. (S) 해당 Application 이 실행 중인 경우 해당 Application 에게 디바이스가 연결되었음을 통보한다.</p> <ul style="list-style-type: none"> <li>- 시스템은 연결 완료 통보시에 Application 이 사용할 수 있는 각 endpoint 와 매핑 되는 pipe 정보도 함께 전달한다.</li> </ul>
추가 동작	<p><b>AF1:</b> 디바이스 연결 오류</p> <p>1. 4, 6, 7, 11 의 과정에서 Host Controller 로부터 응답이 없는 경우 시스템은 디바이스와의 연결이 끊긴 것으로 판단하고 에러 핸들링을 수행한다.</p> <p>2. 에러 처리 완료 후 디바이스 연결 대기 상태로 복귀 한다.</p> <p><b>AF2:</b> 시스템 리소스 부족</p> <p>1. 시스템은 디바이스 Configuration 에 필요한 리소스가 부족한 경우 해당 디바이스 연결 요청은 연결 대기 목록에 추가한다.</p> <p>2. 시스템은 주기적으로 대기 목록을 모니터링 하여 연결 대기 상태의 디바이스에 필요한 리소스가 있는지 확인한다.</p> <p>3. 디바이스에 필요한 리소스가 있는 경우 디바이스 접속 과정을 재 시작한다.</p> <p>4. 시스템은 일정 시간 이상 대기 목록에서 빠져나가지 않는 경우 연결 에러를 Application 에 통보한다.</p> <p><b>AF3:</b> Application 매핑 오류</p> <p>1. 시스템은 디바이스를 처리할 수 있는 Application 이 없는 경우 디바이스 연결을 대기 상태로 전환하여 대기 리스트에 추가한다.</p> <p>2. 시스템은 주기적으로 대기 리스트를 모니터링 하여 디바이스를 처리할 수 있는 Application 이 실행되는 경우 디바이스의 연결 상태를 해당 Application 에 통보한다.</p>

UC_02	디바이스 분리
설명	시스템은 USB 디바이스가 호스트에서 분리되는 경우 이를 인지하고 디바이스에 할당된 자원을 회수한다. 해당 디바이스를 사용중인 Application이 있는 경우 Application에게 디바이스의 분리 상태를 통보한다.
행위자	Host Controller, Application
선행조건	USB 디바이스 연결 상태 Application에서 서비스할 USB 디바이스 종류를 등록해 둔 상태

후행조건	USB 디바이스 분리 상태
기본 동작	<p>1. (A) Host Controller 가 USB 디바이스의 연결 상태 변화를 시스템에 통보한다.</p> <p>2. (S) 시스템은 Host Controller 의 정보를 확인하여 디바이스가 분리된 것을 확인한다.</p> <p>3. (S) 시스템은 디바이스 정보 저장소에서 Device ID(혹은 Address)를 이용하여 검색한다.</p> <p>4. (S) 시스템은 데이터 전송 상태 정보 저장소를 확인한다.</p> <p>5. (S) 시스템은 제거된 디바이스와 통신하던 데이터가 있는 경우 전송을 중단한다[<b>AF1</b>].</p> <p>6. (S) 시스템은 디바이스 정보에 포함된 디바이스를 사용중인 Application 정보를 확인한다(ex, Process ID 혹은 Application Name 등)[<b>AF2</b>].</p> <p>7. (S) 시스템은 해당 Application 에게 디바이스가 제거되었음을 통보한다.</p> <p>8. (S) 시스템은 제거된 디바이스에 할당된 자원(Address, Bandwidth 등)을 다른 디바이스가 활용할 수 있도록 리소스 정보 저장소에 가용 자원으로 업데이트 한다.</p> <p>8.1 (S) 시스템은 디바이스에 할당되었던 Address 는 미사용 상태로 변경한다.</p> <p>8.2 (S) 시스템은 디바이스의 Configuration 에 포함된 전체 endpoint 를 확인하여 각 endpoint 에 할당되었던 자원을 모두 가용 자원으로 변경한다.</p> <p>9. (S) 시스템은 Pipe 정보 테이블에서 해당 디바이스와 관련된 정보를 모두 삭제한다.</p> <p>10. (S) 시스템은 USB Bus topology 정보 저장소에서 해당 디바이스를 삭제한다.</p> <p>11. (S) 시스템은 디바이스 정보 저장소에 해당 디바이스의 상태를 분리 상태로 변경하여 저장한다.</p> <p>12. (S) 제거된 디바이스가 HUB 인 경우, 시스템은 HUB 에 포함된 전체 디바이스에 대해서 과정 3~11 의 과정을 반복한다.</p>
추가 동작	<p><b>AF1:</b> 전송 중인 데이터가 없는 경우 시스템은 추가 동작 없이 4 의 과정을 진행한다.</p> <p><b>AF2:</b> 디바이스와 연결된 Application 이 없는 경우 시스템은 추가 동작 없이 6 의 과정을 진행한다.</p>

UC_03	데이터 Read
설명	시스템은 Application이 요청한 주소의 데이터를 디바이스로부터 Read한다. 이때 전송하는 탑입은 endpoint의 설정 내용에 따라 USB2.0 Spec에서 정의된 4가지 탑입 중 하나를 선택하여 Read 한다.
행위자	Application, Host Controller

선행조건	시스템에서 디바이스의 초기화 과정이 완료되어 Application이 디바이스를 사용 가능한 상태
후행조건	-
기본 동작	<p>1. (A) Application 이 연결된 디바이스에 데이터를 Read 하기 위한 요청(IRP: I/O Request Packet)을 시스템에 전달한다.</p> <ul style="list-style-type: none"> <li>- Read 요청에는 Data 의 address, size, read 한 data 를 저장할 Buffer 정보가 포함된다.</li> </ul> <p>2. (S) 시스템은 Pipe 정보 저장소에서 Application 이 요청한 Pipe 와 연결된 디바이스와 endpoint 를 검색한다[AF1].</p> <p>3. (S) 시스템은 디바이스 정보 저장소에서 endpoint 에 대한 정보를 검색하여 Endpoint 의 정보(전송 type, max size 등)를 확인한다.</p> <p>4. (S) 시스템은 Transfer 목록에 Application 의 Read 요청을 추가한다.</p> <p>5. (S) 시스템은 Transfer 목록에 등록된 Transfer 요청을 우선 순위에 따라 정렬한다.</p> <ul style="list-style-type: none"> <li>- 동일 Type 내에서 별도의 우선 순위가 없다면, FIFO 형태로 처리한다.</li> </ul> <p>6. (S) 시스템은 IRP 에 포함된 Read 요청을 Transaction 단위로 변환한다.</p> <ul style="list-style-type: none"> <li>- Transaction 에는 &lt;디바이스 Address, Endpoint, Read Data address, read Length&gt; 정보가 포함되어야 한다.</li> </ul> <p>7. (S) 시스템은 생성된 Transaction 을 Transaction List 에 등록한다.</p> <p>8. (S) 시스템은 생성된 Transaction 들을 우선순위에 따라 순차적으로 Host Controller 와 공유하는 Transaction Queue 에 추가한다[AF2].</p> <p>9. (A) Host Controller 는 Transaction 처리 완료를 시스템에 통보한다[AF3].</p> <p>10. (S) 시스템은 Transfer 목록에서 전송 완료된 Transaction 에 해당하는 정보를 검색하여 Read 된 data 를 Application 에 전달할 buffer 에 복사한다.</p> <p>11. (S) 시스템은 Transfer 목록에 처리된 Transaction 의 size 를 업데이트한다.</p> <p>12. (S) 시스템은 해당 Transfer 요청이 완료된 상태인 경우 Application 에 데이터 Read 정보를 통보하고(read size), Transfer 목록에서 해당 Transfer 정보를 삭제한다.</p> <p>13. (S) 시스템은 Transfer 가 완료되지 않은 경우 과정 6 으로 복귀하여 이후 과정을 반복한다.</p>
추가 동작	<p><b>AF1:</b> 디바이스 검색 오류</p> <p>1. 디바이스가 검색되지 않은 상태인 경우 시스템은 오류로 판단하고 Application 에 오류를 통보한다.</p> <p><b>AF2:</b> Periodic 방식의 전송 요청인 경우</p> <p>1. 시스템은 디바이스의 endpoint 에 설정된 주기를 확인한다.</p> <p>2. 시스템은 해당 주기마다 디바이스에서 전달되는 정보를 Read 하기 위한 요청을</p>

	<p>생성하여 Periodic 전송 목록에 추가한다.</p> <p><b>AF3:</b> 데이터 전송 도중 디바이스 연결 오류</p> <ol style="list-style-type: none"> <li>1. 시스템은 전송 중 오류가 발생했음을 Application 에 통보한다.</li> <li>2. 시스템은 데이터 전송에 할당된 자원을 회수하고, 디바이스의 정보 저장소에 디바이스 상태 정보를 업데이트 한다.</li> </ol>
--	--

UC_04	데이터 Write
설명	시스템은 Application이 전달한 데이터를 Host Controller 통해 디바이스로 전송한다. 이때 전송하는 타입은 endpoint의 설정에 따라 USB2.0 Spec에서 정의된 4가지 타입 중 하나를 선택하여 전송한다.
행위자	Application, Host Controller
선행조건	디바이스의 초기화 과정이 완료되어 Application이 디바이스를 사용 가능한 상태
후행조건	-
기본 동작	<ol style="list-style-type: none"> <li>1. (A) Application 이 연결된 디바이스에 데이터를 Write 하기 위한 요청(IRP: IO Request Packet)을 시스템에 전달한다.</li> <li>2. (S) 시스템은 Pipe 정보 저장소를 확인하여 Application 이 요청한 Pipe 와 연결된 디바이스를 검색한다[<b>AF1</b>].</li> <li>3. (S) 시스템은 디바이스 정보 저장소에서 endpoint 에 대한 정보를 검색하여 Endpoint 의 정보(전송 type, max size 등)를 확인한다.</li> <li>4. (S) 시스템은 Transfer 목록에 Application 의 Write 요청을 추가한다.</li> <li>5. (S) 시스템은 Transaction List 에 등록된 Transaction 을 우선 순위에 따라 정렬한다. <ul style="list-style-type: none"> <li>- 동일 Type 내에서 별도의 우선 순위가 없다면, FIFO 형태로 처리한다.</li> </ul> </li> <li>6. (S) 시스템은 IRP 에 포함된 Write 요청을 Transaction 단위로 변환한다. <ul style="list-style-type: none"> <li>- Transaction 에는 &lt;디바이스 Address, Endpoint, Write Data address, Write Length&gt; 정보가 포함되어야 한다.</li> </ul> </li> <li>7. (S) 시스템은 Transaction 에 Application 이 전달한 data 를 Transaction 의 최대 허용 크기로 분할하여 Transaction 구조체에 복사한다. <ul style="list-style-type: none"> <li>- High Speed 의 경우 512byte 가 최대 크기이다.</li> </ul> </li> <li>8. (S) 시스템은 생성된 Transaction 들을 우선순위에 따라 순차적으로 Host Controller 와 공유하는 Transaction Queue 에 추가한다[<b>AF2</b>].</li> <li>9. (A) Host Controller 는 Transaction 처리 완료를 시스템에 통보한다[<b>AF3</b>].</li> <li>10. (S) 시스템은 Transfer 목록에 처리된 Transaction 의 size 를 업데이트한다.</li> <li>11. (S) 시스템은 해당 Transfer 요청이 완료된 상태인 경우 Application 에 처리</li> </ol>

	<p>결과를 통보하고, Transfer 목록에서 해당 Transfer 정보를 삭제한다.</p> <p>12. (S) 시스템은 Transfer 가 완료되지 않은 경우 과정 6 으로 복귀하여 이후 과정을 반복한다.</p>
추가 동작	<p><b>AF1:</b> 디바이스 연결 오류</p> <p>1. 디바이스가 연결되지 않은 상태인 경우 시스템은 Application 에 오류를 통보한다.</p> <p><b>AF2:</b> Periodic 방식의 전송 요청인 경우</p> <p>1. 시스템은 디바이스의 endpoint 에 설정된 주기를 확인한다.</p> <p>2. 시스템은 해당 주기마다 디바이스에서 전달되는 정보를 Write 하기 위한 요청을 생성하여 Periodic 전송 목록에 추가한다.</p> <p><b>AF3:</b> 데이터 전송 도중 디바이스 연결 오류</p> <p>1. 시스템은 전송 중 오류가 발생했음을 Application 에 통보한다.</p> <p>2. 시스템은 데이터 전송에 할당된 자원을 회수하고, 디바이스의 정보 저장소에 디바이스 상태 정보를 업데이트 한다.</p>

UC_05	디바이스 설정 관리
설명	시스템은 Application이 요청한 디바이스 Configuration 변경을 수행하고 그 결과를 Application에 통보한다.
행위자	Application, Host Controller
선행조건	디바이스의 초기화 과정이 완료되어 Application이 디바이스를 사용 가능한 상태
후행조건	-
기본 동작	<p>1. (A) Application 은 연결된 디바이스의 Configuration 을 변경하기 위한 요청을 시스템에 전달한다.</p> <p>2. (S) 시스템은 디바이스 정보 저장소를 확인하여 디바이스가 지원 가능한 설정 정보를 확인한다[<b>AF1</b>].</p> <p>3. (S) 시스템은 Application 이 요청한 Configuration 정보가 디바이스가 지원하는 Configuration 과 매칭되는지 판단한다.</p> <p>4. (S) 시스템은 리소스 정보 저장소에서 현재 리소스(ex, bandwidth) 상태 확인한다.</p> <p>5. (S) 시스템은 Application 이 요청한 Configuration 에 필요한 리소스와 현재 사용 가능한 리소스를 비교하여 Configuration 을 지원할 수 있는지 판단한다[<b>AF2</b>].</p> <p>6. (S) 시스템은 Configuration 에 포함된 각 Interface 와 Interface 에 포함된 각각의 endpoint 에 대한 설정 요청을 Host Controller 로 전달한다[<b>AF3</b>].</p> <p>7. (S) 시스템은 디바이스 정보 저장소에 각 endpoint 에 대한 변경된 설정 정보를</p>

	<p>업데이트 한다.</p> <p>8. (S) 시스템은 리소스 정보 저장소에 변경된 자원 할당 정보를 업데이트 한다.</p> <p>9. (S) 시스템은 Configuration 변경 결과를 Application 에 통보한다.</p>
추가 동작	<p><b>AF1:</b> 디바이스가 지원하지 않은 Configuration인 경우 1. 시스템은 Configuration 미지원을 Application 에 통보한다.</p> <p><b>AF2:</b> 자원 할당 오류 1. Application 에서 요청한 설정 변경이 지원할 수 없는 상태인 경우 시스템은 Application 에게 오류를 통보한다.</p> <p><b>AF3:</b> Configuration 변경 오류 1. (S) 시스템은 각 endpoint 에 할당된 자원의 상태를 이전 상태로 변경한다. 2. (S) 시스템은 각 endpoint 에 이전 설정으로 설정 변경 요청을 전달한다.</p>

UC_06	USB 버스 상태 관리
설명	시스템은 Host Controller가 전달하는 USB BUS의 상태 변화를 확인하고, 디바이스 상태 변화에 따라 자원을 회수하거나 재 할당 한다.
행위자	Host Controller
선행조건	디바이스의 초기화 과정이 완료되어 Application이 디바이스를 사용 가능한 상태
후행조건	-
기본 동작	<p>1. (A) Host Controller 가 시스템에 BUS 상태 변화를 통보한다.</p> <p>2. (S) 시스템은 Host Controller 에 저장된 정보를 Read 하여 USB BUS 의 상태 변경 내용을 확인한다.</p> <p>3. (S) 상태 변경된 디바이스가 HUB 인 경우, 시스템은 USB System 의 Bus Topology 정보 저장소를 검색한다.</p> <p>4. (S) 시스템은 HUB 에 포함된 전체 디바이스 목록을 생성한다.</p> <p>5. (S) 디바이스가 Suspend 인 경우[<b>AF1</b>], 시스템은 자원 정보 저장소에서 디바이스가 현재 사용중인 자원(ex, Bandwidth)을 확인한다.</p> <p>6. (S) 시스템은 디바이스가 사용중인 자원을 회수하고 가용한 자원 정보를 업데이트 한다.</p> <p>7. (S) 시스템은 디바이스 상태를 메시지 전송 불가 상태로 변경한다.</p> <ul style="list-style-type: none"> <li>- Root HUB 가 Suspend 된 경우 USB 전체 System 에 포함된 디바이스가 Suspend 상태로 전환된다.</li> </ul> <p>8. (S) 시스템은 HUB 에 포함된 전체 디바이스 목록에 대해서 과정 5 ~ 7 의 과정을 반복한다.</p>

추가 동작	<p><b>AF1:</b> Resume인 경우</p> <ol style="list-style-type: none"> <li>1. (S) 시스템은 USB System 의 Bus Topology 정보 저장소를 확인하고 Resume 된 HUB 에 포함된 디바이스들을 확인한다.</li> <li>2. (S) 시스템은 Resume 된 HUB 에 포함된 디바이스가 요청하는 자원(ex, Bandwidth)를 할당한다.</li> <li>3. (S) 시스템은 HUB 에 포함된 디바이스 상태를 사용 가능 상태로 변경한다.</li> </ol>
-------	--

## 2.2. 비기능적 요구사항

NFR_01	신뢰성	디바이스 연결 성공 비율
설명		호스트에 USB 디바이스가 연결되는 경우 디바이스가 정상적으로 인식되어 서비스 될 확률은 높을수록 좋다.
환경		시스템이 정상 동작 상태
자극		시스템에 새로운 디바이스가 연결됨
반응		<p>시스템은 아래와 같은 반응이 발생한 경우 정상 연결로 판단한다.</p> <p><b>&lt;정상 동작 반응&gt;</b></p> <ol style="list-style-type: none"> <li>1. 시스템은 디바이스가 연결된 것을 인식한다.</li> <li>2. 시스템이 디바이스 초기화 과정(Enumeration)을 수행하고 디바이스를 사용 가능 상태로 만든다.</li> </ol> <p>시스템은 디바이스 정보를 디바이스 정보 저장소에 등록하고, Application 에 디바이스가 연결된 것을 통보한다.</p> <p>시스템에서 아래와 같은 반응이 발생하는 경우 연결 오류로 판단한다.</p> <p><b>&lt;오류 반응 1&gt;</b></p> <ol style="list-style-type: none"> <li>1. 시스템은 디바이스가 연결된 것을 인식한다.</li> <li>2. 시스템이 디바이스 초기화 과정(Enumeration)을 수행 중 디바이스 Configuration 에 필요한 자원(Bandwidth, Address 등) 할당에 실패한다. <ul style="list-style-type: none"> <li>- 시스템에 가용한 자원이 있지만 자원이 할당되지 않은 경우</li> <li>- 시스템에서 해당 디바이스의 이전 상태 정보가 남아있어 새로운 정보와 충돌 되는 경우</li> </ul> </li> <li>3. 시스템은 리소스 할당 오류로 디바이스 연결 실패를 Application 에 통보함.</li> </ol> <p><b>&lt;오류 반응 2&gt;</b></p> <ol style="list-style-type: none"> <li>1. 시스템은 디바이스가 연결된 것을 인식한다.</li> </ol>

	<p>2. 시스템은 디바이스 초기화 과정이 특정 시간 이상 지연 되는 경우 연결 실패로 판단 한다.</p> <ul style="list-style-type: none"> <li>- Enumeration 과정에서 Control 메시지나 Configuration message 의 전송이 내부 스케줄링 오류로 지연되는 경우</li> <li>- 외부 디바이스의 동작 오류로 연결이 지연되는 경우</li> <li>- 내부 컴포넌트의 비정상 동작으로 연결이 처리되지 못하는 경우</li> </ul> <p>3. 시스템은 디바이스 연결 지연으로 Application 에 연결 실패를 통보한다.</p>
측정	[연결 성공률] = [연결 성공 횟수] / [연결 시도 횟수]
제약	[연결 성공률] >= 0.9999

NFR_02	성능	디바이스로부터 입력에 대한 처리 시간
설명		호스트에 키보드 마우스와 같은 Human Interface Device(HID)가 연결되는 경우 HID 장치로부터의 입력에 대한 반응 시간은 사용자 경험과 직결되며, 시간 지연될 경우 오류로 인식될 수 있기 때문에, 입력에 대한 반응시간은 짧을수록 좋다.
환경		디바이스가 정상 동작 상태 디바이스에 다수의 HID 장치가 연결되어 동작하는 상태
자극		Host Controller 가 READ Transaction 완료를 시스템에 통보함
반응		<p>1. 시스템은 Host Controller로부터 전달된 Event 를 처리할 Handler 를 생성한다.</p> <p>2. 시스템은 완료된 READ Transaction 정보에 포함된 Endpoint 를 확인한다.</p> <p>3. 시스템은 Pipe 정보 저장소에서 endpoint 와 매핑되는 Application 정보를 확인 한다.</p> <p>4. 시스템은 Transfer 정보 저장소에서 Transaction 에 해당하는 Transfer 정보를 검색한다.</p> <p>5. 시스템은 Transfer 정보에 포함된 Application 과 공유하는 Buffer 로 디바이스 입력 정보를 복사한다.</p> <p>6. 시스템은 디바이스의 입력(데이터)을 Application 에 통보한다.</p>
측정		[입력 처리시간] = [시스템이 디바이스 입력을 Application에 통보한 시간] – [Host Controller가 READ Transaction 완료를 시스템에 통보한 시간]
제약		[입력 처리 시간] <= 5ms

### 2.3. 품질 속성

QA_01	성능	데이터 전송 속도
설명		USB의 Data 전송 모드 중 Isochronous를 제외한 모드는 전송 속도에 대한 Spec에서의 제약은 없지만, 전송 속도는 사용자 경험 측면에서 매우 중요하기 때문에 데이터 전송(Read/Write) 속도는 빠를수록 좋다.
환경		시스템이 정상 동작 중 시스템에 사용 가능한 디바이스가 연결된 상태
자극		Application 이 시스템에 데이터 전송(Read/Write)을 요청
반응		<ol style="list-style-type: none"> <li>1. 시스템은 Application 의 전송 요청을 내부적으로 관리하는 Transfer 목록에 추가한다.</li> <li>2. 시스템은 pipe 매핑 테이블을 검색하여 Application 이 전달한 Pipe 에 해당하는 endpoint 를 확인한다.</li> <li>3. 시스템은 디바이스 정보 저장소를 확인하여 endpoint 에 해당하는 설정 정보를 확인한다.</li> <li>4. 시스템은 전송 요청을 Transaction 단위로 Fragment 하여 Transaction 을 생성한다.</li> <li>5. 시스템은 생성된 Transaction 을 우선 순위에 따라 Host Controller 와 공유하는 Transaction Queue 에 추가한다.</li> <li>6. 시스템은 Transaction 완료를 Host Controller 로부터 전달받는다.</li> <li>7. 시스템은 완료된 Transaction 을 Transfer 정보에 업데이트 한다.</li> <li>8. 시스템은 Transfer 의 모든 Transaction 이 완료된 경우, 해당 Transfer 를 요청한 Application 에게 전송 완료를 통보한다.</li> </ol>
측정		[전송 속도] = [Application이 요청한 전송 데이터 크기] / ([시스템이 Application에게 전송 완료를 통보한 시간] – [Application이 시스템에 데이터 전송을 요청한 시간])

QA_02	성능	디바이스의 연결/해제시 인식 지연 시간
설명		다수의 디바이스가 연결된 상태에서 호스트에 새로운 디바이스가 연결되거나 제거되는 경우 디바이스 연결 상태 변경을 인식하는 속도는 빠를수록 좋다.
환경		시스템이 정상 동작 상태
자극		시스템에 새로운 디바이스 연결/제거 정보가 전달됨

반응	<ol style="list-style-type: none"> <li>1. 시스템은 디바이스가 연결/제거된 것을 인식한다.</li> <li>2. 시스템은 디바이스로부터 Configuration 정보를 확인하여 디바이스 정보 저장소에 추가한다.</li> <li>3. 시스템은 디바이스에 Address 를 할당하고 할당된 Address 를 디바이스로 전달한다.</li> <li>4. 시스템은 USB System 의 가능한 자원을 확인하여 디바이스 Configuration 을 지원할 수 있는지 판단한다.</li> <li>5. 시스템은 디바이스의 지원 가능한 Configuration 을 선택하여 디바이스에 전달한다.</li> <li>6. 시스템은 내부 자원 정보 저장소에 가능한 자원 정보를 업데이트 한다.</li> <li>7. 시스템은 추가된 디바이스 정보를 디바이스 정보 저장소, BUS Topology 정보 저장소 등에 업데이트 한다.</li> <li>8. 시스템은 디바이스 정보 저장소에서 디바이스를 처리할 Application 을 확인한다.</li> <li>9. 시스템은 해당 Application 에 디바이스가 연결/제거된 것을 통보한다.</li> </ol>
측정	[디바이스 인식 지연 시간] = [시스템이 Application으로 장치 연결을 통보한 시간] – [시스템에 새로운 디바이스 연결 정보가 전달된 시간]

QA_03	성능	동시에 연결할 수 있는 디바이스의 수
설명		하나의 호스트에는 다수의 디바이스가 연결될 수 있으면 HUB를 통해 연결을 계속 해서 확장할 수 있기 때문에, USB Spec이 정한 범위 내에서(최대 127개) 시스템은 다수의 디바이스 연결을 동시에 처리할 수록 좋다.
환경		디바이스가 정상 동작 상태 시스템에 다수의 디바이스가 연결된 상태
자극		시스템에 새로운 디바이스 연결 정보가 전달됨
반응		<ol style="list-style-type: none"> <li>1. 시스템은 디바이스가 연결/제거된 것을 인식한다.</li> <li>2. 시스템은 디바이스로부터 Configuration 정보를 확인하여 디바이스 정보 저장소에 추가한다.</li> <li>3. 시스템은 디바이스에 Address 를 할당하고 할당된 Address 를 디바이스로 전달한다.</li> <li>4. 시스템은 USB System 의 가능한 자원을 확인하여 디바이스 Configuration 을 지원할 수 있는지 판단한다.</li> <li>5. 시스템은 디바이스의 지원 가능한 Configuration 을 선택하여 디바이스에 전달한다.</li> </ol>

	<p>6. 시스템은 내부 자원 정보 저장소에 가용한 자원 정보를 업데이트 한다.</p> <p>7. 시스템은 추가된 디바이스 정보를 디바이스 정보 저장소, BUS Topology 정보 저장소 등에 업데이트 한다.</p> <p>8. 시스템은 디바이스 정보 저장소에서 디바이스를 처리할 Application 을 확인한다.</p> <p>9. 시스템은 해당 Application 에 디바이스가 연결/제거된 것을 통보한다.</p>
측정	동시에 연결 가능한 디바이스의 수

QA_04	변경 용이성	새로운 디바이스 클래스 추가 용이성
설명	새로운 디바이스 클래스가 추가되는 경우 신규 클래스 지원을 위한 개발 비용이 작을 수록 좋다.	
환경	시스템이 개발되어 호스트 시스템에 적용된 상태	
자극	현재 시스템이 지원하는 Mass Storage, HID 클래스와는 다른 디바이스 클래스(ex, Audio Device Class, Communication Device Class 등)에 대한 추가 지원 요청	
반응	<ol style="list-style-type: none"> <li>개발자는 신규 디바이스 클래스 지원 기능을 구현하여 시스템에 적용한다.</li> <li>개발자는 구현된 시스템을 호스트에 적용할 수 있도록 패키징 하여 재 배포한다.</li> </ol>	
측정	추가 용이성 = [변경에 필요한 개발 비용(M/M)]	

QA_05	변경 용이성	새로운 실행 환경(HW, OS) 지원 용이성
설명	시스템이 동작 하는 실행환경이 변경되는 경우 신규 실행 환경 지원을 위한 개발 비용은 작을수록 좋다.	
환경	시스템이 개발되어 호스트 시스템에 적용된 상태	
자극	<p>현재 시스템이 동작하는 실행 환경과 다른 새로운 HW 및 OS 지원 요청</p> <ul style="list-style-type: none"> <li>- <b>HW 변경 요인 1:</b> 신규 Host Controller Interface(ex, xHCI) 지원 요청</li> <li>- <b>HW 변경 요인 2:</b> 기존 HW 의 Interface(Register Address 등) 변경</li> <li>- <b>OS 변경 요인 1:</b> OS 버전 업데이트에 따른 OS Interface 변경</li> <li>- <b>OS 변경 요인 2:</b> 신규 OS 에 시스템을 적용하는 경우</li> </ul>	
반응	<ol style="list-style-type: none"> <li>개발자는 신규 HW 지원에 필요한 모듈을 구현하여 시스템에 포함한다.</li> <li>개발자는 신규 OS 에서 지원하는 Interface 를 이용하여 시스템을 구현한다.</li> <li>개발자는 신규 실행 환경을 지원하는 시스템을 배포한다.</li> </ol>	
측정	지원 용이성 = [변경에 필요한 개발 비용(M/M)]	



### 3. 시스템 구조

#### 3.1. 시스템 전체 구조

본 과제에서는 시스템의 가장 우선 순위가 높은 품질 속성인 데이터 전송 성능을 향상시키기 위하여 각 Task의 병렬 처리가 용이하도록 “Message Buffer 기반의 Dispatcher” 구조로 시스템을 설계하였다. 시스템의 주요 Task를 처리할 별도의 Thread를 구성하고, 각 Thread간의 병렬 처리가 용이하도록 Message Buffer를 기반으로 통신하도록 하였다. 특히, 데이터 전송과 관련된 작업은 병렬 처리 효율을 향상시키기 위하여 Dispatcher를 이용하여 미리 생성된 Thread에 Task를 할당하는 구조로 설계하였다. 다음 그림은 전체 시스템 구성에 대한 설명이다.

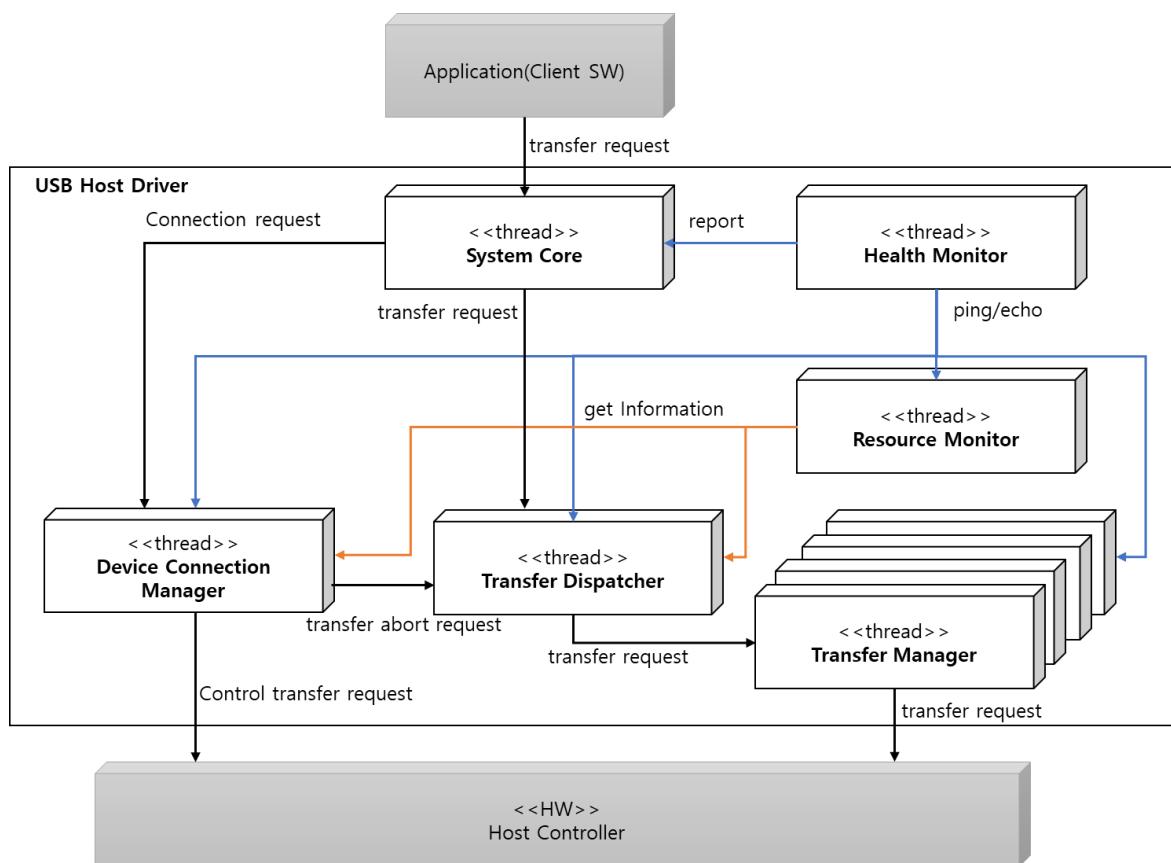


그림 6 시스템 전체 구조

전체 시스템은 USB 기능과 관련된 작업을 처리하는 Device Connection Manager, Transfer Dispatcher, Transfer Manager와 시스템의 안정성을 향상시키기 위한 Health Monitor, Resource Monitor 그리고 전체 시스템을 관리하기 위한 System Core로 구성된다. 시스템을 구성하는 각

Thread의 세부 역할은 다음과 같다.

- **System Core:** Application과 Interface 기능과 시스템의 Main controller 역할을 한다. 디바이스 연결, 데이터 전송 완료 등과 같은 시스템에 전달되는 다양한 이벤트들을 처리하며, Health Monitor를 통해 Thread의 비정상 동작이 Report되는 경우 해당 Thread의 복구를 진행하는 역할을 한다.
- **Device Connection Manager:** System Core가 전달한 디바이스 연결 요청을 처리한다. 디바이스에 대한 enumeration 과정 및 disconnection 과정을 처리하고, Thread 내부적으로 각 디바이스에 대한 정보, USB System의 Resource 정보 등 디바이스 연결과 관련된 정보들을 관리한다.
- **Transfer Dispatcher:** System Core의 Transfer 요청을 처리한다. Transfer 요청을 내부적으로 관리하는 Queue에 저장하고 우선 순위에 따라 scheduling하는 역할과, 각 Transfer Manager Thread의 Load를 분석하여 요청을 분산하여 Transfer Manager에 전달하는 Load balancer 역할을 수행한다.
- **Transfer Manager:** Transfer Dispatcher에서 전달된 Transfer 요청을 처리한다. 전송 데이터를 transaction 단위로 분할하고, Host Controller가 지원하는 HW Interface에 따라 생성한 transaction을 HW에 전달하는 역할을 수행한다.
- **Health Monitor:** 시스템 내부 Thread들의 정상 동작 여부를 모니터링 하며, 비정상 동작 Thread가 확인된 경우 결과를 Core System Manager에 전달하는 역할을 한다.
- **Resource Monitor:** USB System은 물리적으로 제한된 자원을 가지고 있기 때문에 제한된 자원이 낭비되지 않도록 모니터링 하는 역할을 한다. Device Connection Manager를 통해 디바이스별로 할당된 자원 정보를 수집하고, Transfer Dispatcher에서 실시간으로 각 디바이스의 자원 사용 정보를 수집하여 할당된 자원에 비해 사용율이 낮거나 사용되지 않는 디바이스의 자원을 회수하는 역할을 한다.

다음은 시스템의 주요 동작 시나리오 수행 과정에서 각 Thread 가 어떻게 연관되어 동작하는지에 대한 설명이다.

### 3.1.1. 디바이스 연결 동작

다음 그림은 디바이스 연결 과정에 대한 Sequence diagram으로 System Core, Device Connection Manager Thread간 동작 관계를 나타낸다.

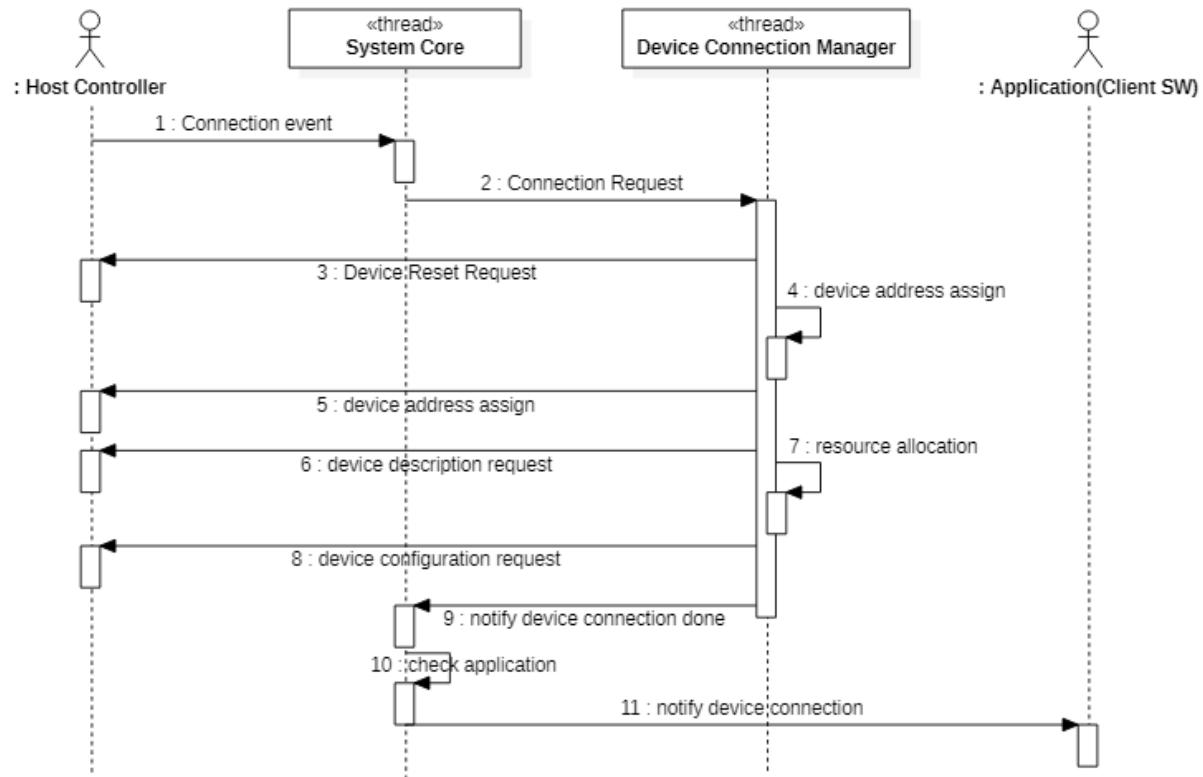


그림 7 디바이스 연결을 위한 Thread 동작 Sequence

### 3.1.2. 데이터 전송 동작

다음 그림은 데이터 전송 과정에 대한 Sequence diagram으로 System Core, Transfer Dispatcher, Transfer Manager Thread간 동작 관계를 나타낸다.

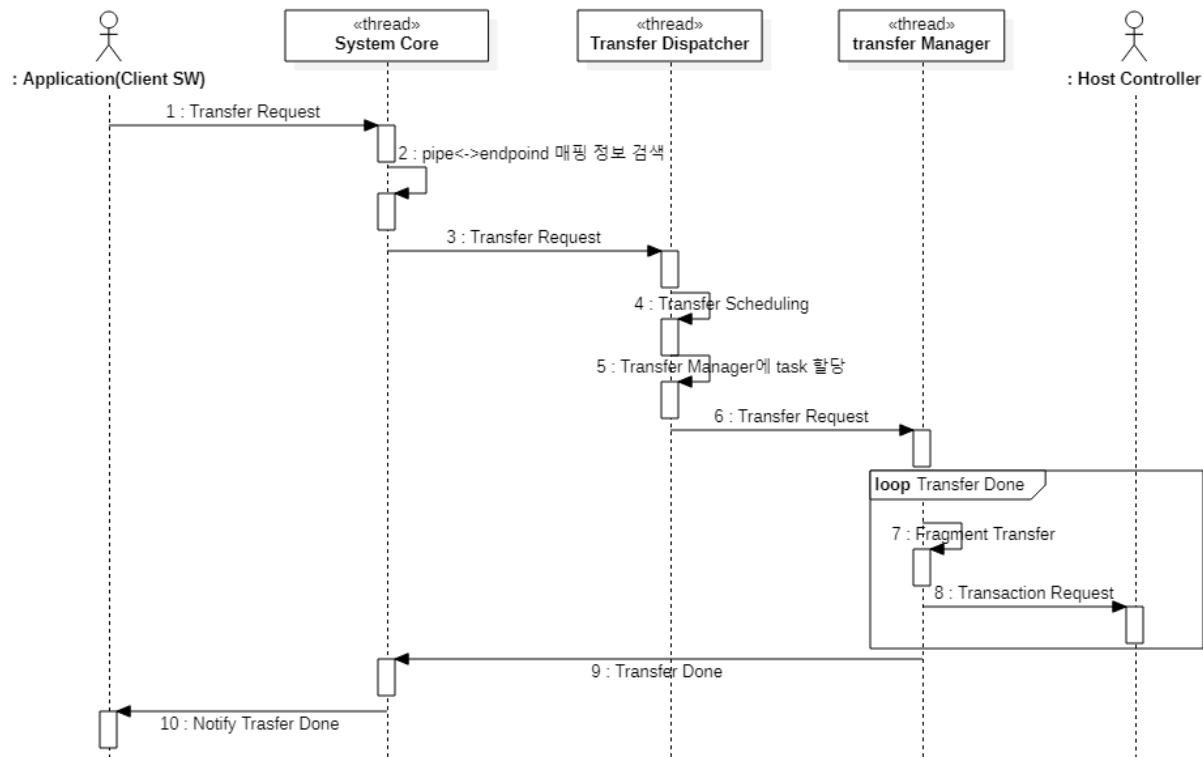


그림 8 데이터 전송을 위한 Thread 동작 sequence

### 3.1.3. Health Monitoring 동작

다음 그림은 Health Monitoring 동작 과정에 대한 Sequence diagram으로 Health Monitor와, Health Monitoring 대상 Thread, 그리고 Health Monitoring 결과를 처리하는 System Core Thread 사이의 동작 관계를 나타낸다. Health Monitoring의 대상이 되는 Device Connection Manager, Transfer Manager, Resource Monitor 모두 동일한 방식으로 동작한다.

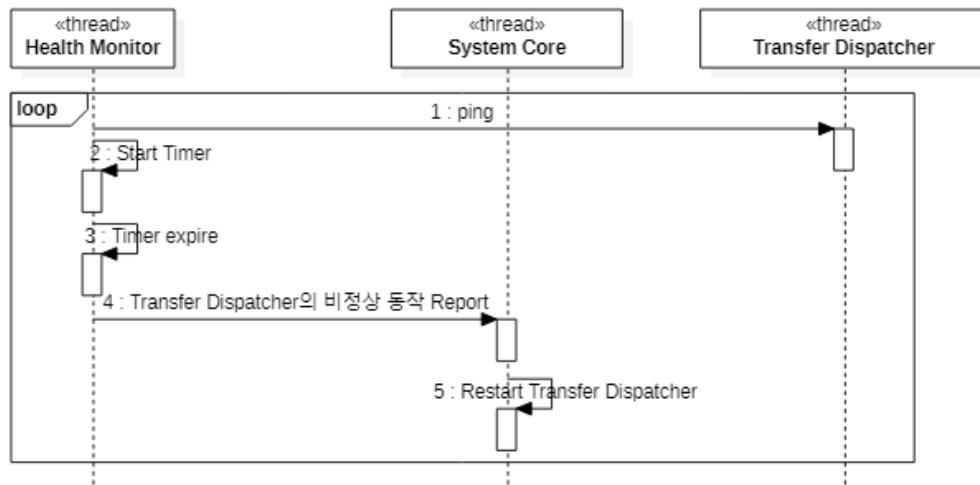


그림 9 Health Monitoring을 위한 thread 동작 sequence

### 3.1.4. Resource Monitoring 동작

다음 그림은 Resource Monitoring 동작 과정에 대한 Sequence diagram으로 Resource Monitor, Device Connection Manager, Transfer Dispatcher, System Core Thread 사이의 동작 관계를 나타낸다.

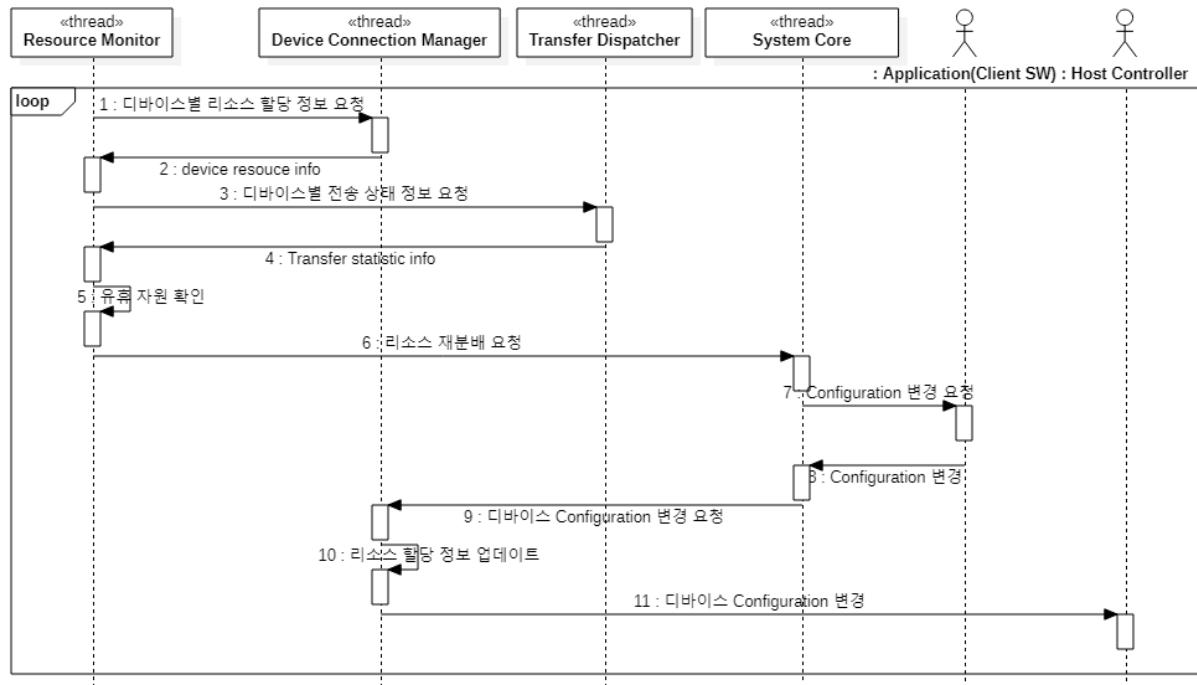


그림 10 Resource Monitoring을 위한 thread 동작 sequence

### 3.2. 시스템 세부 구조

시스템을 구성하는 Thread별 주요 컴포넌트를 명세한 Deployment view는 다음 그림과 같다.

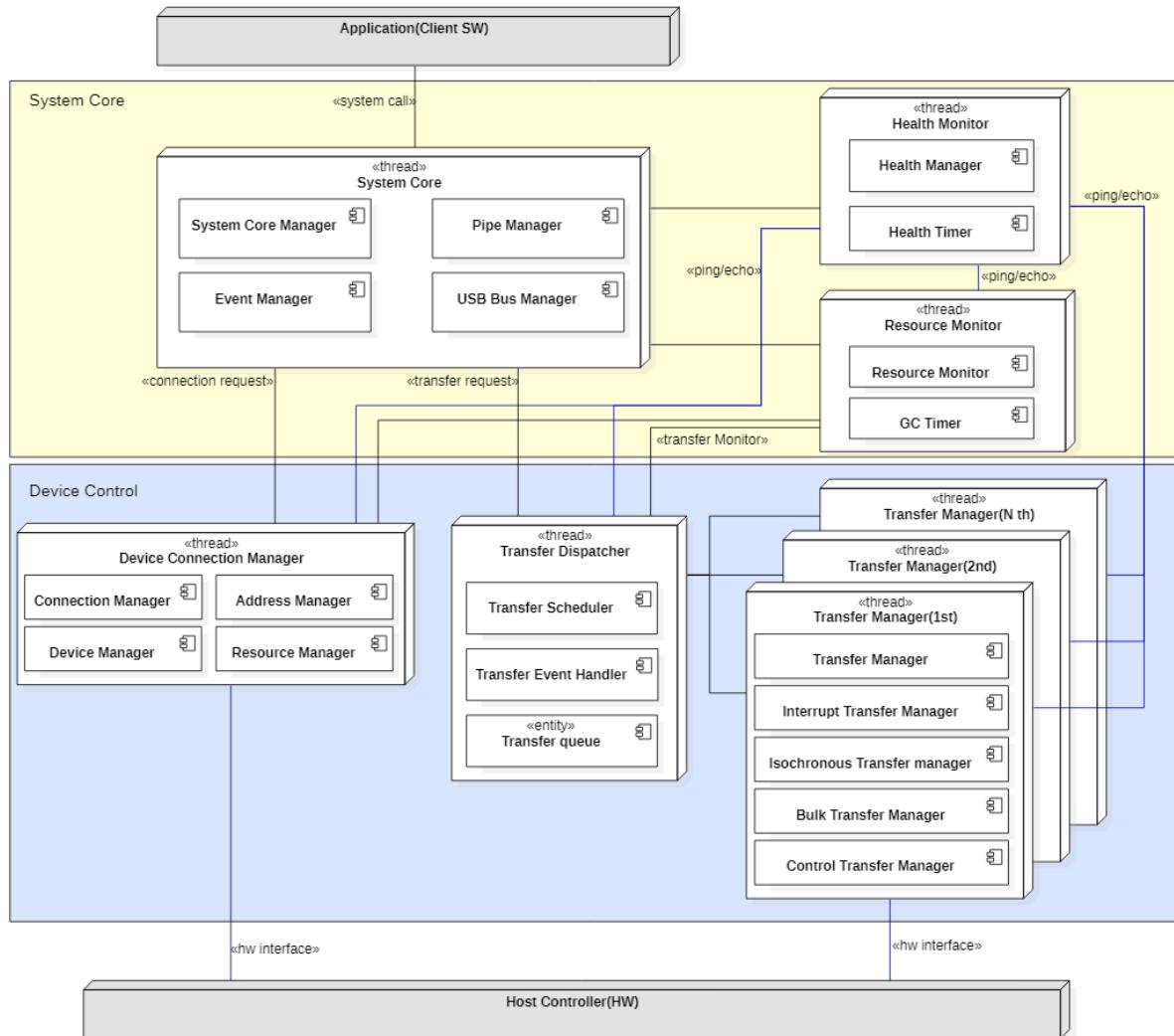


그림 11 시스템 Deployment view

**System Core Thread**는 각 Application에서 전달되는 데이터 전송과 Host Controller로부터 발생하는 디바이스 연결 요청들을 처리하는 기능을 수행한다. 이를 위해 Thread 내부적으로 시스템 전체를 관리하는 Core System Manager, Pipe<->endpoint 간의 매핑 정보를 관리하는 Pipe Manager, 시스템에 전달되는 Event들을 처리하는 Event Manager, USB Bus topology 정보를 처리하는 USB Bus Manager 컴포넌트를 포함하고 있다.

**Device Connection Manager Thread**는 System Core에서 전달된 디바이스 연결/해제와 관련된 요

성을 처리한다. 이를 위해 Thread 내부적으로 디바이스 연결을 관리하는 Connection Manager, 디바이스 정보를 관리하는 Device Manager, 디바이스에 할당되는 Address 정보를 관리하는 Address Manager, USB System의 전체 resource와 각 디바이스별 자원할당 정보를 관리하는 Resource Manager 컴포넌트를 포함하고 있다.

**Transfer Dispatcher Thread**는 데이터 전송 요청에 대한 Scheduling과 Transfer Manager Thread 사이의 Load balancer 역할을 한다. 이를 위해 내부적으로 Transfer 요청을 저장하기 위한 Transfer Queue와 Transfer Queue에 저장된 Request의 우선 순위를 정하고, Transfer Manager Thread의 load를 분산하기 위한 Transfer Scheduler 컴포넌트를 포함하고 있다.

**Transfer Manager Thread**는 Host Controller(HW)를 통해 데이터를 전송하는 역할을 한다. 이를 위해 Thread로 전달된 전체 요청을 관리하는 Transfer Manager와 각 전송 타입별로 실제 전송 과정을 진행하는 Interrupt Transfer Manager, Isochronous Transfer Manager, Bulk Transfer Manager, Control Transfer Manager 컴포넌트를 포함하고 있다.

**Health Monitor Thread**는 시스템 내부 Thread들의 정상 동작 여부를 모니터링 하고 비정상 동작을 탐지한 경우 이를 System Core에 알리는 역할을 한다. 이를 위해 각 Thread의 상태를 모니터링 하는 Health Manager, 모니터링 주기 및 ping message에 대한 timeout을 관리하기 위한 Health Timer 컴포넌트를 포함하고 있다.

**Resource Monitor Thread**는 각 디바이스의 자원 사용 상황을 모니터링 하여 낭비되는 자원이 있는지 모니터링 하고, 자원이 낭비되는 상황으로 판단하면 자원 재분배를 System Core에 요청하는 역할을 한다. 이를 위해 자원 낭비를 모니터링 하고 재분배를 요청하는 Garbage Correction Manager, 모니터링 주기를 판단하기 위한 Garbage Correction Timer 컴포넌트를 포함하고 있다.

다음은 각 Thread의 세부 컴포넌트 구조와 동작에 대한 설명이다.

### 3.2.1. 시스템 내부 Thread간 통신 구조

시스템 내부 Thread간의 기본적인 통신 방식은 Message Buffer를 기반으로 한다. 시스템 내부의 각 Thread들은 자체적인 Message Buffer를 가지고 있으며, Message Buffer에 등록된 Request를 순차적으로 처리한다. 따라서 시스템 내부의 각 Thread는 Message Buffer에 저장된 request를 기반으로 독립적으로 동작 하기 때문에 다른 Thread의 동작에 대한 의존성을 낮추고 병렬적인 처리가 가능하다. 다음 그림은 시스템 내부 컴포넌트 사이의 Message Buffer 연결 관계를 나타내는 것으로 System USB Host Driver

Core Thread를 예로 설명한 것이다.

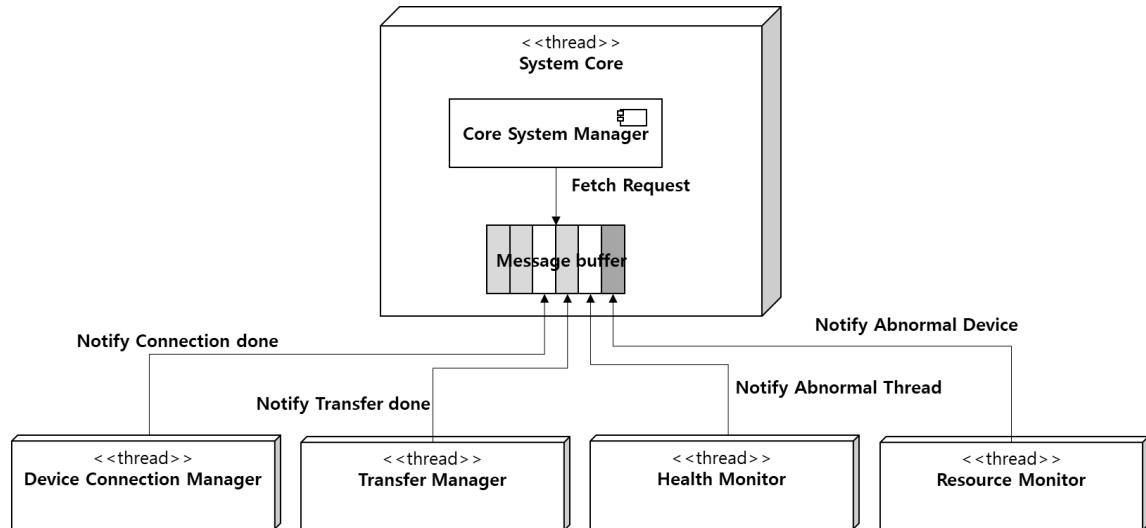


그림 12 Message Buffer를 이용한 Thread간 통신 구조 방식

### 3.2.2. System Core

다음 그림은 System core Thread를 구성하는 세부 컴포넌트와 컴포넌트 사이의 연결 관계를 나타내고 있다.

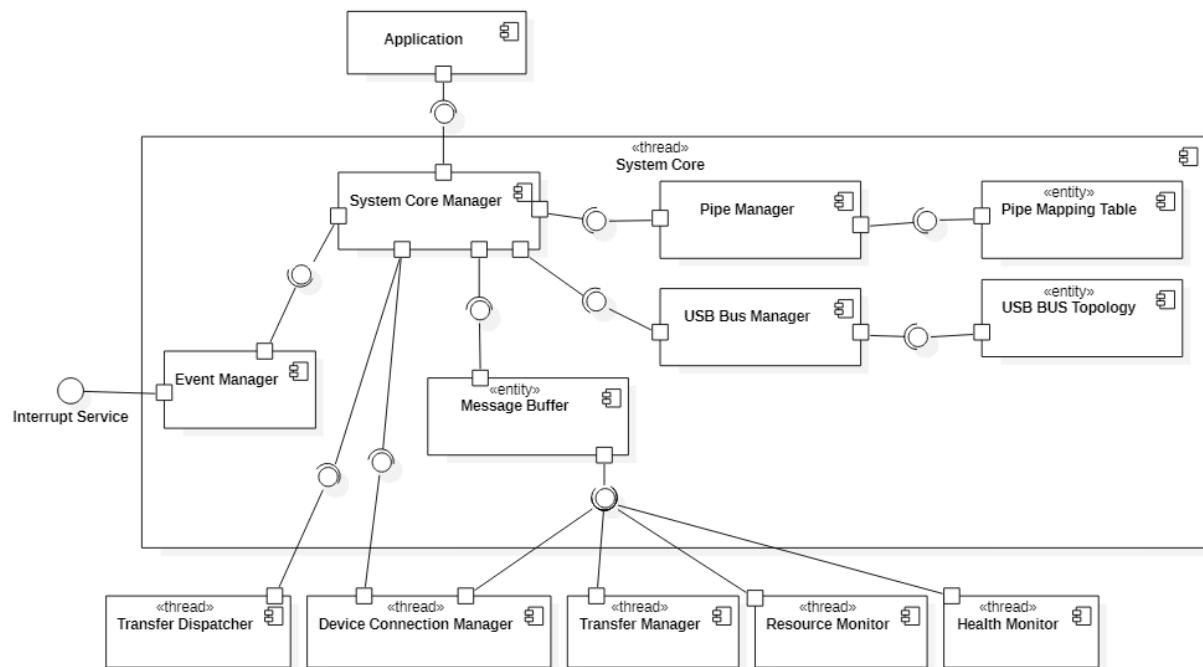


그림 13 System Core Thread의 Component Diagram

**System Core Manager**는 Application과 Interface 역할을 하며, Application으로부터 데이터 전송 요청을 전달받아 시스템 내부 컴포넌트로 전달하는 역할을 한다. 또한, 디바이스 연결, 데이터 전송 완료와 같은 USB 동작 관련 이벤트 발생시 Application으로 전달하는 역할도 수행한다.

**Pipe Manager**와 **Pipe Mapping table**은 Pipe와 endpoint 간의 매팅 정보를 관리하는 역할을 한다. USB 통신을 위해 시스템은 <Pipe, Device Address, endpoint, application> 정보를 관리하며, 데이터 전송시 Application이 요청한 pipe에 해당하는 endpoint를 Pipe Mapping Table에서 검색하여 System Core Manager로 전달한다. 시스템은 효율적인 Pipe Mapping 정보 관리를 위해 Mapping 정보에 대한 Cache와 전체 Mapping 정보 Table을 함께 관리한다.

**USB Bus Manager**와 **USB Bus Topology**은 USB System에 연결된 전체 디바이스의 Topology 정보를 관리한다. USB BUS Topology는 디바이스 혹은 HUB의 상태 변경시(디바이스 연결/해제, Idle 상태 변경 등) 영향을 받는 전체 디바이스를 검색하기 위한 정보로 사용된다.

**Event Manager**는 Host Controller가 전달하는 이벤트를 처리하는 역할을 한다. 이벤트의 종류에 따라 필요한 경우 System Core Manager에 요청하여 추가적인 동작을 수행하도록 한다.

**Message Buffer**는 System Core Thread로 전달되는 외부 Thread의 요청을 보관하는 역할을 한다. System Core Manager는 Message Buffer에 저장된 요청을 순차적으로 꺼내서 처리한다.

다음은 System Core의 주요 기능별 동작 흐름과 내부 컴포넌트의 동작에 대한 설명이다.

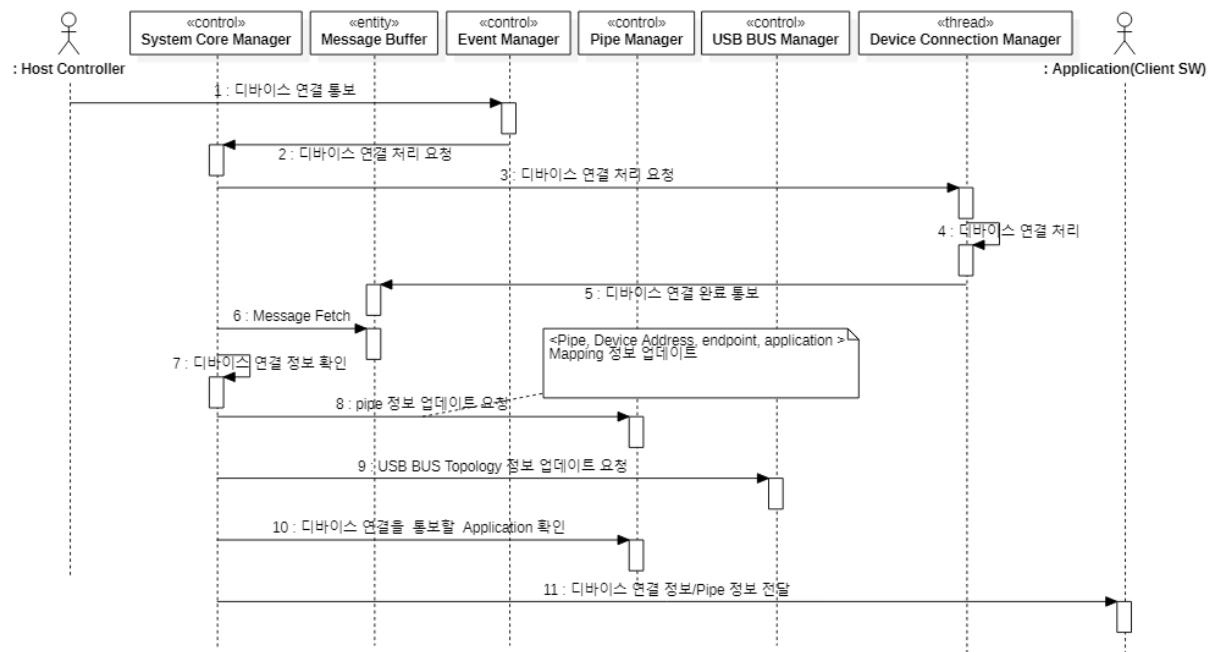


그림 14 System Core의 디바이스 연결 처리 Sequence

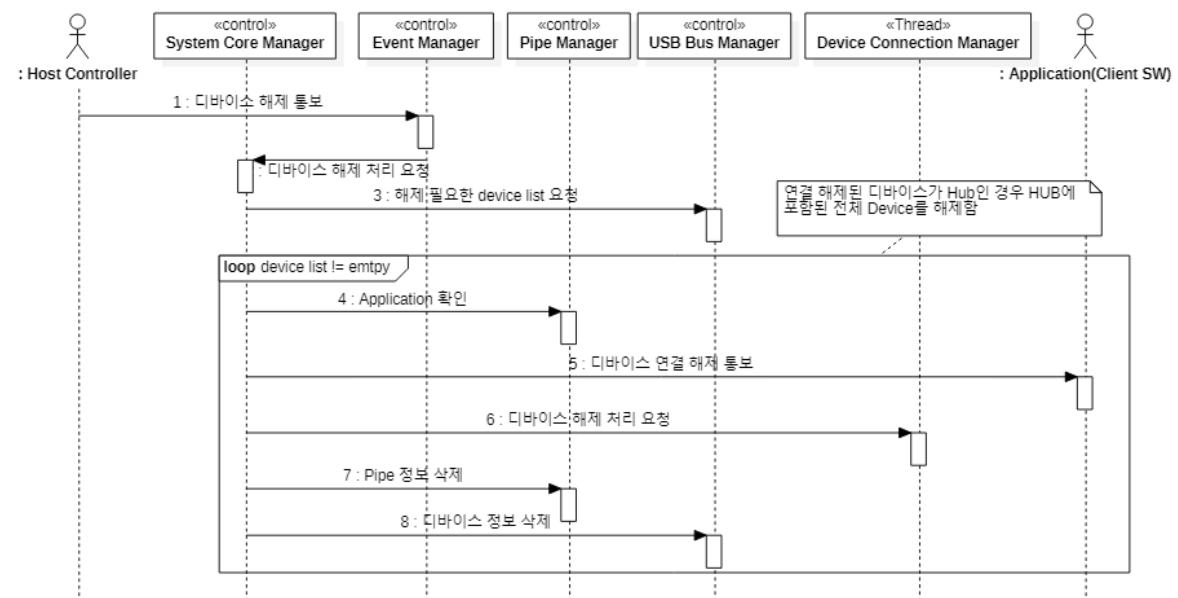


그림 15 System Core의 디바이스 연결 해제 처리 Sequence

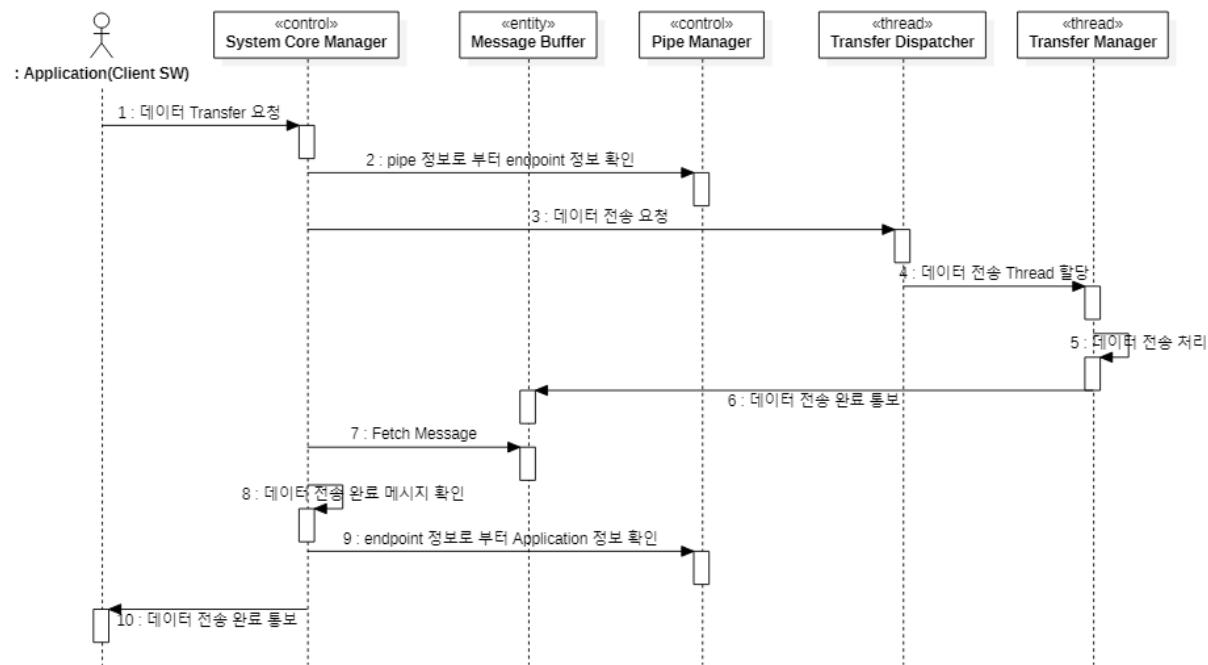


그림 16 System Core의 데이터 전송 요청 처리 Sequence

### 3.2.3. Device Connection Manager

다음 그림은 Device Connection Thread를 구성하는 세부 컴포넌트와 컴포넌트 사이의 연결 관계를 나타내고 있다.

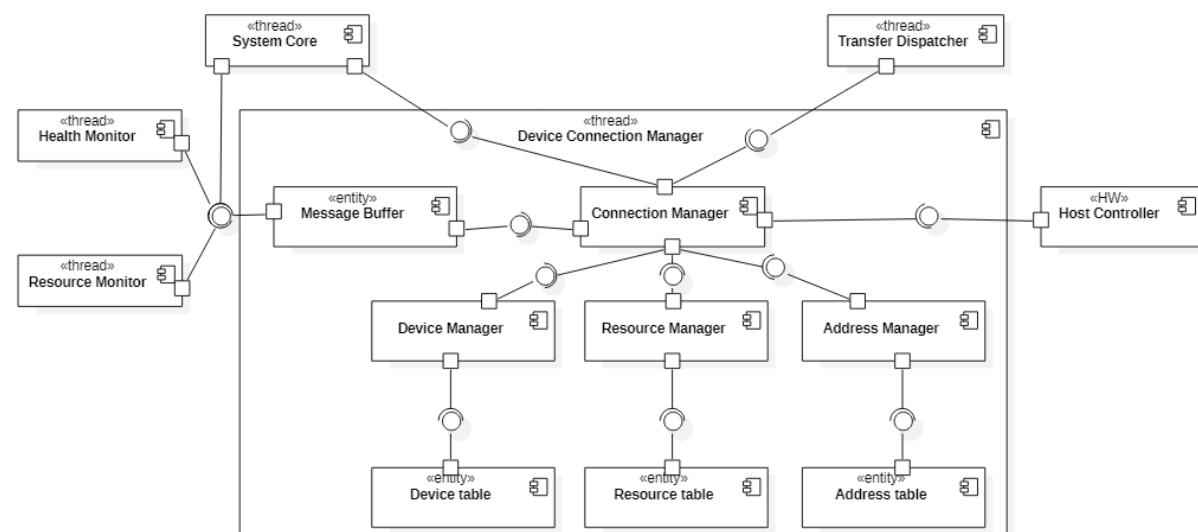


그림 17 Device Connection Manager의 Component diagram

**Connection Manager**는 System Core가 전달하는 디바이스 연결 요청에 대한 메인 Control 역할을 수행한다. 또한, 디바이스 연결을 위한 Emulation과정에 대한 처리 Logic를 포함하고 있으며, 디바이스 연결에 필요한 Control Message를 생성하여 Host Controller로 전달한다.

**Device Manager**와 **Device Table**은 연결된 디바이스와 관련된 정보를 관리하는 역할을 한다. 시스템이 디바이스와 연결하기 위해서는 디바이스에 할당된 Address, Description, Configuration, 디바이스 상태, 사용하는 Application 정보 등과 같은 많은 정보들이 관리되어야 한다. 이러한 정보들을 각각의 컴포넌트들이 개별 관리하는 경우 정보의 동기화 및 관리의 복잡성이 발생할 수 있기 때문에 Device Manager는 해당 정보들을 통합하여 관리하며, 외부 컴포넌트는 Device Manager를 통해 해당 정보를 접근할 수 있다.

**Resource Manager**와 **Resource Table**은 USB 전체 System의 자원과 각 디바이스에 할당된 자원을 관리하는 역할을 한다. Connection Manager는 디바이스와 연결을 설정하기 위해서 Power, Bandwidth 등의 USB BUS의 자원을 Resource Manager를 통해 할당 받고, 연결이 종료된 디바이스에 대한 자원 회수를 요청한다.

**Address Manager**와 **Address Table**은 각 디바이스에 할당되는 Address 정보를 관리한다. 시스템은 디바이스 Address를 “**가용 Address 목록**”과 “**할당 Address 목록**”으로 관리하며 Connection Manager의 요청에 의해 새로 연결된 디바이스에 Address를 할당하거나 해제된 디바이스의 Address를 회수하는 역할을 한다.

Device Connection Manager의 주요 기능별 동작 흐름과 내부 컴포넌트의 동작은 다음과 같다.

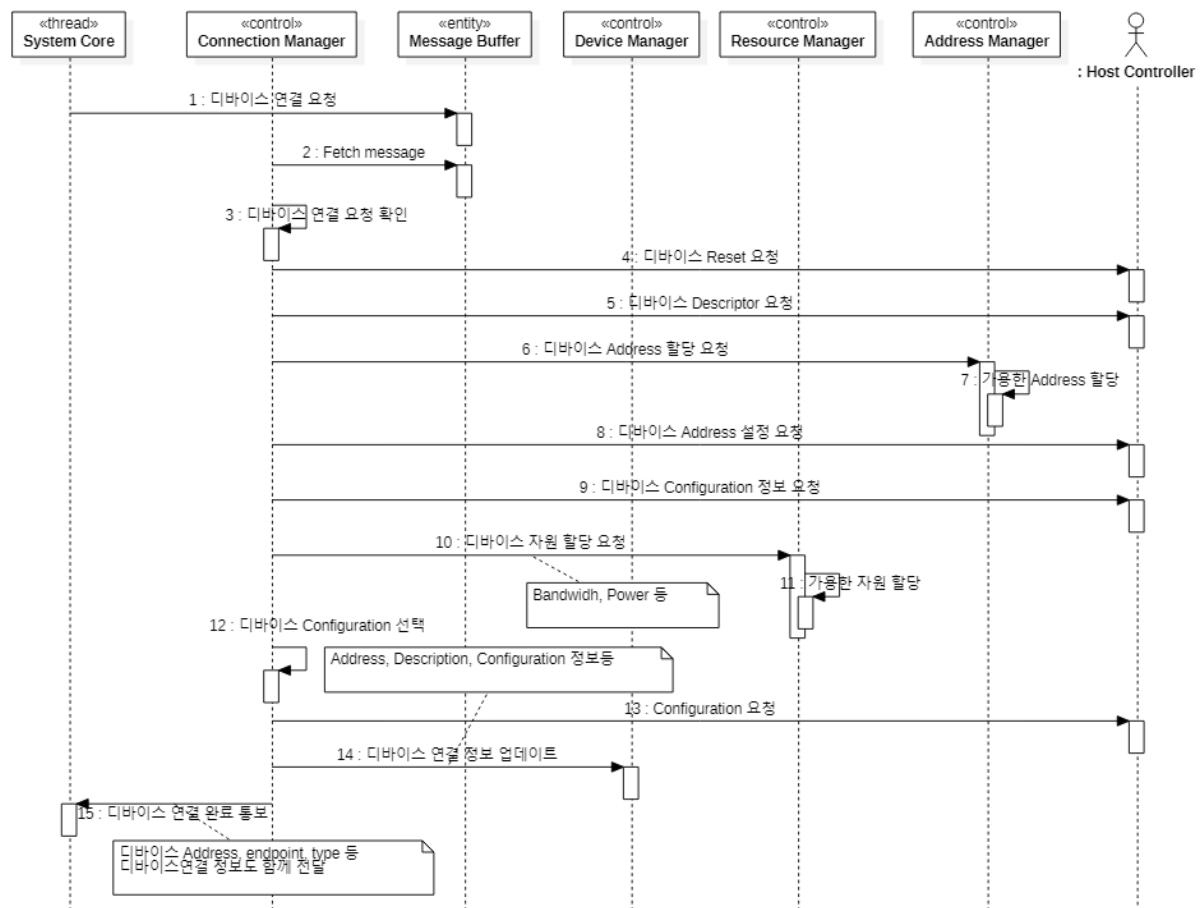


그림 18 Device Connection Manager의 디바이스 연결 처리 Sequence

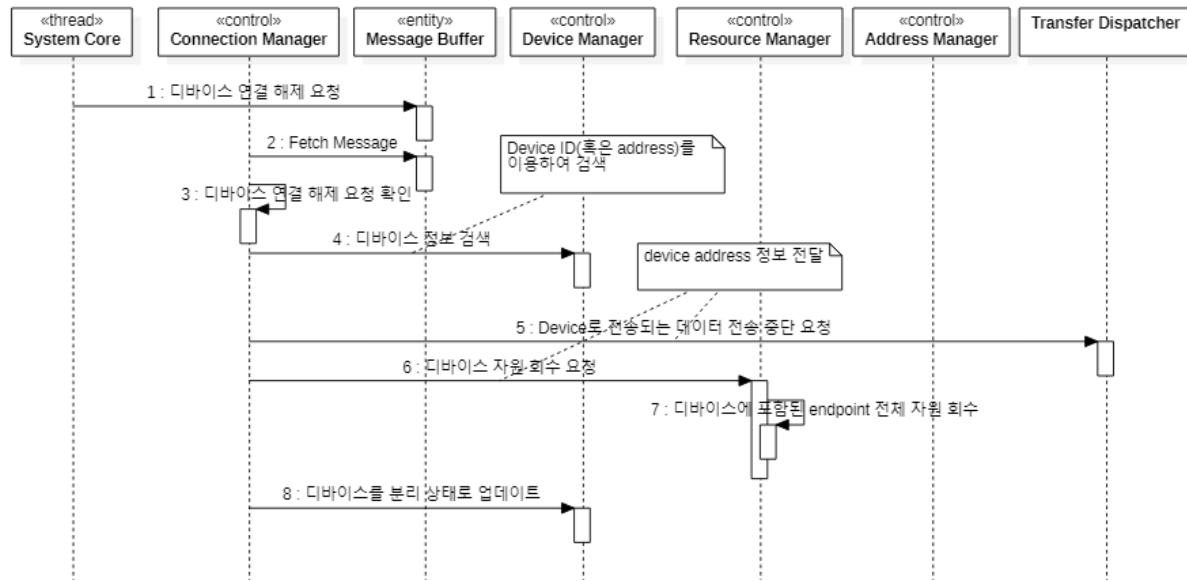


그림 19 Device Connection Manager 디바이스 연결 해제 Sequence

### 3.2.4. Transfer Dispatcher

다음 그림은 Transfer Dispatcher Thread를 구성하는 세부 컴포넌트와 컴포넌트 사이의 연결 관계를 나타내고 있다.

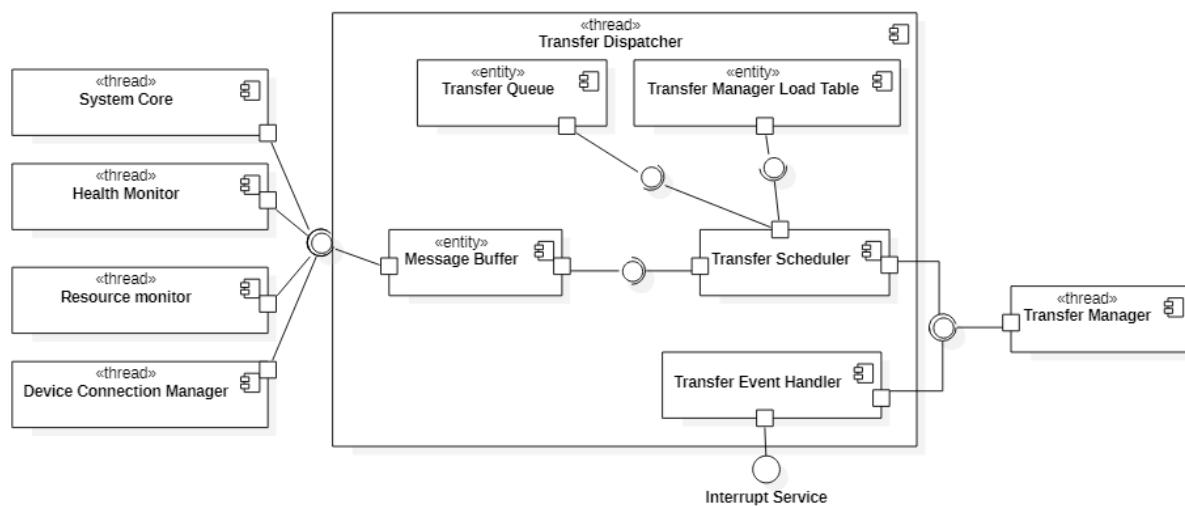


그림 20 Transfer Dispatcher의 Component diagram

**Transfer Scheduler**는 Message Buffer를 통해 전달된 Transfer 요청의 우선 순위를 정하고, Transfer

Manager Thread에 요청을 할당하는 역할을 한다. Transfer Manager에 요청을 할당할 때는 Transfer Manager Load Table에 저장된 정보를 확인하고, 각 Transfer Manager Thread의 Load가 분산되도록 Load balancing을 고려하여 할당한다. 또한, Device connection Manager로부터 디바이스 연결이 해제된 디바이스가 통보되는 경우 Transfer Queue에 포함된 해당 디바이스와 관련된 전송 요청을 모두 삭제하는 등의 Transfer Queue 관리 기능을 한다.

**Transfer Queue**는 Message Buffer를 통해 전달된 요청 중 Transfer 요청들을 우선 순위를 정해 저장하고 있는 자료구조이다. Transfer Dispatcher로 다수의 Transfer 요청이 동시에 전달되는 경우 Transfer Scheduler는 요청의 우선 순위를 정해 Transfer Queue에 저장하고, 우선 순위에 따라 Transfer 요청을 순차적으로 처리한다.

**Transfer Manager Load Table**은 Transfer를 처리하는 각 Thread의 Load 정보를 저장하는 자료 구조이다. Transfer Manager Thread의 Load를 결정하는 주요 정보는 Thread의 Message Buffer의 현재 길이, 전달된 데이터 전송의 종류 등이다.

**Transfer Event Handler**는 Transfer 과정에 발생하는 Host Controller의 Interrupt를 처리하는 역할을 한다. Host Controller는 transaction에 interrupt flag가 설정된 경우 해당 transaction이 완료된 시점에 interrupt를 발생시킨다. 따라서 SW는 경우에 따라 transaction 완료 시점에 interrupt를 받을 수 있도록 설정할 수 있다. 특히, Bulk 전송과 같이 Asynchronous 전송의 경우 완료 시점을 SW가 알 수 없기 때문에 Transfer 마지막 transaction에 interrupt를 설정하여 Transaction 완료를 확인할 수 있다. Transfer Event Handler는 이와 같은 전송 과정에 발생할 수 있는 interrupt를 수신하고, 해당 interrupt를 처리할 수 있는 Thread를 확인하여 interrupt 발생을 통보하는 역할을 한다.

다음은 Transfer Dispatcher의 주요 기능별 동작 흐름과 내부 컴포넌트의 동작에 대한 설명이다.

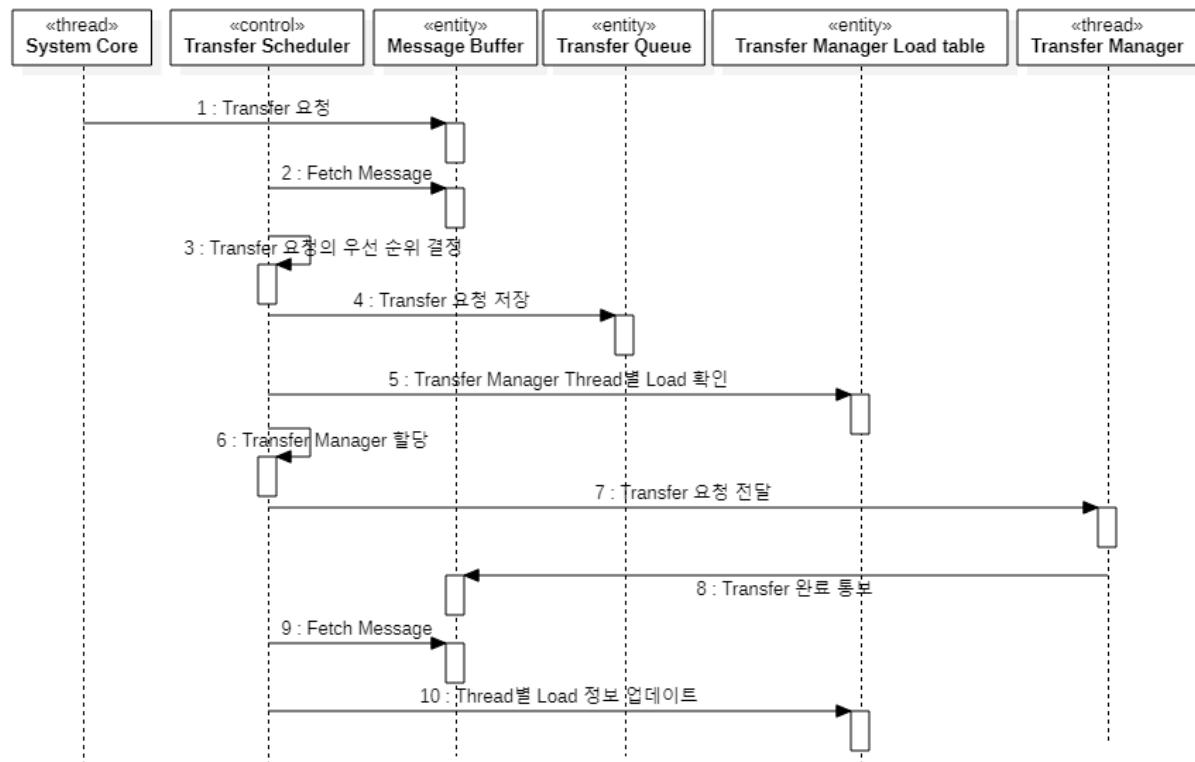


그림 21 Transfer Dispatcher의 Transfer 요청 처리 Sequence

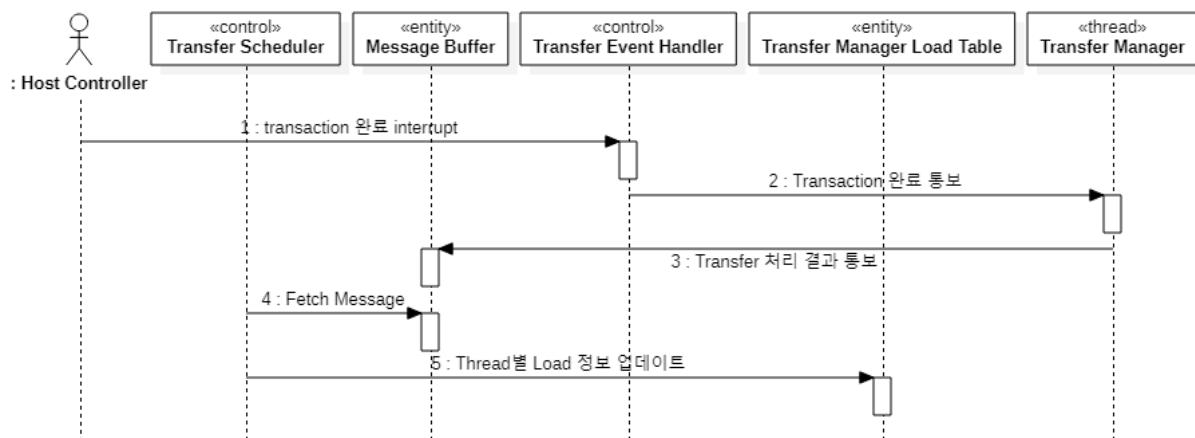


그림 22 Transfer Dispatcher의 Interrupt 처리 Sequence

### 3.2.5. Transfer Manager

다음 그림은 Transfer Manager Thread를 구성하는 세부 컴포넌트와 컴포넌트 사이의 연결 관계를 나타내고 있다.

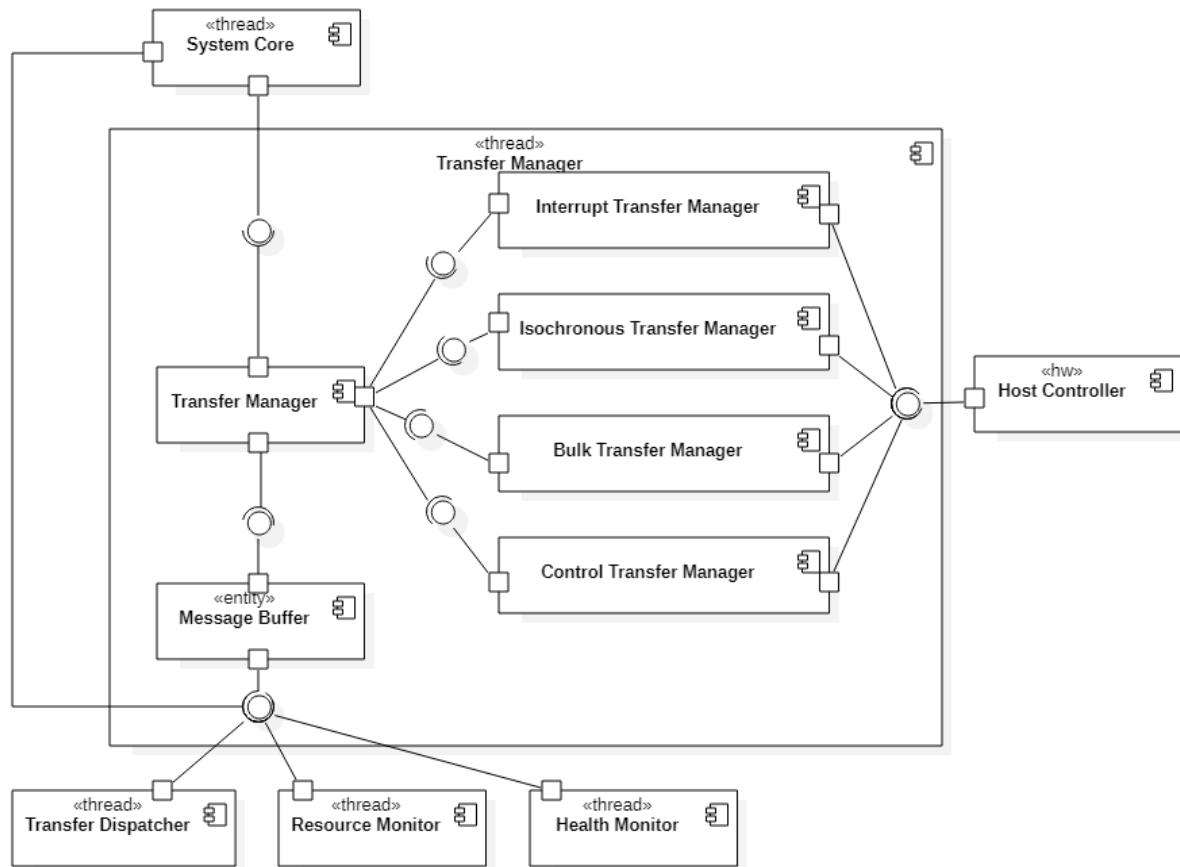


그림 23 Transfer Manager의 Component diagram

**Transfer Manager**는 데이터 전송을 처리하는 Main component로 Message Buffer를 통해 전달된 데이터 전송 요청을 순차적으로 처리하는 역할을 한다. 데이터 전송은 특징에 따라 Interrupt, Isochronous 전송과 같은 Periodic 방식, Bulk/Control 전송과 같은 Non-Periodic 방식으로 구분할 수 있다. Transfer Manager는 전송 타입 따라 전송을 담당할 컴포넌트를 선택하고 전송을 위임한다. 또한, 전송이 완료된 transaction에 대한 처리 완료 메시지가 전달된 경우 이를 처리하고, System Core로 전송 처리 결과를 전달한다.

**Interrupt Transfer Manager**는 Interrupt type의 전송 요청을 처리한다. Interrupt 전송 처리를 위해 요청된 주기에 따라 transaction을 생성하여 Host Controller로 전달한다.

**Isochronous Transfer Manager**는 Isochronous type의 전송 요청을 처리한다. Isochronous 전송의 경우도 interrupt 방식과 동일하게 주기적인 데이터 전송이 필요하기 때문에 전달된 데이터를 전송 가능한 크기로 분할하여 transaction을 생성하고, 이를 Host Controller로 전달하는 역할을 한다.

**Bulk Transfer Manager**는 Bulk type의 전송을 처리한다. 전달된 Transfer 요청의 크기가 Transaction의 크기보다 큰 경우 Data를 fragment 하여 transaction 단위로 분할하고, 생성된 transaction을 Host Controller에 전달한다. Bulk 전송의 경우 별도의 전송 주기가 없기 때문에 생성된 transaction을 Host Controller와 공유하는 Buffer에 순차적으로 추가한다.

**Control Transfer Manager**는 Control type의 전송을 처리한다. Bulk type의 전송과 동일하게 별도의 주기 설정 없이 생성되는 transaction을 순차적으로 Host Controller와 공유하는 Buffer에 추가한다.

Transfer Type별 Transfer Manager의 주요 동작 흐름과 내부 컴포넌트의 동작은 다음과 같다.

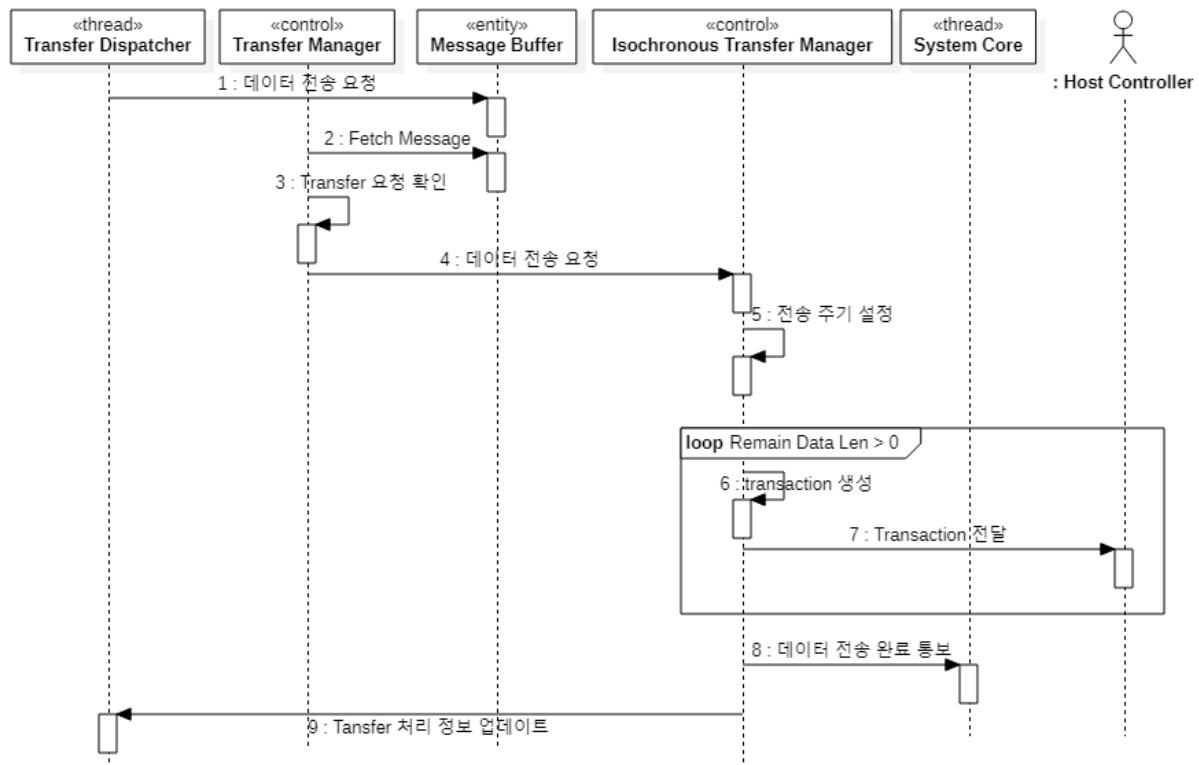


그림 24 Isochronous 데이터 전송 sequence

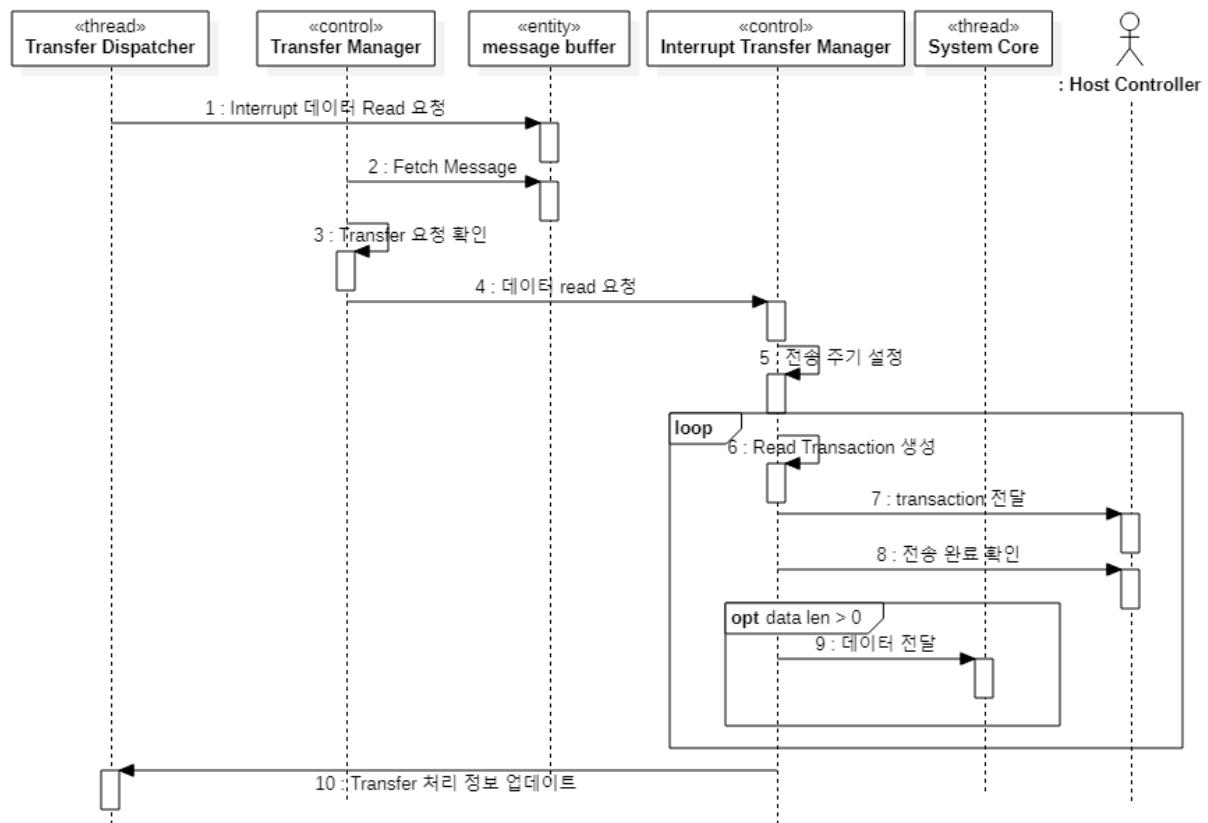


그림 25 Interrupt 데이터 전송 sequence

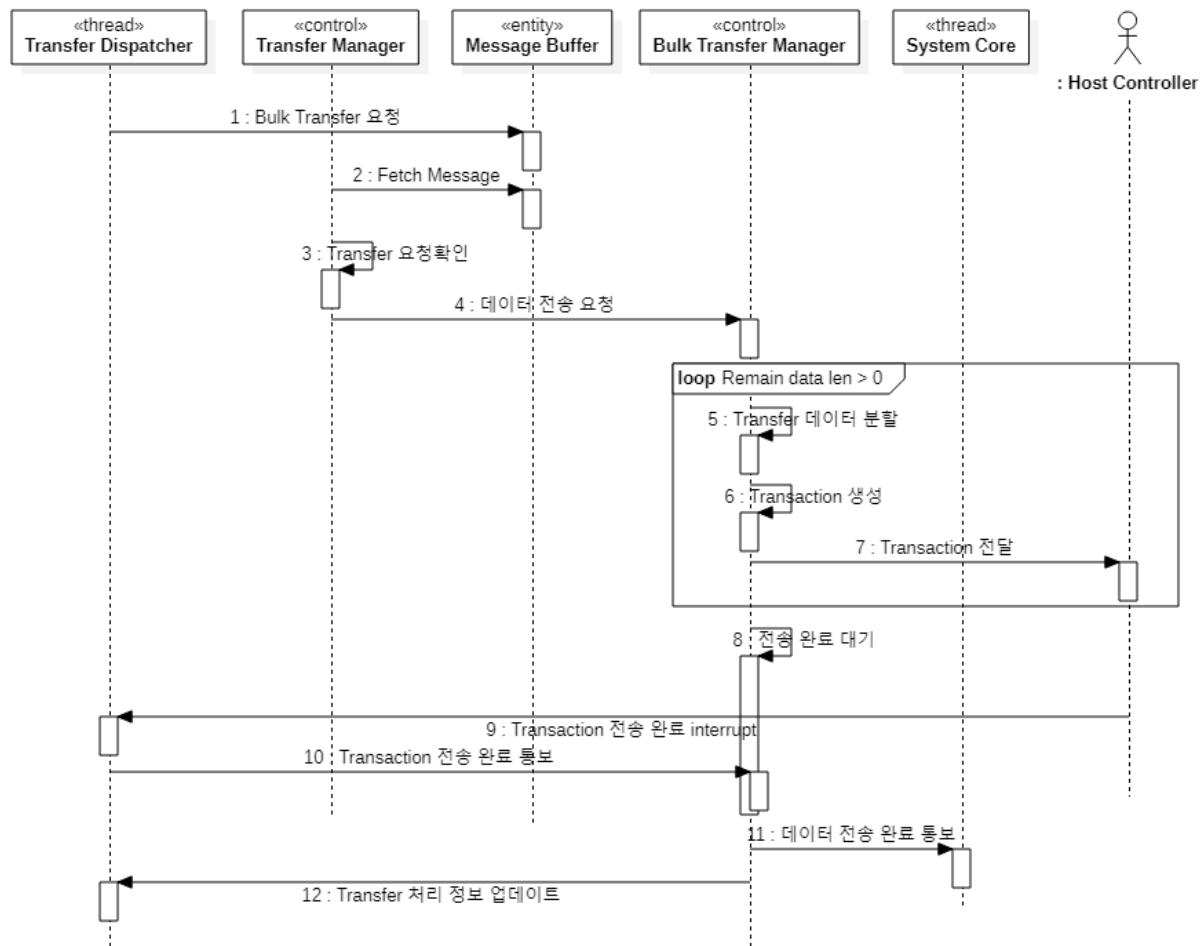


그림 26 Non-Periodic 전송 Sequence(Bulk Transfer)

### 3.2.6. Health Monitor

다음 그림은 Health Monitor Thread를 구성하는 세부 컴포넌트와 컴포넌트 사이의 연결 관계를 나타내고 있다.

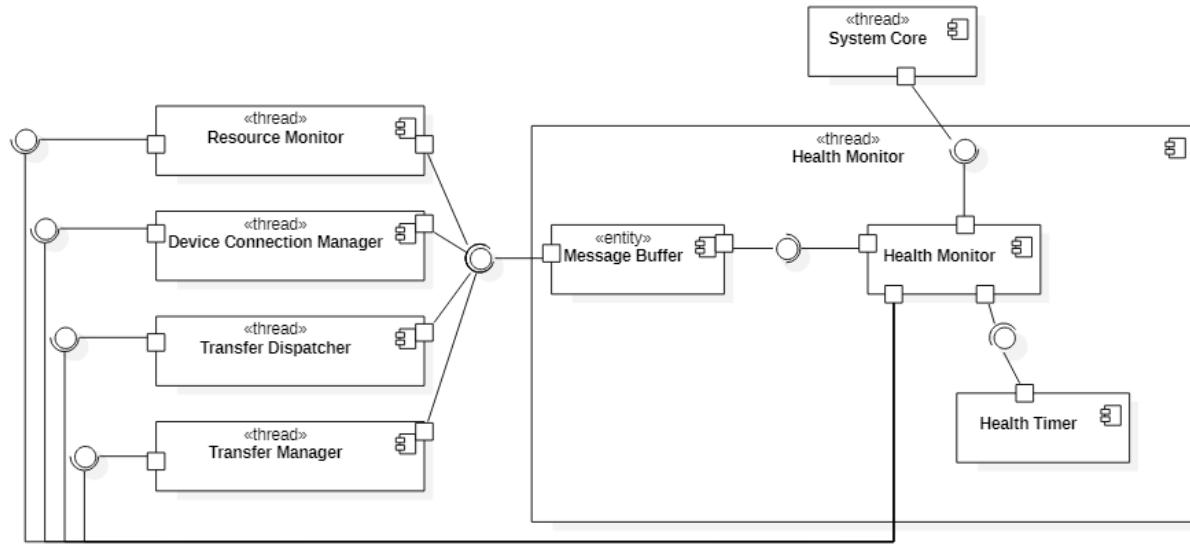


그림 27 Health Monitor의 Component diagram

**Health Monitor**는 각 Thread의 정상 동작 여부를 판단하기 위해 각 Thread로 Ping Message를 전달하고, 그에 대한 응답을 확인하여 시스템 내부 Thread의 정상 동작 여부를 판단하는 역할을 한다. Health Monitor는 지정된 timeout 시간동안 응답이 전달되지 않은 Thread가 발견된 경우 Thread 복구 요청을 System core로 전달한다.

**Health Timer**는 Health Monitor의 요청에 따라 Ping Message를 전달하는 주기와 각 Ping Message에 대한 Timeout 시간을 관리하는 역할을 한다. Timeout 시간을 초과하 Alarm이 발생한 경우 Alarm 발생을 Health Monitor에 통보한다.

Health Monitor의 동작 흐름과 내부 컴포넌트의 동작은 다음과 같다.

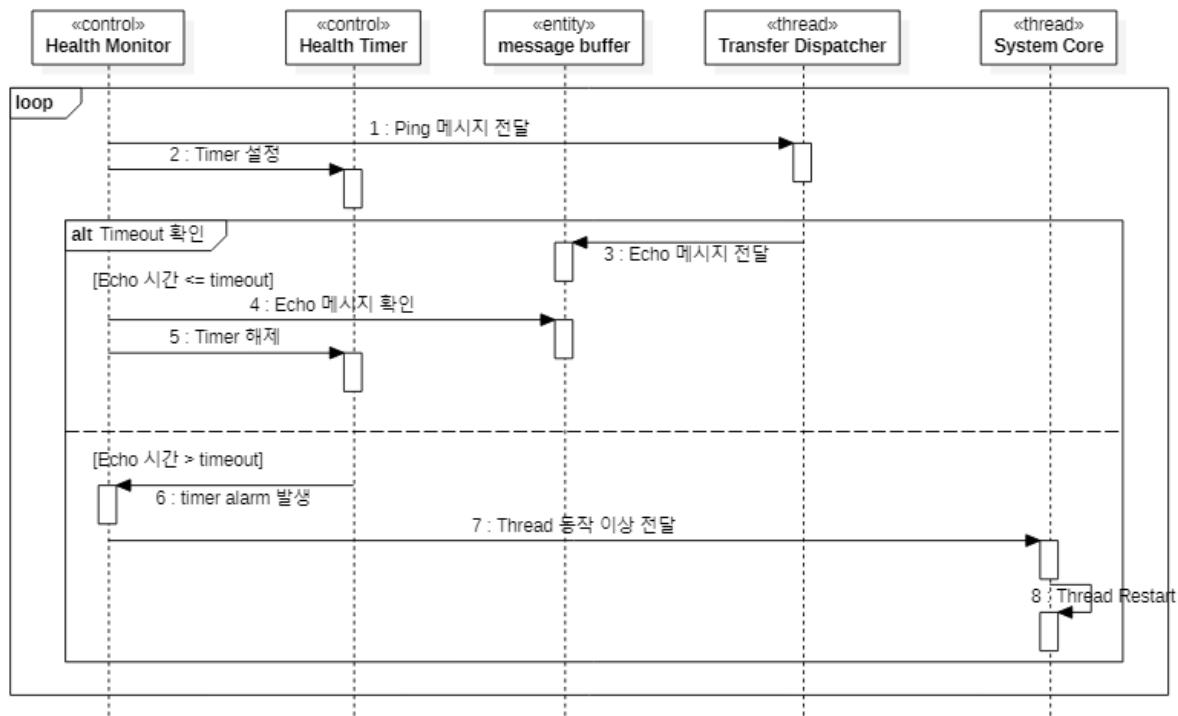


그림 28 Health Monitoring Sequence Diagram(Transfer Dispatcher Monitoring 예)

### 3.2.7. Resource Monitor

다음 그림은 Resource Monitor Thread를 구성하는 세부 컴포넌트와 컴포넌트 사이의 연결 관계를 나타내고 있다.

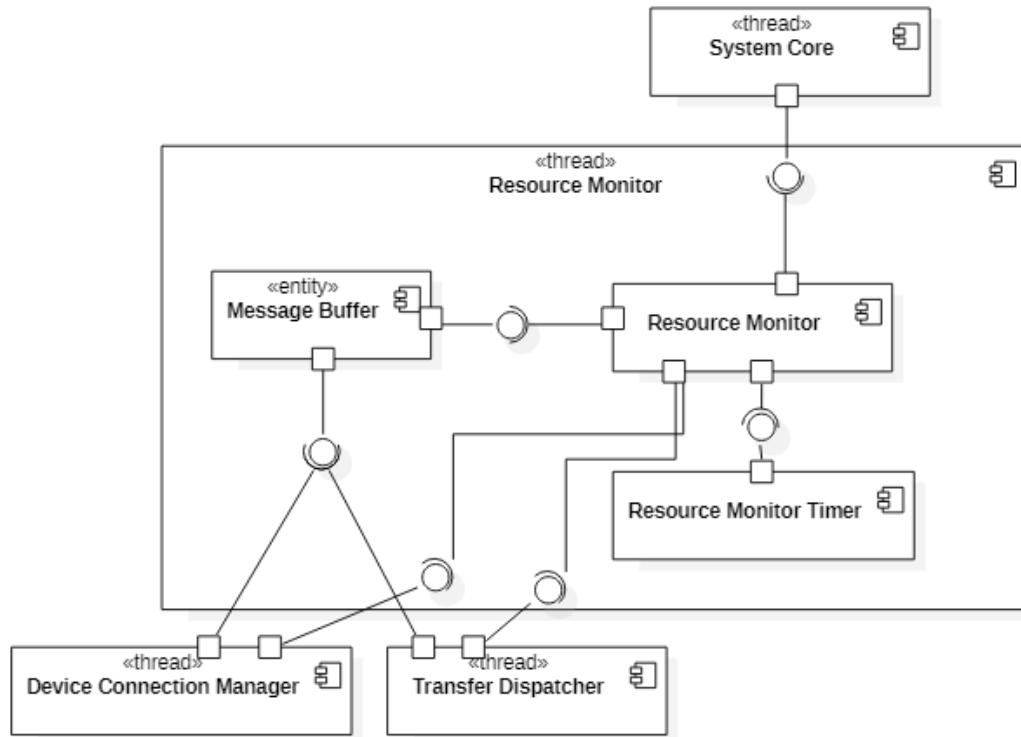


그림 29 Resource Monitor의 Component Diagram

**Resource Monitor**는 디바이스에 할당된 자원이 낭비되는지를 모니터링 하고, 낭비되는 자원이 발견된 경우 이를 System Core로 통보하여 자원을 재분배할 수 있도록 요청하는 역할을 한다. 이를 위해 Resource Monitor는 Device Connection Manager를 통해 각 디바이스별 할당된 자원의 양을 수집한다. 또한, Transfer Dispatcher를 통해 각 디바이스가 사용하고 있는 자원의 양을 실시간으로 수집한다. 디바이스에 할당된 자원과 실제 사용중인 자원의 양이 임계치 이상으로 차이가 나는 경우 Resource Monitor는 비정상 적인 상태로 판단하고 System Core에 자원의 재분배를 요청한다.

**Resource Monitor Timer**는 자원 모니터링 주기를 관리한다. Resource Monitor는 Device Connection Manager와 Transfer Dispatcher로부터 정보를 수집할 주기를 Resource Monitor Timer를 통해 설정하고 Alarm 발생시 자원 수집 요청 메시지를 각 Thread로 전달한다.

Resource Monitor의 주요 기능별 동작 흐름과 내부 컴포넌트의 동작은 다음과 같다.

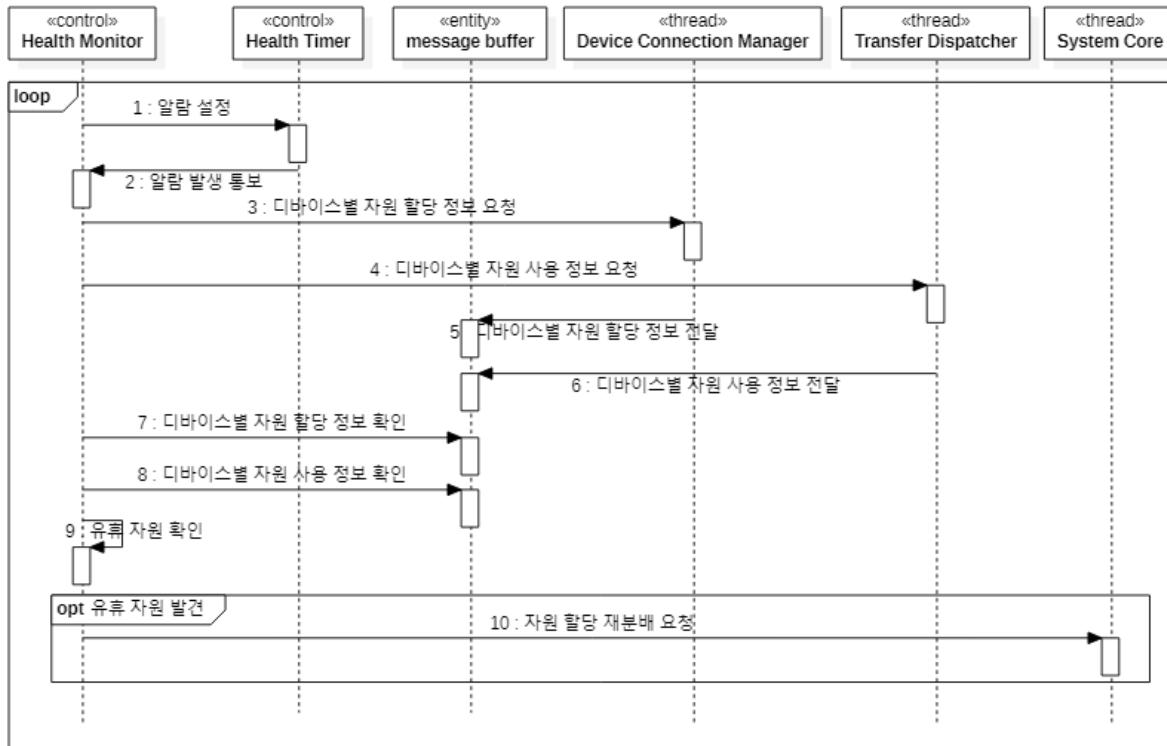


그림 30 Resource Monitoring Sequence Diagram

### 3.3. 시스템 구조 특징 분석

#### 3.3.1. 강점

- 성능:** 본 시스템은 데이터 전송 및 디바이스 연결 시간 등과 관련된 성능을 향상시키기 위한 구조들을 중심으로 설계하였다. 다수의 디바이스가 동시에 사용될 수 있는 환경을 고려하여, 동시에 여러 개의 데이터 전송 요청을 처리하기 위해 전송 Thread를 별도로 분리하여 병렬처리 할 수 있도록 하였다. 또한, 각 Thread간의 데이터 교환은 Message Buffer를 기반으로 통신하도록 하여 Thread간의 의존성을 낮추고, 각 Thread가 병렬적으로 동작 할 수 있도록 구성하였다.
- 변경 용이성:** 변경 용이성과 관련해서는 2가지 관점의 구조적 고려사항을 반영하여 설계하였다. 첫 번째는 내부 컴포넌트의 변경을 용이하게 하기 위한 것으로, 내부 Thread간의 통신을 Message Buffer를 기반으로 독립적으로 동작하게 함으로써, Thread간의 상호의존성을 최소화하였다. 이를 통해 각 Thread 기능을 개별적으로 변경하거나 업데이트 할 수 있도록 구성

하였다. 두 번째는 외부 실행 환경 변경에 따른 시스템 변경을 용이하게 하기 위한 것으로, 외부와 의존하는 기능들을 별도 Interface를 통해 구현하도록 하였다. OS에 의존적인 기능은 OS Interface를 통해 접근하도록 하고, Host Controller에 의존적인 기능은 별도의 Host Controller Interface를 추가하였다. 이를 통해 시스템은 해당 Interface를 통해 외부 시스템을 접근하게 함으로써, 외부 시스템의 구체적인 구현 사항에 관계없이 동작 할 수 있도록 하였다.

### 3.3.2. 단점 및 위험 요인

- **공유자원 동기화 및 공유 자원 관리:** 시스템의 성능을 향상시키기 위해 여러 Thread가 Message Buffer를 기반으로 독립적으로 동작하게 함으로 인해 공유 자원 접근에 대한 동기화 처리에 대한 SW 복잡도가 증가하게 된다. 이러한 문제를 보완하기 위해 공유자원 별로 해당 자원을 관리하는 별도의 Management 컴포넌트를 추가하여 공유 자원 접근에 대한 동기화 문제에 대한 Risk를 감소시키도록 하였다. 또한, 여러 Thread가 개별적으로 동작하면서 발생할 수 있는 시스템 내부적인 오류 요인들을 감소시키기 위하여 별도의 Monitoring 시스템을 도입하여 공유 자원 관리의 Risk를 보완하고 시스템 안정성을 향상시킬 수 있도록 하였다.

## 4. 모듈 사양

### 4.1. 전체 모듈 구조

시스템의 전체 모듈 구조는 다음 그림과 같은 **Layered 구조**로 2개의 Layer로 구성된다. Core Driver Layer는 USB Driver 전체 시스템을 관리하는 기능을 구현하며, Host Controller Driver Layer는 Core Driver Layer의 요청을 Host Controller(HW)를 제어하여 실행하는 기능을 구현한다. 추가적으로 OS의 기능에 의존적인 부분들은 OS Abstraction을 담당하는 별도의 Package로 구현하여 다른 모듈과의 의존성을 줄일 수 있도록 설계하였다.

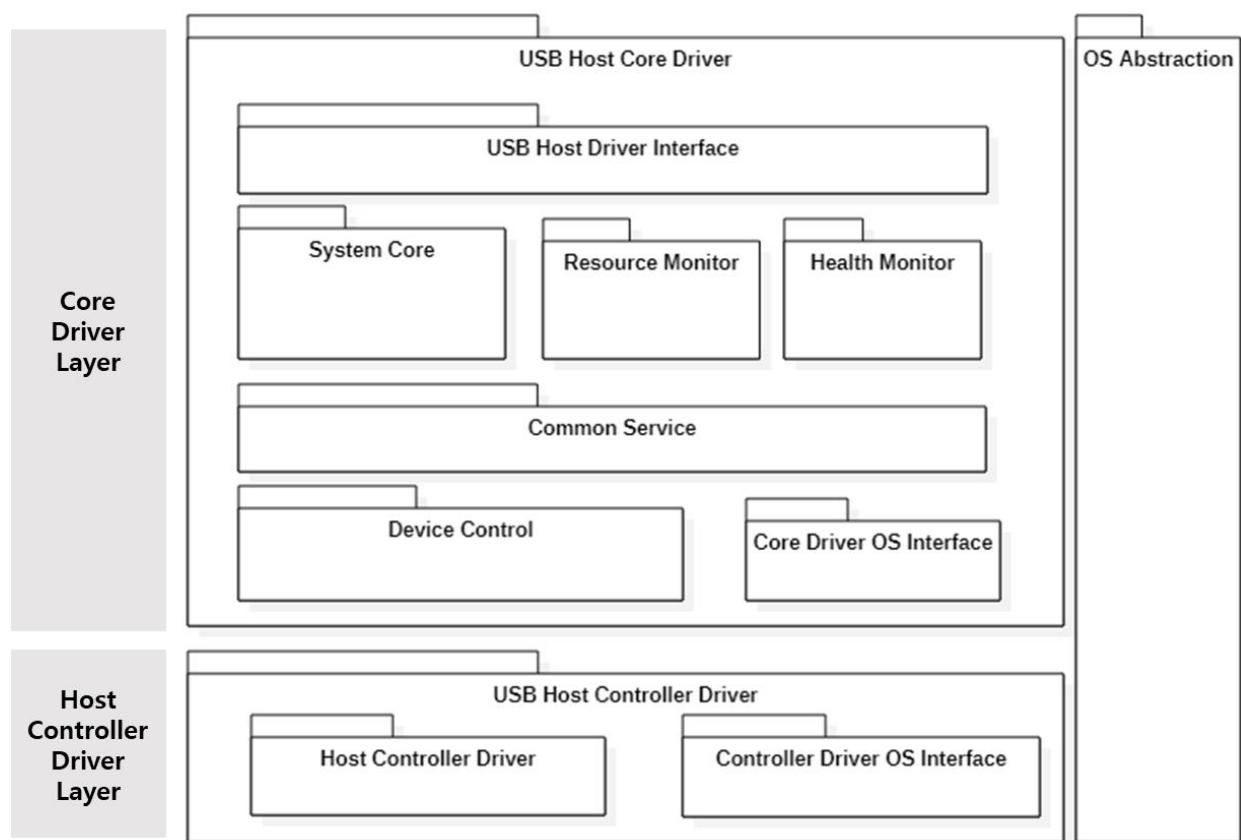


그림 31 시스템 전체 Package 구조

시스템을 구성하는 각 Package의 구현 기능은 다음과 같다.

- **USB host Driver Interface:** USB Host Controller Driver가 외부 Application과 통신하기 위한 Interface를 구현한다. 또한, 디바이스 클래스별 지원기능을 별도의 내부 모듈로 구현하여 디바이스 클래스에 대한 전용 Interface를 제공할 수 있도록 한다.

- **System Core:** USB Host Driver의 Main Control 기능을 구현한다. Application에서 전달되는 요청과 시스템 내부 모듈에서 발생하는 다양한 요청을 중개하는 기능 등을 구현한다.
- **Resource Monitor:** 시스템에 연결된 각 디바이스가 사용하는 자원(Bandwidth, Power 등)을 모니터링 하고, 유휴 자원 확인시 System Core에 통보하는 기능을 구현한다.
- **Health Monitor:** 시스템 내부의 각 Thread의 정상 동작 여부를 확인할 수 있는 모니터링 기능을 구현한다. 주기적인 모니터링과 ping message에 대한 timeout check를 위해 Common Service Package의 Timer 기능을 이용한다.
- **Common Service:** 시스템 내부에서 공통적으로 사용되는 Utility 기능들을 구현한다. 주기적인 Monitoring 등에 사용되는 Timer 기능, 데이터 우선 순위 확인을 위한 Sorting, 메시지 저장장에 사용되는 Message Buffer 기능과 같은 알고리즘과 자료구조 등을 구현한다.
- **Device Control:** 디바이스 연결 관리, 데이터 전송과 같은 Host Controller(HW)를 사용하는 USB Host Driver의 주요 기능을 구현한다. 디바이스 연결에 필요한 USB 자원 관리, 데이터 전송을 위한 Transfer 요청 관리와 같은 기능들의 구현도 포함한다.
- **Core Driver OS Interface:** Host Core Driver Layer에서 사용하는 OS 기능에 대한 Interface를 정의한다.
- **Host Controller Driver OS Interface:** Host Controller Driver Layer에서 사용하는 OS 기능에 대한 Interface를 정의한다.
- **Host Controller Driver:** USB Host Controller(HW)를 직접 제어하는 기능을 구현한다. 상위 Layer에 대한 변경 영향성을 줄이기 위해 USB Host Core Driver Layer의 Interface Package에 정의된 Host Controller Interface 정의를 상속하여 패키지 내부 모듈을 구현한다.
- **OS Abstraction:** OS의 기능을 추상화 하여 구현한다. OS의 기능은 각 Layer가 공통적으로 사용하기 때문에, 각 Layer가 정의한 OS Interface를 다중 상속하여 내부 모듈을 구현한다.

## 4.2. 세부 모듈 구조

시스템을 구성하는 각 패키지에 포함된 모듈의 세부 구조는 다음 그림과 같다.

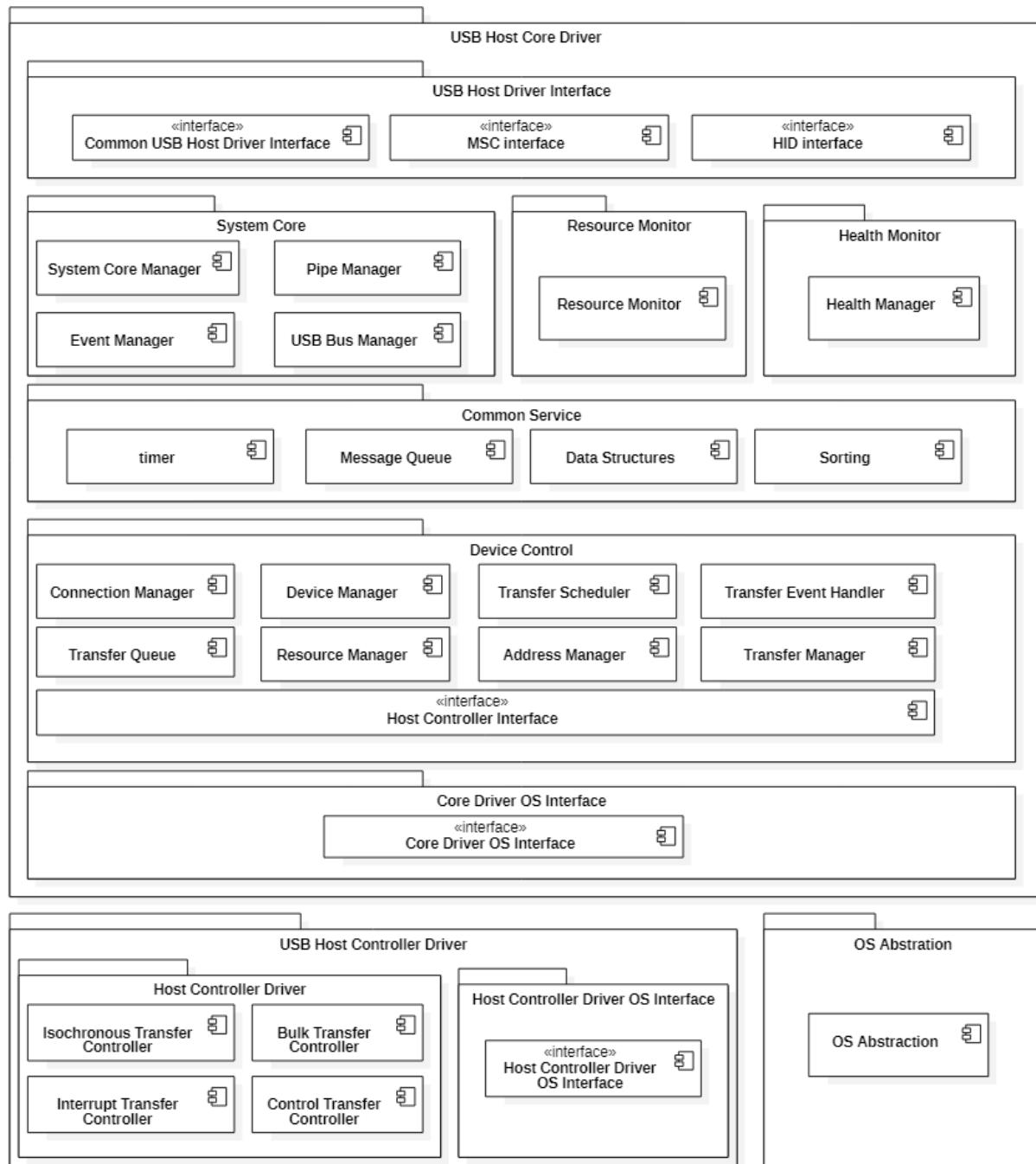


그림 32 시스템 Module View

**USB Host Driver Interface 패키지는** 시스템이 외부 Application에 제공하는 Interface를 구현한다. 일반적인 USB Host Driver 기능을 제공하는 Common USB Host Driver Interface와 디바이스 Class별로 특화된 Interface 기능을 제공하는 MSC Interface, HID Interface 기능을 별도 모듈로 분리하였다.

**System Core 패키지**는 시스템의 Main Control 기능을 구현한다. 이를 위해 시스템 내부와 외부의 요청을 중개하는 System Core Manager, 외부 Event를 처리하는 Event Handler 등을 포함한다.

**Resource Monitor**과 **Health Monitor 패키지**는 각 디바이스의 자원 사용량과 각 Thread의 비정상 동작 여부를 확인하기 위한 Resource Monitor, Health Monitor 모듈을 각각 포함한다.

**Common Service 패키지**는 시스템 내부에서 공통적으로 사용하는 기능들을 구현한다. 주기적인 모니터링 등을 위한 Timer, Thread간 메시지 전달을 위한 Message Buffer, 우선 순위 Scheduling 등을 위한 Sorting 알고리즘 모듈 등을 포함한다.

**Device Control 패키지**는 시스템의 USB Host Driver의 주요 기능들을 구현한다. 이를 위해 디바이스 연결을 관리하는 Connection Manager, 데이터 전송을 관리하는 Transfer Manager, Transfer 요청에 대한 우선 순우를 정하고 각 Thread의 load를 분산하는 Transfer Scheduler 등의 모듈을 포함한다.

**Core Driver OS Interface 패키지**는 Host Core Driver 구현에 필요한 OS 기능에 대한 Interface를 정의한다. OS 기능을 추상화한 Core Driver OS Interface 모듈을 포함한다.

**Host Controller Driver 패키지**는 Host Controller를 직접 제어하여 USB Host 기능을 제공하기 위한 모듈을 구현한다. 각 기능의 HW변경으로 인한 영향을 줄이기 위해 USB 전송 Type별로 각각 분리된 모듈을 포함하고 있다. 또한, Host Controller Driver가 사용하는 OS 기능을 정의하기 위한 Host Controller Driver OS Interface를 포함하고 있다.

**Host Controller Driver OS Interface 패키지**는 Host Controller Driver 구현에 필요한 OS 기능에 대한 Interface를 정의한다. OS 기능을 추상화한 Host Controller Driver OS Interface 모듈을 포함한다.

**OS Abstraction 패키지**는 OS가 제공하는 기능을 추상화한 모듈을 구현하며, OS별로 독립된 추상화 모듈을 포함한다.

다음은 각 패키지를 구성하는 세부 모듈에 대한 설명이다.

#### 4.2.1. USB Host Driver Interface

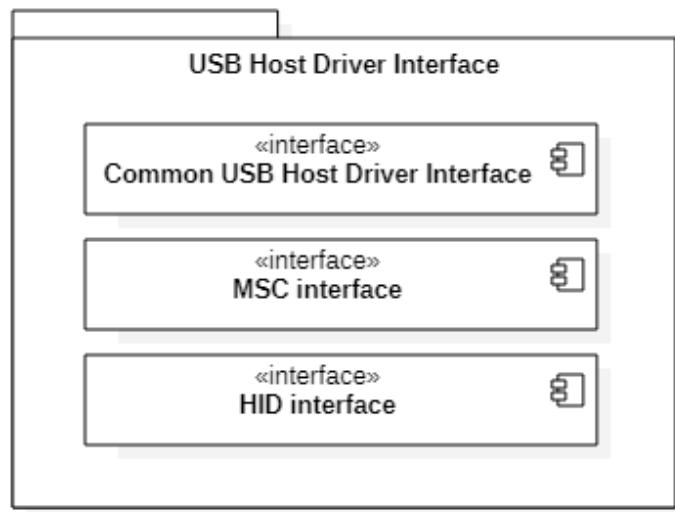


그림 33 USB Host Driver Interface 패키지 구성

**Common USB Host Driver Interface**는 USB Host Driver가 제공하는 공통적인 기능을 정의하고 구현한다. 상위 Application은 USB Host Driver Interface에 정의된 Interface를 이용하여 USB Host Driver의 기능을 이용할 수 있으며, System Core Manager는 Interface를 realization하여 내부 기능을 구현한다.

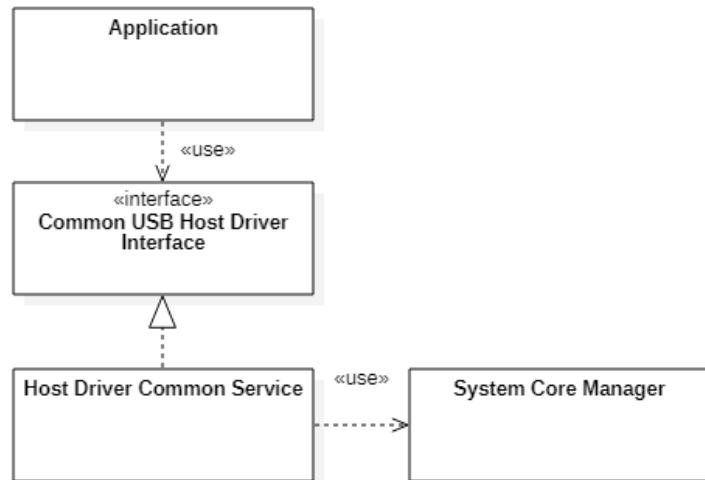


그림 34 Common USB Host Driver Interface 모듈 구조

**MSC Interface**와 **HID Interface**는 Mass Storage Class와 HID Class에 특화된 기능을 제공하기 위한 Interface를 정의한다. Class별 Interface는 USB Host Driver의 Common 기능을 Wrapping(Adaptation)

하는 역할을 하기 때문에 Common Service를 사용하여 구현한다.

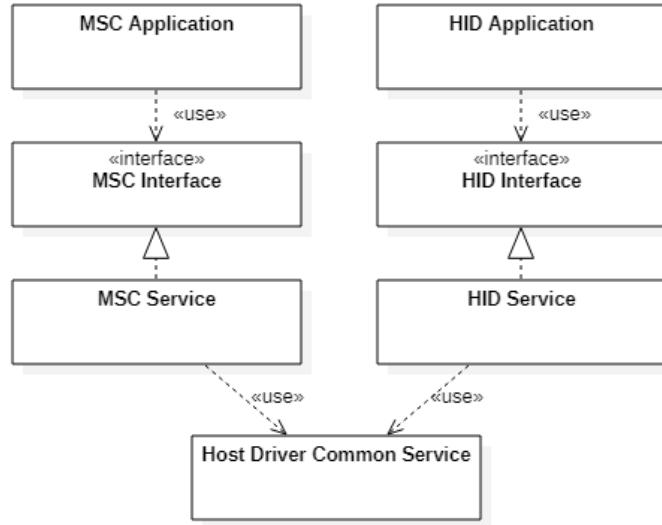


그림 35 MSC/HID Interface 모듈 구조

#### 4.2.2. System Core

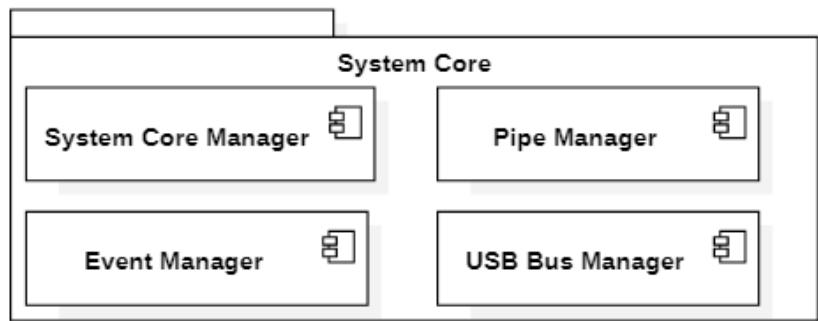


그림 36 System Core 패키지 구성

**System Core Manager**는 시스템 외부 Interface를 통해 전달되는 요청과 시스템 내부에서 발생하는 요청들을 처리하는 Main Control 역할을 구현한다. System Core Manager는 Message Buffer에 저장된 요청을 순차적으로 꺼내 요청에 해당하는 기능을 수행한다. Message Buffer를 통해 전달될 수 있는 명령은 Device Connection Manager로부터 전달되는 디바이스 연결 완료 통보, Health Monitor로부터 전달되는 Thread 이상 처리 요청, Resource Monitor로부터 전달되는 유휴 자원 통보 등과 같이 다양한 요청들이 전달 되 수 있다. 따라서, 각각의 요청 타입 별에 처리 기능을 통합하는 Interface를 추가하고 System Core Manager는 해당 Interface를 이용하여 각 명령을 처리 할 수 있도록 한다.

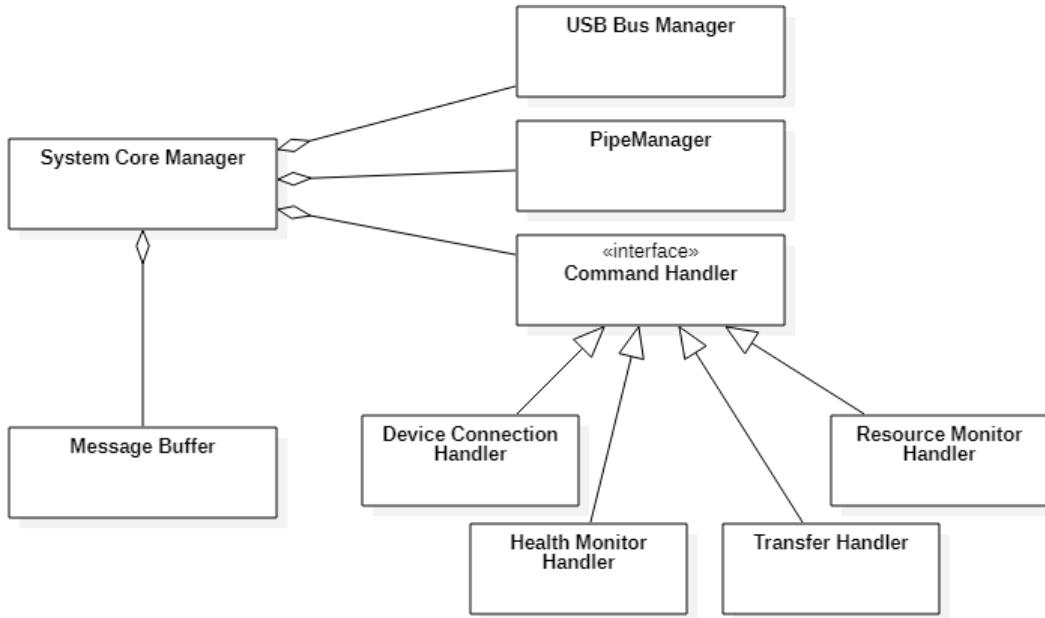


그림 37 System Core Manager 모듈 구성

**Pipe Manager**와 **USB BUS Manager**는 Pipe 정보와 USB Bus Topology 정보를 각각 관리합니다. Pipe 정보를 관리하기 위한 자료 구조는 검색 성능 향상을 위해 **Array**를 이용한 **Cache**와 전체 **Mapping 정보 List**를 조합하여 관리합니다. BUS Topology 정보는 Tree 형태로 관리합니다. 또한, 향후 해당 정보의 자료구조가 변경될 수 있는 상황을 고려하여 자료 구조는 별도의 Interface로 분리하여 관리합니다.

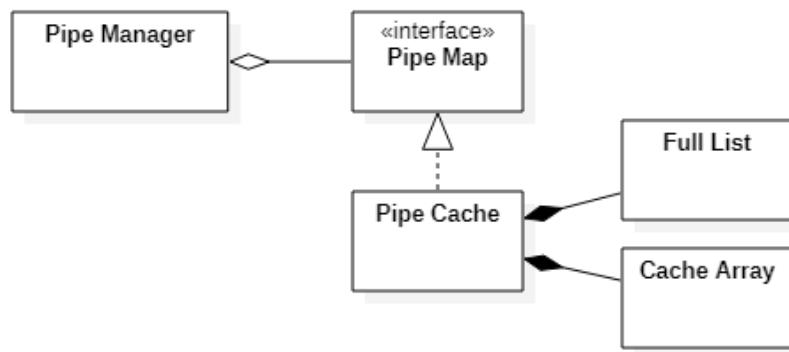


그림 38 Pipe Manager 모듈 구조

**Event Manager** 시스템 외부에서 발생한 이벤트를 처리하는 기능을 구현합니다. 디바이스 연결 이벤트와 같이 외부 모듈에서 이벤트 처리가 필요한 경우 해당 이벤트를 처리할 수 있는 외부 모듈로 Event 발생을 통보합니다. 이를 위해 이벤트 별로 발생 내용을 통보할 대상을 별도로 관리하여야 한다.

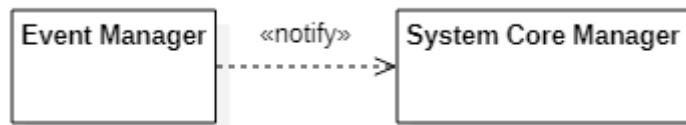


그림 39 Event Manager 모듈 구조

#### 4.2.3. Resource Monitor

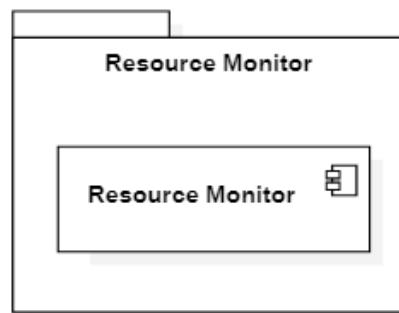


그림 40 Resource Monitor 패키지 구성

**Resource Monitor**는 주기적으로 각 디바이스의 유휴 자원을 모니터링 하고, 발견된 유휴 자원을 System Core로 리포트 하는 기능을 구현한다. 이를 위해 각 디바이스에 할당되는 자원과 실제 사용되고 있는 자원 상황을 주기적으로 수집하고, 할당된 자원이 일정 시간 이상(임계치 설정 필요) 사용되지 않고 있는 경우 이를 유휴 자원으로 판단한다. 주어진 정보를 이용하여 유휴 자원을 판단하는 알고리즘은 향후 시스템 적용 환경에 따라 변경이 용이하도록 별도의 Interface로 분리하여 구현한다.

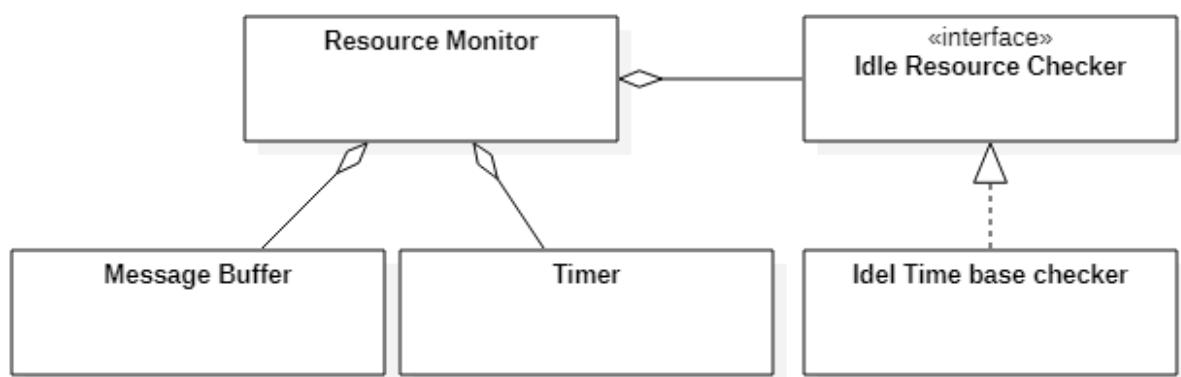


그림 41 Resource Monitor 모듈 구조

#### 4.2.4. Health Monitor

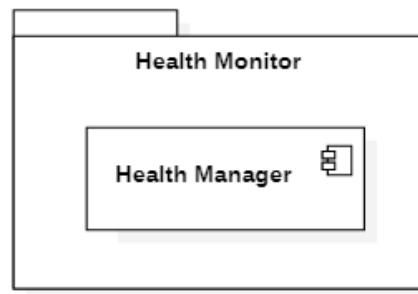


그림 42 Health Monitor 패키지 구성

**Health Monitor**는 각 Thread의 이상 여부를 주기적으로 모니터링 하는 기능을 구현한다. 이를 위해 주기적으로 각 Thread로 메시지를 보내 응답을 확인하고, 응답 시간이 임계치를 초과하는 경우 비정상 동작으로 판단하여 해당 Thread에 대한 복구를 System Core 모듈에 요청한다.

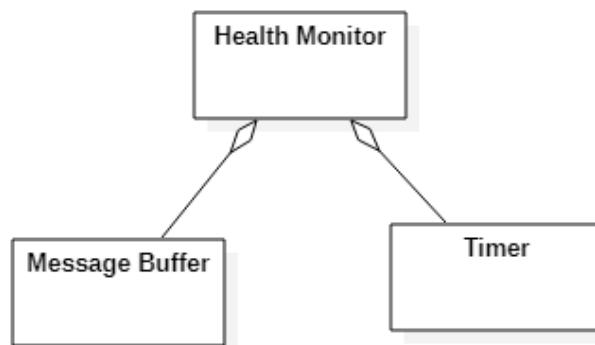


그림 43 Health Monitor 모듈 구조

#### 4.2.5. Common Service

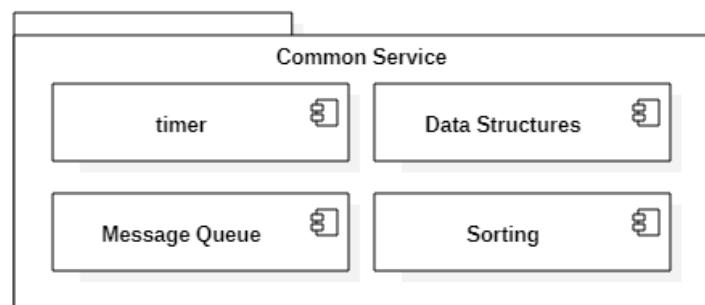


그림 44 Common Service 패키지 구성

**Timer**는 Timeout 발생시 Alarm을 발생시키는 기능과 현재 시간을 확인할 수 있는 기능을 구현한다. 시스템 내부적으로 Resource Monitor, Health Monitor와 같은 주기적 동작이 필요한 모듈에서 사용 한다.

**Message Queue**는 입력된 데이터를 Buffer에 저장하는 기능을 구현한다. 저장된 메시지는 저장 방식에 따라 FIFO 혹은 Priority Queue 등과 같이 우선 순위에 따라 데이터를 저장하고 사용할 수 있는 기능을 제공한다. 각 Thread별로 외부에서 전달되는 Message를 관리하기 위한 Message Buffer 기능 구현을 위해 사용한다.

**Sorting**은 입력된 데이터를 정렬하기 위한 일반적인 Sorting 알고리즘을 제공한다. 시스템 내부에서 Transfer의 우선 순위를 정하거나, Thread 간이 Load의 우선 순위를 정하는 모듈 등에서 사용한다.

**Data Structures**는 Array, Linked List와 같은 일반적인 자료 구조를 제공한다.

#### 4.2.6. Device Control

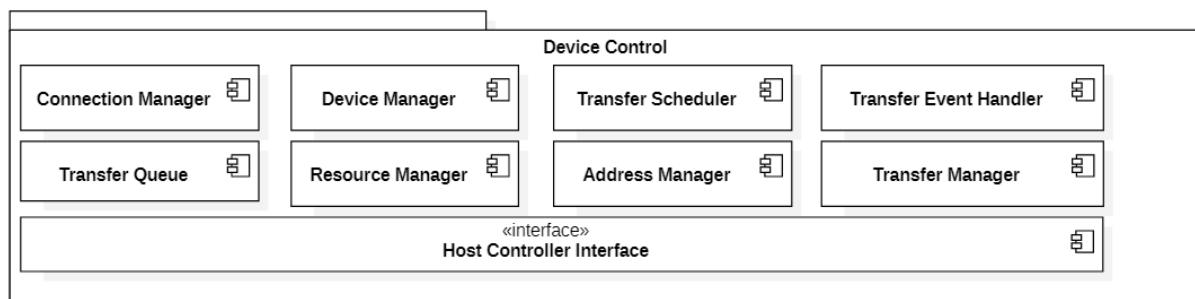


그림 45 Device Control 패키지 구성

**Connection Manager**는 Message Buffer를 통해 System Core로부터 전달된 디바이스 연결/해제 요청을 관리하는 기능을 구현한다. 이를 위해 모듈 내에 USB 디바이스 Emulation 과정을 관리하는 Logic를 포함하며, 디바이스 연결을 위해 필요한 디바이스 정보, 리소스 정보, 디바이스 Address 정보 등을 별도의 관리 모듈을 통해 제공받는다.

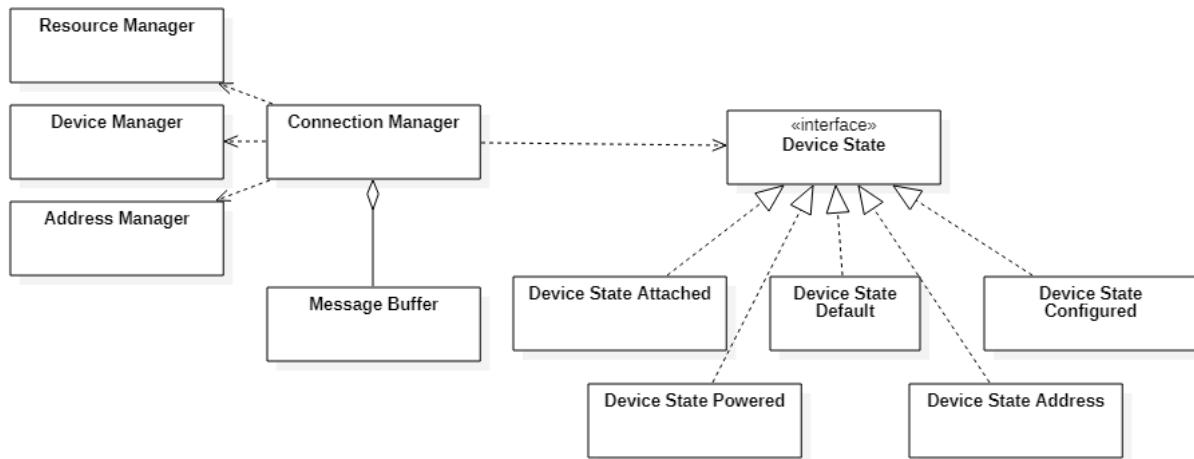


그림 46 Connection Manager 모듈 구조

**Device Manager**는 디바이스의 정보를 통합하여 관리하는 기능을 구현한다. 시스템 내부적으로 디바이스와 관련된 Address, Description, Configuration, 디바이스 상태 정보 등을 관리하여야 한다. 특히 Configuration 정보의 경우 디바이스별로 여러 Configuration을 포함할 수 있고 Configuration 내부에는 Interface, endpoint 정보 등이 계층적으로 포함되어 있어 Tree 형태의 자료 구조를 이용하여 관리한다. 또한, 디바이스 정보는 여러 Thread에서 동시에 접근 가능 하도록 동기화 처리를 포함하여 구현한다.

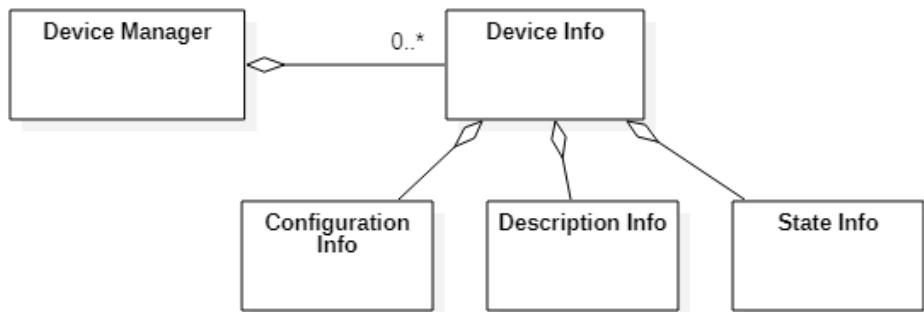


그림 47 Device Manager 모듈 구조

**Resource Manager**는 각 디바이스별 자원 정보를 통합하여 관리한다. 자원 정보에는 디바이스별 Bandwidth, Power 할당 정보 등이 포함된다. 리소스 정보는 여러 Thread에서 동시에 접근 가능 하도록 동기화 처리를 포함하여 구현한다.



그림 48 Resource Manager 모듈 구조

**Address Manager**는 각 디바이스에 할당되는 Address 정보를 통합하여 관리한다. 내부적으로 “가용 Address”와 “할당 Address” 목록을 이용하여 Address 할당을 관리한다. Address 할당 요청이 있는 경우 가용 Address 목록에서 Address를 할당하고, 할당된 Address는 “할당 Address” 목록에 포함하도록 구현한다.



그림 49 Address Manager 모듈구조

**Transfer Scheduler**와 **Transfer Event Handler**는 Message Buffer를 통해 전달되는 데이터 전송 요청을 처리한다. Transfer Scheduler는 Message Buffer에 있는 전송 요청을 순차적으로 꺼내 정해진 우선 순위에 따라 Transfer Queue에 저장한다. 각 Transfer Manager의 Load를 계산하여 가장 Load가 적은 Thread를 선택하고, 선택된 Thread에 Transfer Queue에서 가장 우선 순위가 높은 요청을 전달 한다. Transfer Event Handler는 전송 완료 Event를 수신한 경우, Transfer를 처리중인 Thread를 확인하여 해당 Thread에 전송 완료를 통보한다. Transfer 요청에 대한 Scheduling 알고리즘은 상황에 따라 변경이 용이하도록 구현한다.

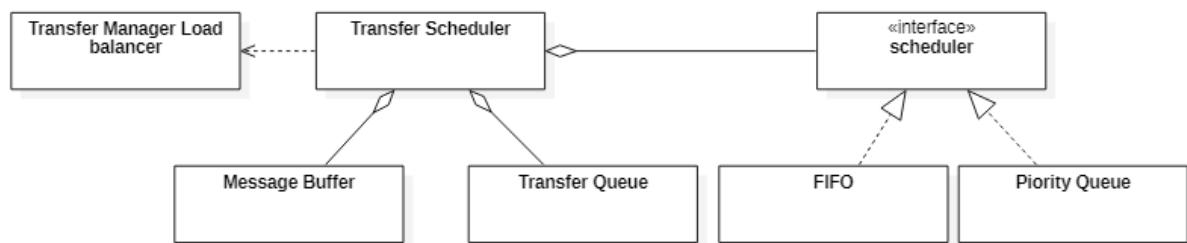


그림 50 Transfer Scheduler 모듈 구조

**Transfer Manager**는 Message Buffer에 저장된 전송요청을 순차적으로 확인하여 Host Controller Interface를 통해 전송 요청을 처리할 수 있는 Controller Module로 전달하는 기능을 구현한다.

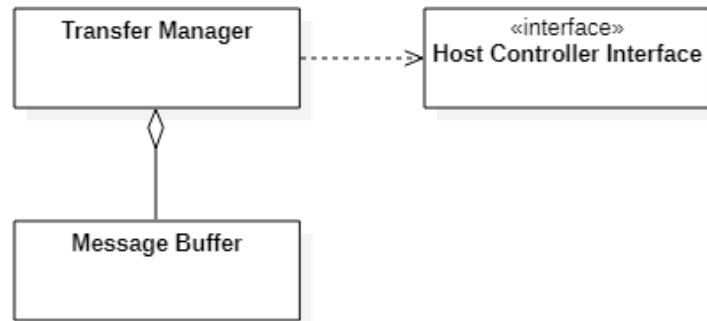


그림 51 Transfer Manager 모듈 구조

**Host Controller Interface**는 Host Core Driver가 사용하는 Host Controller HW 기능에 대한 추상화된 Interface를 정의한다. Host Controller Driver의 상위 Layer인 Host Core Driver는 해당 Interface를 통해서만 HW 기능을 사용함으로써 HW 변경에 대한 영향을 최소화할 수 있다. Host Controller Interface에 대한 Realization은 하위 Layer인 Host Controller Driver에서 구현한다.

#### 4.2.7. Core Driver OS Interface

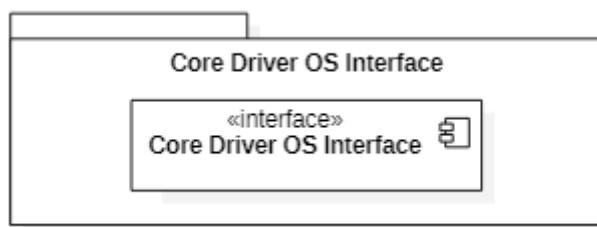


그림 52 OS Interface 패키지 구성

**Core Driver OS Interface**는 Host Core Driver가 사용하는 OS 기능에 대한 추상화된 Interface를 정의한다. OS의 기능들에 대한 Primitive Interface를 Host Core Driver가 사용하는 형태로 추상화 함으로써 OS 변경에 따른 영향을 최소화 할 수 있다. Core Driver OS Interface에 대한 Realization은 OS Abstraction 패키지에서 구현한다.

#### 4.2.8. Host Controller Driver

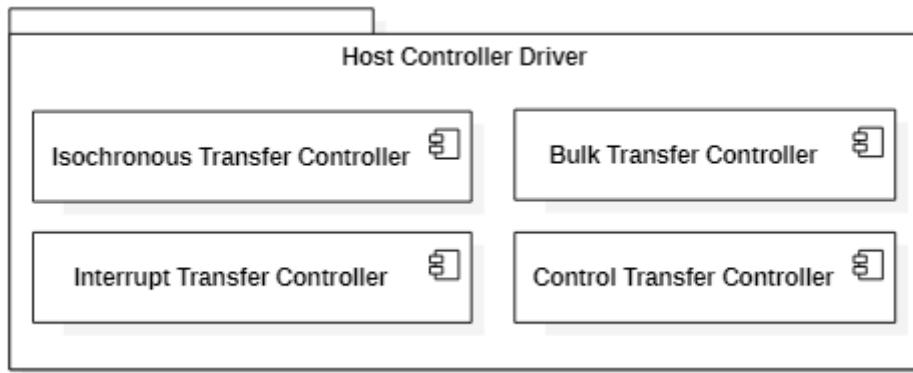


그림 53 Host Controller Driver 패키지 구성

**Isochronous Transfer Controller, Interrupt Transfer Controller, Bulk Transfer Controller, Control Transfer Controller**는 각 전송 Type별 데이터 전송을 위한 Host Controller(HW) 제어 기능을 제공한다. 각 모듈은 입력된 Data size가 Isochronous 전송의 최대 transaction 크기보다 큰 경우 데이터를 분할하고 관리하는 기능 등을 공통으로 포함하고 있다. 반면 이를 USB Protocol에 맞게 transaction 단위로 구성하는 것은 각 모듈이 개별적으로 구현하며 생성된 transaction을 Host Controller HW와 공유하는 Shared Memory를 통해 HW에 전달하는 방식도 각 모듈별로 구현이 필요하다. 또한, Transaction을 생성하는 방법과 Shared Memory를 관리하는 방법은 Host Controller(HW)에 따라 달라 지기 때문에 Host Controller에 의존 적인 부분은 별도의 모듈로 분리하여 구현한다.

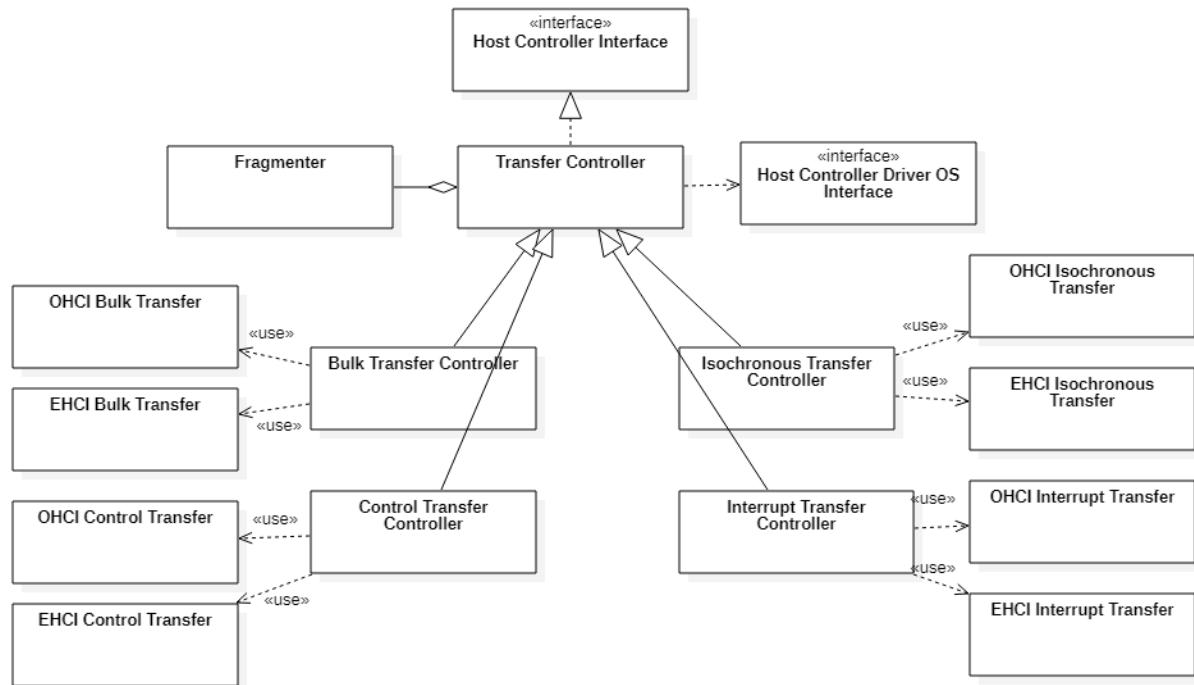


그림 54 Transfer Controller 모듈 구조

#### 4.2.9. Host Controller Driver OS Interface

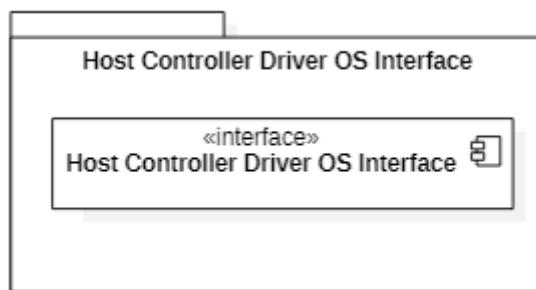


그림 55 Host Controller Driver OS Interface 패키지 구성

**Host Controller Driver OS Interface**는 Host Controller Driver가 사용하는 OS 기능에 대한 추상화된 Interface를 정의한다. Core Driver와의 의존성을 분리하기 위하여 별도의 분리된 모듈로 정의하며, USB Host Controller Driver Layer에 별도로 포함한다. 해당 Interface에 대한 Realization은 OS Abstraction 패키지에서 구현한다.

#### 4.2.10. OS Abstraction

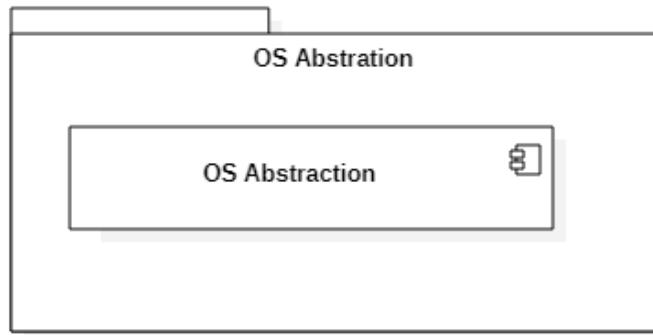


그림 56 OS Abstraction 패키지 구성

**Android Abstraction**은 Android OS에서 제공하는 일반적인 OS 기능들을 Host Controller Driver가 정한 Interface 형태로 추상화한 기능을 제공한다. 시스템은 각 Layer간의 영향을 최소화하기 위하여 시스템 내부의 각 Layer별로 각각의 독립적인 OS 추상화 interface Module를 정의하고 있다. 따라서, OS 추상화 기능을 구현하는 Module은 두개 Layer의 Interface를 다중 상속하여 구현한다.

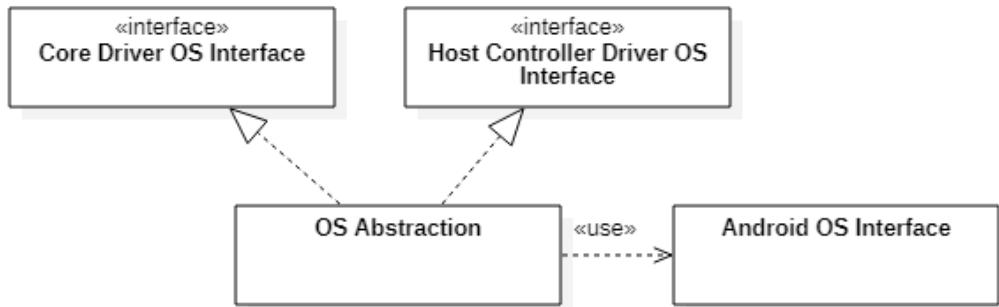


그림 57 OS Abstraction 모듈 구조

#### 4.3. Work Assignment

시스템 개발 그룹은 크게 시스템의 Framework 역할을 하는 기능들을 개발하는 **System Core 개발 파트**, Host Controller를 제어하며 외부 디바이스와 통신하는 기능을 개발하는 **Device Control 개발 파트**, 그리고 전체 시스템을 제어하는 부가적인 기능을 개발하는 **Monitoring 기능 개발 파트**로 구성된다. 본 시스템의 개발 그룹 구성과 세부 개발 모듈은 다음 그림과 같다.

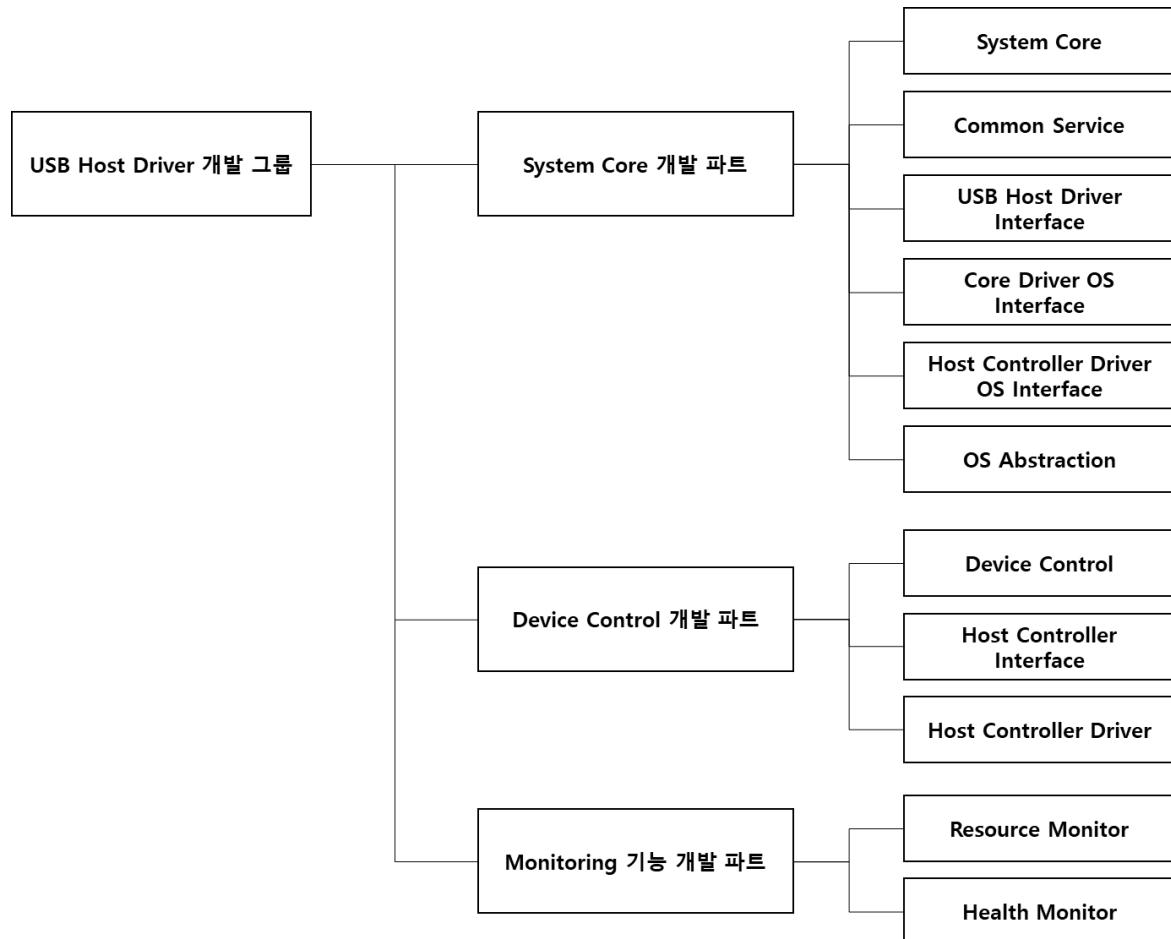


그림 58 시스템 전체 Work Assignment View

# 부록

<b>A. 도메인 모델.....</b>	<b>72</b>
A1. UC01_디바이스 연결 .....	73
A2. UC02_디바이스 분리 .....	73
A3. UC03_데이터 Read.....	74
A4. UC04_데이터 Write.....	75
A5. UC05_디바이스 관리 .....	75
A6. UC06_USB 버스 관리 .....	76
<b>B. 품질 시나리오 .....</b>	<b>77</b>
<b>C. 품질 시나리오 분석 .....</b>	<b>79</b>
C1. 시스템에 대한 비즈니스 드라이버 분석 .....	79
C2. 품질 시나리오 순위 선정 및 근거 .....	79
C2.1. QS별 추가 분석 내용 .....	82
<b>D. 후보 구조.....</b>	<b>85</b>
D1. NFR_01 디바이스 연결 성공 비율 .....	85
D1.1. CA_1. 자원 관리자를 통한 시스템 자원 관리 통합 .....	87
D1.2. CA_2. 낭비되는 자원에 대한 garbage correction .....	88
D1.3. CA_3. 자원 할당 재시도 .....	88
D1.4. CA_4. 디바이스 매니저를 통한 디바이스 정보 통합 관리 .....	88



D1.5. CA_5. Control 정보를 위한 Bandwidth Reservation .....	89
D1.6. CA_6. Emulation 과정을 재시도.....	89
D1.7. CA_7. 메모리 복사 기반의 전송 데이터 전달 .....	90
D1.8. CA_8. Ping/Echo 기반 시스템 health check.....	90
D1.9. CA_9. Heart bit 기반 시스템 health check .....	91
D2. NFR_02 디바이스로부터 입력에 대한 처리 시간 .....	91
D2.1. CA_10. Array 기반의 pipe 매핑 정보 관리 .....	93
D2.2. CA_11. List 기반의 pipe 매핑 정보 관리 .....	94
D2.3. CA_12. Cache 기반의 pipe 매핑 정보 관리 .....	94
D2.4. CA_13. Periodic Transaction 처리를 위한 전용 Thread 운영 구조 .....	95
D2.5. CA_14. Interrupt 방식의 시스템↔Application간 이벤트 전달 구조.....	96
D2.6. CA_15. Polling 방식의 시스템↔Application간 이벤트 전달 구조.....	96
D2.7. CA_16. 디바이스 입력에 대한 Pre-fetch 구조.....	97
D3. QA_01 데이터 전송 속도 .....	97
D3.1. CA_17. 데이터 Transfer와 Transaction을 단일 Thread에서 운영 하는 구조 .....	99
D3.2. CA_18. 데이터 Transfer와 Transaction을 개별 Thread에서 운영 하는 구조 .....	100
D3.3. CA_19. Transfer 요청에 대한 Thread 생성(Creation) 구조 .....	100
D3.4. CA_20. Transfer 요청 속성별 Thread 할당 구조 .....	101
D3.5. CA_21. Transfer 요청에 대한 Thread 임의 할당(Random Dispatch) 구조.....	102
D3.6. CA_22. Round Robin 방식의 Transfer 할당 스케줄링 .....	102
D3.7. CA_23. Priority Queue 방식의 Transfer 할당 스케줄링 .....	103

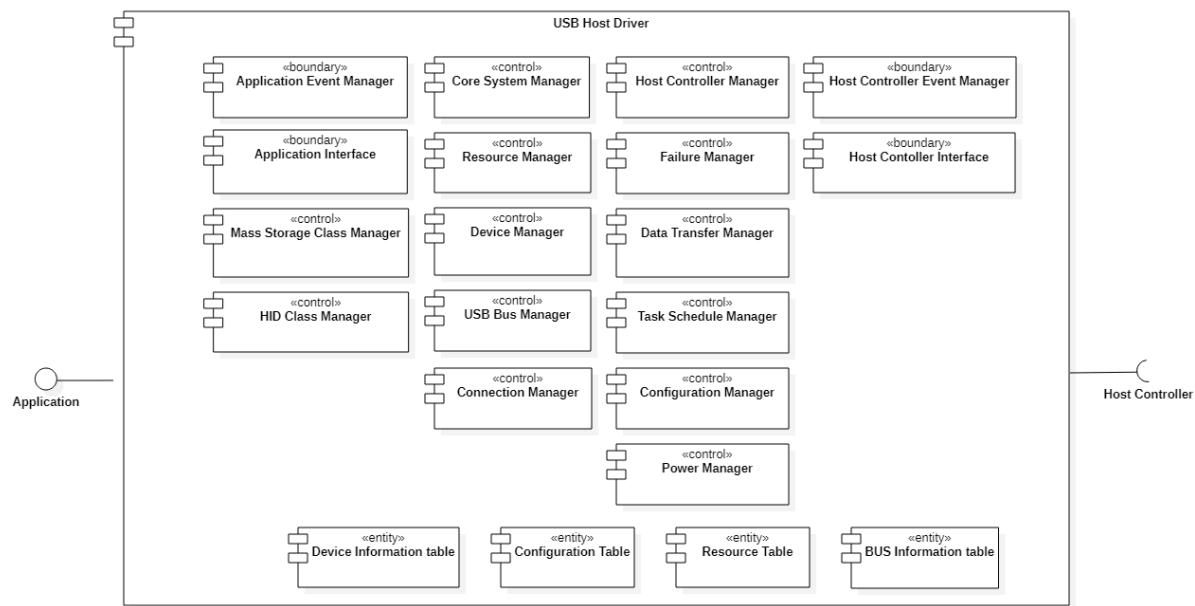
D3.8. CA_24. On-demand 방식의 Transaction 생성 .....	103
D3.9. CA_25. Pre-generation 방식의 Transaction 생성 .....	104
D3.10. CA_26. Buffered 방식의 Transaction 생성 구조 .....	105
D3.11. CA_27. 단일 List 기반 Transfer 목록 관리 구조 .....	106
D3.12. CA_28. Transfer type별 Transfer목록 관리 구조.....	107
D3.13. CA_29. FIFO 기반 Transfer scheduling.....	107
D3.14. CA_30. Priority 기반 Transfer scheduling .....	108
D3.15. CA_31. 단일 Message Queue 기반 컴포넌트간 메시지 전달 구조.....	108
D3.16. CA_32. 컴포넌트별 Message Queue 관리 구조 .....	109
D3.17. CA_33. 메모리 참조 기반의 전송 데이터 전달 구조.....	110
D3.18. CA_34. Tree 기반의 Configuration 정보 관리 구조 .....	111
D3.19. CA_35. Tree 기반의 BUS topology 정보 관리 구조 .....	111
D3.20. CA_36. 데이터 Cache 관리 구조 .....	112
D4. QA_02 디바이스의 연결/해제시 인식 지연 시간.....	113
D4.1. CA_37. List 기반 디바이스 목록 관리 .....	115
D4.2. CA_38. Array 기반 디바이스 목록 관리 .....	115
D4.3. CA_39. 디바이스 연결 처리를 위한 독립 Thread 구조 .....	115
D4.4. CA_40. 디바이스 연결 요청 별 독립된 Thread 구조 .....	116
D4.5. CA_41. 연결 관리 객체의 Pre-Generation .....	117
D4.6. CA_42. 연결 관리 객체의 Lazy Clear .....	117
D4.7. CA_43. 가용 Address 목록 관리 구조(Heap).....	118

D4.8. CA_44. 가용 Address 목록 관리 구조(Array).....	118
D5. QA_03 동시에 연결할 수 있는 디바이스의 수.....	119
D5.1. CA_45. 디바이스 Reconfiguration을 통한 자원 재 분배 구조 .....	121
D5.2. CA_46. Delayed Resource Allocation 구조 .....	122
D5.3. CA_47. 디바이스 정보 저장 자료 구조의 크기 .....	122
D6. QA_04 새로운 디바이스 클래스 추가 용이성 .....	123
D6.1. CA_48. USB Host Driver Interface 모듈 분리 .....	123
D6.2. CA_49. 디바이스 클래스별 Interface 적용 .....	124
D7. QA_05 새로운 실행 환경(HW, OS) 지원 용이성 .....	125
D7.1. CA_50. Layer별 OS 기능 추상화 Interface 구조 .....	126
D7.2. CA_51. Host Controller 기능 사용 모듈 통합 .....	126
D7.3. CA_52. Transfer type별 Interface 모듈 분리 구조 .....	127
D7.4. CA_53. 전송 특징(Async와 Periodic)에 따라 Interface 모듈 분리 구조 .....	128
<b>E. 후보 구조 평가 .....</b>	<b>130</b>
E1. NFR_01 디바이스 연결 성공 비율 .....	130
E2. NFR_02 디바이스로부터 입력에 대한 처리 시간 .....	134
E3. QA_01 데이터 전송 속도 .....	136
E4. QA_02 디바이스의 연결/해제시 인식 지연 시간 .....	143
E5. QA_03 동시에 연결할 수 있는 디바이스의 수 .....	145
E6. QA_04 새로운 디바이스 클래스 추가 용이성 .....	146
E7. QA_05 새로운 실행 환경(HW, OS) 지원 용이성 .....	147

E8. 후보 구조 평가 결과 .....	148
<b>F. 최종 구조 설계.....</b>	<b>151</b>
F1. 동작 측면의 최종 구조 .....	151
F1.1. System Core Thread 구성과 관련된 후보 구조.....	153
F1.2. Device Connection Manager Thread 구성과 관련된 후보 구조.....	153
F1.3. Transfer Dispatcher Thread 구성과 관련된 후보 구조.....	154
F1.4. Transfer Manager Thread 구성과 관련된 후보 구조 .....	154
F1.5. Health Monitor Thread 구성과 관련된 후보 구조.....	154
F1.6. Resource Monitor thread 구성과 관련된 후보 구조 .....	155
F1.7. 시스템 메시지 교환과 관련된 후보 구조 .....	155
F2. 개발 측면의 최종 구조 .....	155
F3. 최종 구조의 단점/Risk 분석.....	157
F3.1. 전송 타입에 따른 고정 우선 순위 방식 .....	157
F3.2. 공유 자원 접근 성능 .....	157
F3.3. 메모리 참조 방식의 데이터 전달 .....	158

## A. 도메인 모델

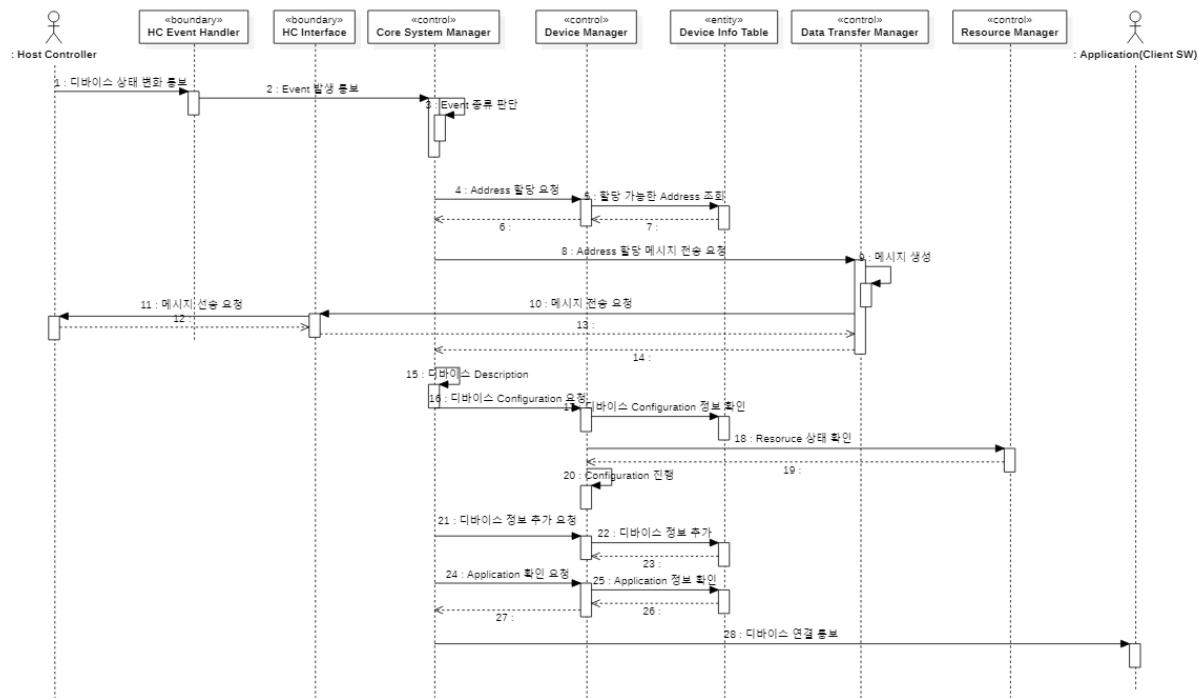
본 과제에서 제안하는 USB Host Driver의 도메인 모델은 다음 그림과 같다.



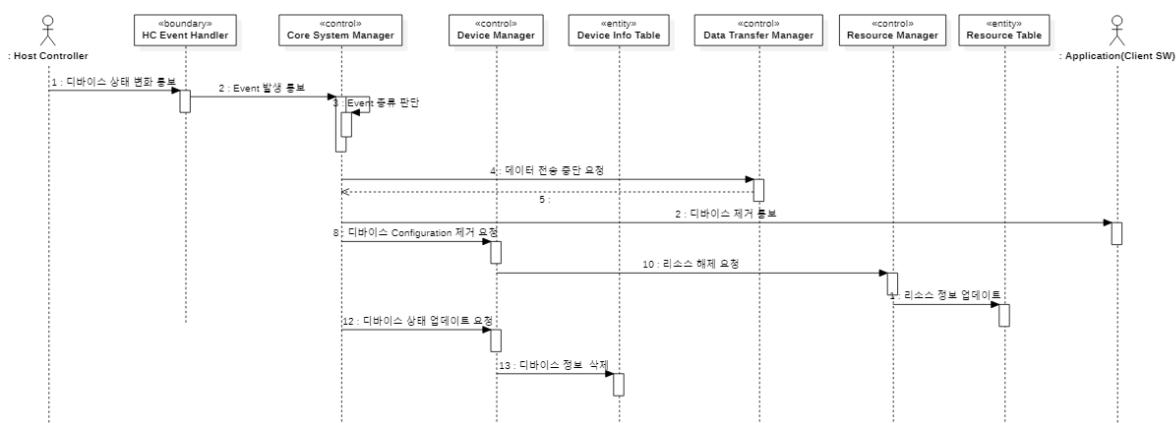
도메인 모델에 포함된 각 컴포넌트는 Use Case를 바탕으로 boundary/control/entity를 명시하였다. 시스템의 외부 Actor인 Application은 Application Interface와 Application Event Manager를 통해 시스템을 사용한다. USB Host Driver는 Host Controller Interface를 이용하여 Host Controller를 제어하며, Host Controller Event Manager를 이용하여 Host Controller가 전달하는 비동기적 Event를 처리한다. Application의 요청에 의해 디바이스를 관리하거나 디바이스와 통신하는 등의 USB Host Driver의 기능은 역할에 따라 각각의 Control Component로 구성하였고, 시스템에 필요한 주요 정보들은 각각의 entity로 명시하였다.

다음은 도출된 Component를 기반으로 작성된 시스템의 각 사용 시나리오에 대한 Use Case diagram이다.

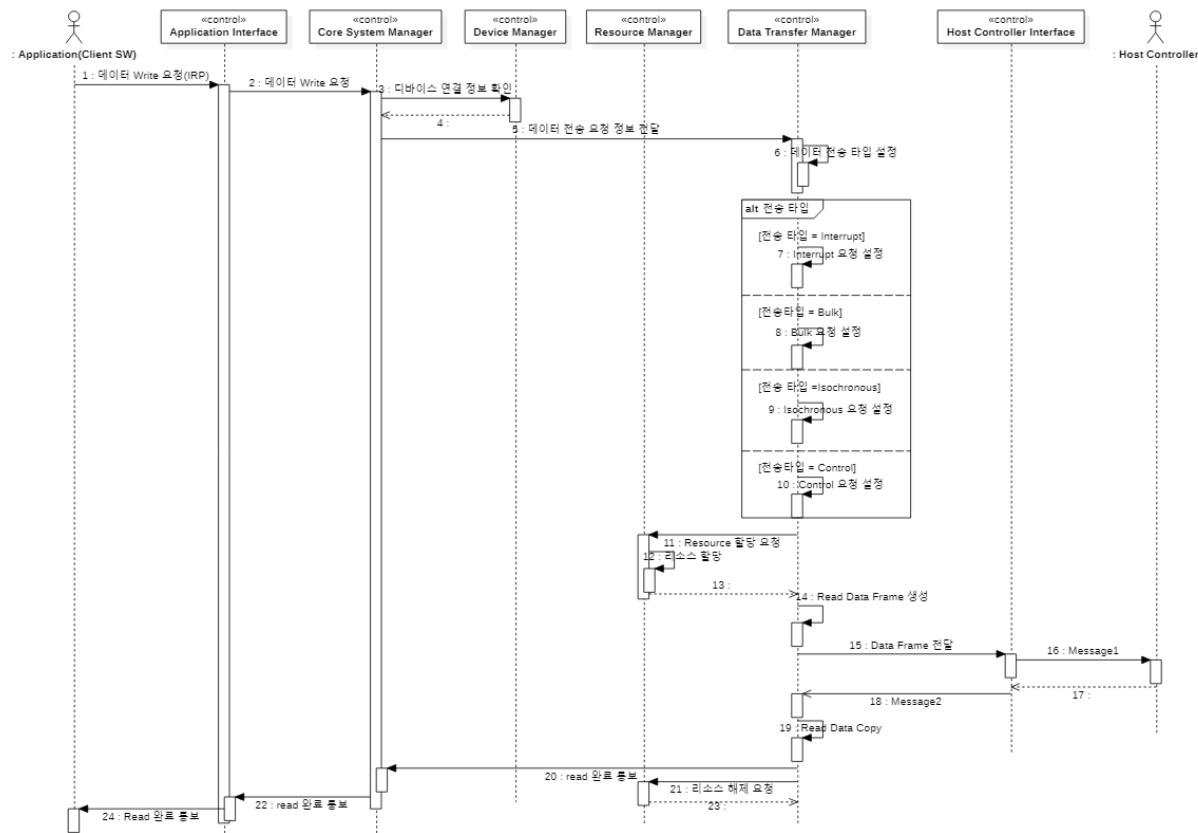
## A1. UC01\_디바이스 연결



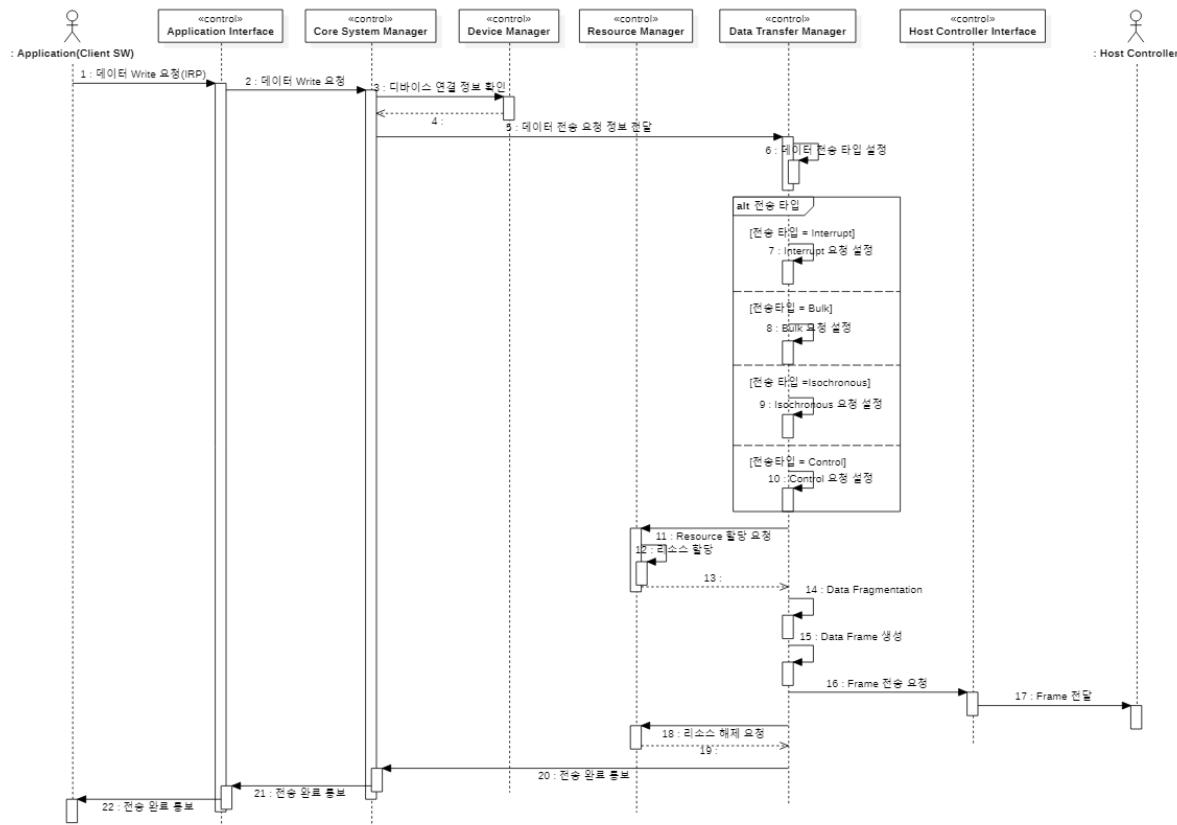
## A2. UC02\_디바이스 분리



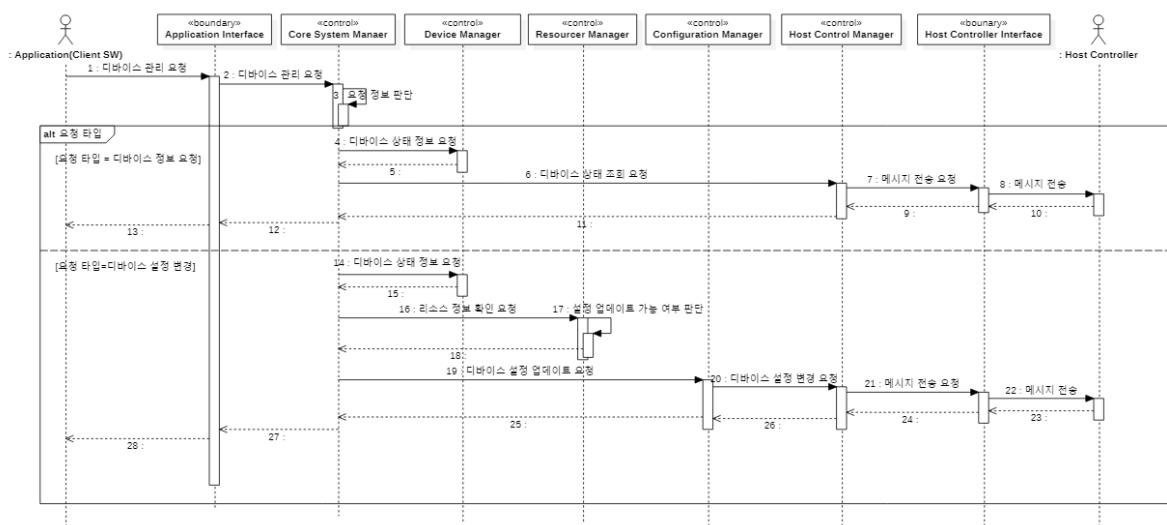
### A3. UC03\_데이터 Read



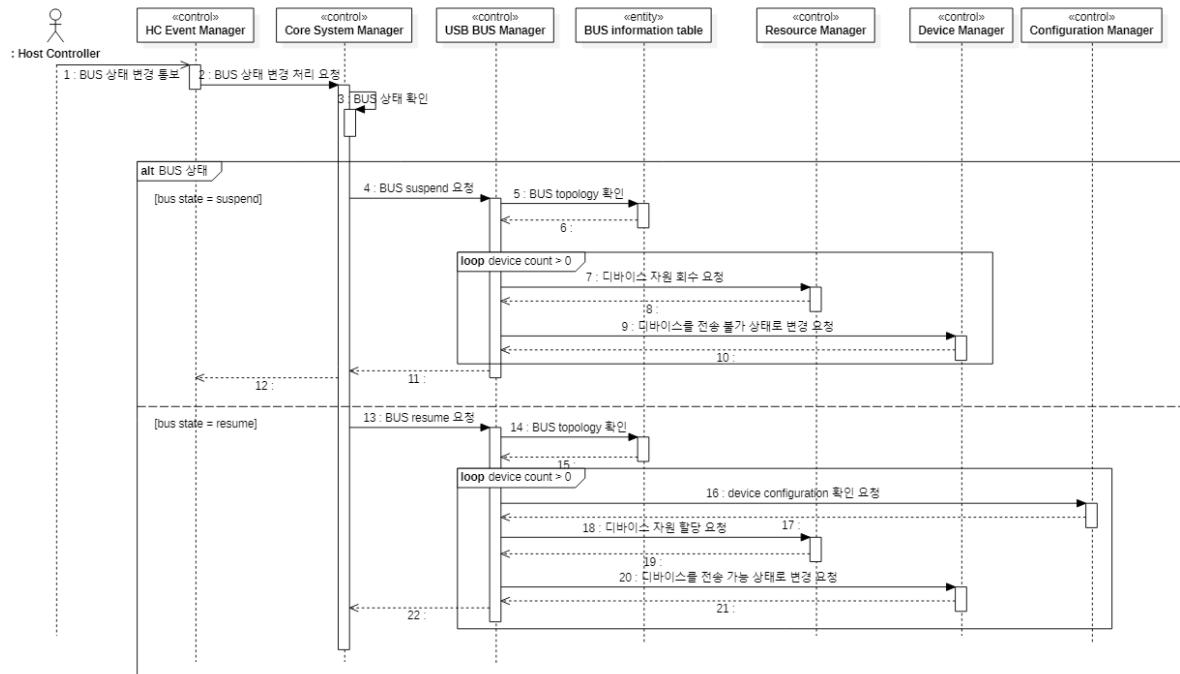
## A4. UC04\_데이터 Write



## A5. UC05\_디바이스 관리



## A6. UC06\_USB 버스 관리



## B. 품질 시나리오

QS_01	가용성	디바이스 연결 성공 비율
설명	호스트에 USB 디바이스가 연결되는 경우 디바이스가 정상적으로 인식되어 서비스 될 확률은 높을수록 좋다. 디바이스 연결은 호스트의 제품의 신뢰성에 영향을 주는 Critical한 요인으로 <b>지정된 수준 이상을 만족하여야 한다.</b>	
측정	$[연결 성공률] = [연결 성공 횟수] / [연결 시도 횟수]$	
QS_02	성능	디바이스로부터 입력에 대한 처리 시간
설명	호스트에 연결되는 HID의 경우 사용자와 Interaction 하는 장치로 반응시간이 매우 중요하다. 또한 특정 시간이상 지연이 발생하는 경우 오류로 인식될 수 있기 때문에 <b>지정된 시간 기준 이상의 지연이 발생하지 않도록 해야 한다.</b>	
측정	$[입력 처리시간] = [시스템이 디바이스 입력을 Application에 통보한 시간] - [Host Controller가 READ Transaction 완료를 시스템에 통보한 시간]$	
QS_03	성능	데이터 전송 속도
설명	USB의 Data 전송 속도에 대한 Spec에서의 제약은 없지만, 전송 속도는 사용자 경험 측면에서 매우 중요하기 때문에 데이터 전송(Read/Write) 속도는 빠를수록 좋다.	
측정	$[전송 속도] = [Application이 요청한 전송 데이터 크기] / ([시스템이 Application에게 전송 완료를 통보한 시간] - [Application이 시스템에 데이터 전송을 요청한 시간])$	
QS_04	가용성	오류에 대한 영향 범위
설명	시스템은 동시에 여러 개의 디바이스를 지원하기 때문에 특정 디바이스에서 오류가 발생하는 경우, 해당 오류가 전체 USB System에 영향을 주지 않도록 오류의 영향 범위를 제한해야 한다. 따라서 <b>오류의 영향 범위는 작을수록 좋다.</b>	
측정	오류 발생범위 = 하나의 디바이스에서 오류 발생시 영향받는 전체 디바이스의 수	
QS_05	성능	디바이스의 연결/해제시 인식 지연 시간
설명	호스트에 디바이스가 연결되거나 제거되는 경우 디바이스 연결 상태의 변경을 인식하는 속도는 USB Spec에 정의된 제한은 없지만 <b>사용자 경험 측면에서 빠를수록 좋다.</b>	
측정	$[디바이스 인식 지연 시간] = [시스템이 Application으로 장치 연결을 통보한 시간] - [시스템에 새로운 디바이스 연결 정보가 전달된 시간]$	

QS_06	성능	동시에 연결할 수 있는 디바이스의 수
설명	하나의 호스트에는 다수의 디바이스가 연결될 수 있으며 HUB를 통해 연결을 계속 해서 확장할 수 있기 때문에, 시스템은 다수의 디바이스 연결을 동시에 처리할 수록 좋다.	
측정	[동시에 연결 가능한 디바이스의 수]	
QS_07	변경 용이성	새로운 디바이스 클래스 추가 용이성
설명	USB 디바이스는 계속해서 새로운 장치가 개발되고 새로운 클래스가 만들어지고 있기 때문에 새로운 디바이스 클래스 추가에 대한 개발 용이성 확보가 필요하다. 시스템 개발자 측면에서는 신규 클래스 지원 기능 개발이 용이 할수록 좋다.	
측정	[추가 용이성] = [변경에 필요한 개발 비용(M/M)]	
QS_08	변경 용이성	새로운 실행 환경(HW, OS) 지원 용이성
설명	시스템이 동작 하는 실행환경이 변경되는 경우 신규 실행 환경 지원을 위한 개발 비용은 작을수록 좋다.	
측정	[추가 용이성] = [변경에 필요한 개발 비용(M/M)]	
QS_09	가용성	데이터 전송 오류 복구 비율
설명	시스템이 USB 디바이스로 데이터 전송 중 예기치 못한 오류가 발생하는 경우 이를 복구 할 수 있는 비율이 높을수록 좋다.	
측정	[오류 복구 비율] = [복구된 횟수]/[데이터 전송 중 오류 발생 횟수]	
QS_10	보안성	데이터 전송 과정의 보안성
설명	호스트와 디바이스 사이의 비밀성이 보장되어야 하는 데이터(ex, DRM Contents, 기밀문서 등)을 전송하는 경우 USB를 통해 전송되는 구간에서도 비밀성이 보장되어야 하며, 보안성 레벨이 높을수록 좋다.	
측정	[보안성 레벨] = [전송 구간에 적용된 암호화 알고리즘의 보안성 레벨]	
QS_11	성능	시스템이 사용하는 리소스 사용량
설명	시스템이 동작하는데 필요한 호스트의 자원(메모리/CPU)의 사용량은 낮을수록 좋다.	
측정	$[평균 자원(메모리/CPU) 사용량] = [시스템 동작 시간동안 소모한 자원량의 합]/[시스템 동작 시간]$ $[최대 자원(메모리/CPU) 사용량] = \text{MAX}[\text{특정 시점에 소모한 자원량}]$	

## C. 품질 시나리오 분석

### C1. 시스템에 대한 비즈니스 드라이버 분석

시스템의 품질 시나리오의 중요도 선정에는 다음과 같은 주요 Stakeholder의 요구 사항을 고려하였다.

1. **컴퓨팅 제품(호스트) 사용자:** 사용자는 USB 디바이스를 호스트에 연결하여 주변 장치를 사용하는 사용자로, 시스템의 품질 속정에 직접적인 영향을 받기 때문에 사용자의 요구 사항은 중요도가 높다.
  - USB 디바이스 사용시 디바이스 인식/전송 속도 등 성능이 높아야 한다.
  - USB 디바이스 사용도중 오류가 발생하지 않아야 한다.
  - 여러 개의 USB 디바이스를 호스트에 연결하여 동시에 사용할 수 있어야 한다.
2. **시스템 개발자:** 시스템 개발자는 시스템의 초기 개발을 담당하고 실행환경 및 요구 사항의 변경에 따른 추가적인 개발을 담당한다. 따라서 시스템의 구조에 변경에 직접적인 영향을 받을 수 있기 때문에 시스템 개발자 요구 사항은 중요도가 높다.
  - 새로운 요구 사항이 추가되는 경우에 대한 개발이 용이해야 한다.
  - 새로운 실행환경에 적용이 용이해야 한다.
3. **시스템을 이용하는 Application 개발자:** Application 개발자는 시스템에 정의된 Interface를 이용하여 개발을 하기 때문에, Interface의 변경 등에 영향을 받는다. 하지만, Interface가 정의된 이후 변경사항이 빈번하지 않다면 Application 변경 또한 제한되기 때문에 Application 개발자 요구사항의 중요도는 상대적으로 낮다고 볼 수 있다.
  - Host Driver의 변경으로 인한 Application 추가 구현이 적어야 한다.

### C2. 품질 시나리오 순위 선정 및 근거

앞서 분석된 비즈니스 드라이버와 각 품질 시나리오에 대한 중요도 및 복잡도를 바탕으로 선정된 품질 시나리오의 순위는 다음과 같다.

구분	ID	품질 시나리오	중요도	복잡도	선정 결과
성능	QS_02	디바이스로부터 입력에 대한 처리 시간	H	H	NFR_02
	QS_03	데이터 전송 속도	H	H	QA_01
	QS_05	디바이스의 연결/해제시 인식 지연 시간	H	H	QA_02
	QS_06	동시에 연결할 수 있는 디바이스의 수	H	M	QA_03
	QS_11	시스템이 사용하는 리소스 사용량	M	H	
가용성	QS_01	디바이스 연결 성공 비율	H	M	NFR_01
	QS_04	오류에 대한 영향 범위	M	H	
	QS_09	데이터 전송 오류 복구 비율	M	H	
변경 용이성	QS_07	새로운 디바이스 클래스 추가 용이성	H	M	QA_04
	QS_08	새로운 실행 환경(HW, OS) 지원 용이성	H	M	QA_05
보안성	QS_10	데이터 전송 과정의 보안성	L	M	

각 품질 시나리오에 대해서는 아래와 같은 분석을 바탕으로 중요도와 복잡도를 선정하였고, 비즈니스 드라이버와의 연관성을 고려하여 우선 순위를 선정하였다.

### [QA\_01] 데이터 전송 속도

시스템의 비즈니스 드라이버 중 가장 높은 요구 사항은 “**USB 디바이스 사용시 디바이스 인식/전송 속도 등 성능이 높아야 한다**”는 것이기 때문에 데이터 전송 속도의 경우 품질 시나리오 중에서도 중요도가 높다. 또한, USB System은 가용한 자원이 정해져 있기 때문에 제한된 자원을 효과적으로 사용하여 데이터 전송 속도를 높이는 것은 고려할 사항들이 많고 구현에 대한 난이도 또한 높기 때문에 효율적으로 구현될 경우 **시스템의 자체적인 경쟁력도 높일 수 있으며, 사용자 경험 측면에서도 강점이 될 수 있다**. 부가적으로, 데이터의 전송 속도가 높을 경우 동일한 전송 요청을 짧은 시간에 완료할 수 있고, 이후 디바이스가 suspend 상태로 진입하여 USB bus가 idle이 되는 구간이 늘어나기 때문에 파워를 절감하는 것과 같이, 시스템의 다른 품질 속성도 간접적으로 향상될 수 있다. 따라서 데이터 전송 속도는 품질 속성 중 가장 중요한 **첫 번째 품질 속성**으로 판단하였다.

### [QA\_02] 디바이스의 연결/해제시 인식 지연 시간

디바이스 연결/해제시 인식 지연 시간은 비즈니스 드라이버의 “**USB 디바이스 사용시 디바이스 인식/전송 속도 등 성능이 높아야 한다**”라는 성능 요구사항과 직접 관련된 품질로 판단하여 두 번째 달성을 해야 할 품질 속성으로 판단하였다. 특히, 다수의 디바이스가 동시에 연결되어 동작하는 시스템 동작 환경을 고려할 때 디바이스의 연결을 빠르게 처리하기 위해서는 기술적으로 고려해야 할 부분들이 많고, 인식 시간을 줄일 수 있는 경우 사용자 경험 측면에서도 시스템의 장점으로 작용할 수 있기 때문에 두 번째 품질 속성으로 선정하였다.

#### [QA\_03] 동시에 연결할 수 있는 디바이스의 수

모바일 장치의 경우 다수의 USB 장치가 연결될 수 있고 다수의 장치가 연결된 상태에서도 안정적인 성능과 연결성을 확보하여야 하기 때문에 비즈니스 드라이버의 “**여러 개의 USB 디바이스를 호스트에 연결하여 동시에 사용할 수 있어야 한다**.”는 요구 사항과 관련하여 세 번째 높은 품질 속성으로 판단하였다. 특히, USB System은 제한된 자원(Bandwidth, Power등)을 가지고 있기 때문에 이를 효율적으로 분배하여 여러 디바이스를 동시에 지원하는 것은 기술적 구현 난의도가 높다. 따라서, 이를 효율적으로 관리하여 **동일한 자원 상황에서 타 시스템에 비해 다수의 디바이스를 지원하는 것은 시스템의 비교 우위가 될 수 있다고** 판단하여 세 번째 품질 속성으로 선정하였다.

#### [QA\_04] 새로운 디바이스 클래스 추가 용이성

시스템이 동작하는 모바일 환경은 USB를 이용한 새로운 디바이스들이 계속해서 개발되고 있기 때문에 시스템 개발 이후에도 신규 디바이스 클래스가 추가될 가능성이 매우 높다. 따라서 시스템을 새로운 디바이스 클래스로의 확장해야 할 가능성이 매우 높고, 빈도도 빈번하게 발생할 수 있을 것으로 판단하여 새로운 클래스 추가 용이성을 중요 품질 속성으로 판단하였다. 또한, OS나 HW의 변경에 비해서 지원해야 할 빈도가 높을 것으로 판단하여 더 높은 우선 순위로 고려하였다. 다만, 비즈니스 드라이버 측면에서 **새로운 디바이스 클래스 추가는 제품의 개발 비용과 관련된 것으로 사용자 요구 사항에 비해 낮은 순위로** 판단하여 품질 속성은 4번째로 평가하였다.

#### [QA\_05] 새로운 실행 환경(HW, OS) 지원 용이성

시스템이 동작하는 모바일 환경은 OS의 업데이트 주기가 짧고, 새로운 모바일 OS가 적용된 제품들의 개발 가능성도 높다. 따라서, 비즈니스 드라이버 측면에서 새로운 실행 환경을 지원하는 것은 개발 비용과 관련된 것으로 중요도가 높고, **모바일 환경의 경우 HW 및 OS의 변경 주기도 짧은 편으로 새로운 실행 환경을 빠르게 지원하는 것이 중요한 것으로** 판단하였다. 다만 디바이스 클래스 추가 등에 비해 변경 빈도는 낮을 것으로 판단하여 5번째 품질 속성으로 평가하였다.

### [NFR\_01] 디바이스 연결 성공 비율

디바이스 연결 성공은 USB 디바이스 사용을 위한 첫 번째 단계로 디바이스 인식에 대한 가용성/신뢰성을 반드시 만족해야 할 품질 속성으로 판단하다. 특히, “**USB 디바이스 사용시 디바이스 인식/전송 속도 등 성능이 높아야 한다**”는 비즈니스 드라이버와 직접 관련된 품질 속성으로 판단하여 **첫 번째 NFR**로 선정하였다.

### [NFR\_02] 디바이스로부터 입력에 대한 처리 시간

디바이스 입력 처리 시간을 줄이는 것은 사용자 경험 측면에서 매우 중요하기 때문에 주요한 품질 시나리오로 선정하였다. 특히, 디바이스 입력에 대한 처리 시간이 지연될 경우 사용자는 HID 장치의 오류로 판단할 수 있기 때문에 “**USB 디바이스 사용도중 오류가 발생하지 않아야 한다**”는 비즈니스 드라이버 측면에서 **반드시 지정된 시간 이내에 입력을 처리해야 할 것으로** 판단하여 **2번째 NFR**로 선정하였다.

#### C2.1. QS별 추가 분석 내용

**QS\_01. 디바이스 연결 성공 비율**은 사용성 측면에서 핵심적인 요인이기 때문에 중요도는 H로 선정하였다. 디바이스의 연결 과정은 USB Spec에 이미 정의되어 있기 때문에 구현 난의도가 높지 않은 것으로 판단하였고, 연결 과정에 발생할 수 있는 오류에 대한 처리 등이 필요한 것을 감안하여 복잡도는 M으로 선정하였다.

**QS\_02. 디바이스로부터 입력에 대한 처리 시간**은 키보드 마우스와 같은 HID 장치의 사용자 경험 측면에서 핵심적인 요인이기 때문에 중요도는 H로 선정하였다. 디바이스 입력을 빠르게 처리하기 위해서는 입력에 대한 Interrupt 처리, 자원의 재할당 등 구현에 참여하는 모듈이 다양하고 빠른 처리를 위한 스케줄링 알고리즘 등의 적용이 필요할 것으로 판단하여 구현 난의도는 H로 선정하였다.

**QS\_03. 데이터 전송 속도**는 사용자가 제품의 성능 측면에서 체감하는 중요 요인으로 판단하여 중요도는 H로 선정하였고, 데이터 전송 속도를 높이기 위해서는 자원의 할당, 작업 스케줄링등 다양한 요인에 대한 고려가 필요하기 때문에 복잡도는 H로 선정하였다.

**QS\_04. 오류에 대한 영향 범위**는 하나의 디바이스에서 발생한 오류가 다른 디바이스에 영향을 주지 않아야 하는 것으로, 사용자 경험 측면에서는 중요한 요인지만 데이터 전송 오류 발생 확률 등의 다른 QS를 만족할 경우 함께 달성될 확률이 높은 것으로 판단하여 중요도는 M으로 선정하였다. 오류

의 영향 범위를 제한하기 위해서는 다양한 오류 상황을 모니터링 하고 그에 대한 대응 기술을 구현해야 하기 때문에 구현 난의도는 H로 선정하였다.

**QS\_05. 디바이스의 연결/해제시 인식 지연 시간은** 사용자 경험에 영향을 주는 요인으로 연결/해제 시간이 지연될 경우 제품의 신뢰성에 영향을 줄 수 있다. 또한 디바이스 연결 시간이 지연되는 경우 사용자의 제품 사용 효율성을 떨어뜨릴 수 있기 때문에 중요도는 H로 선정하였다. 디바이스 연결 기능의 구현은 Spec에 있는 protocol을 참고할 수 있지만, 연결 속도를 높이기 위해서는 시스템 자원 할당 및 스케줄링과 같은 부가적인 고려사항들이 필요하기 때문에 구현 복잡도는 H로 선정하였다.

**QS\_06. 동시에 연결할 수 있는 디바이스의 수는** 다양한 USB 장치를 동시에 사용하는 사용자 경험을 참고하였 때 사용성 측면의 핵심적인 요인으로 판단하여 중요도는 H로 선정하였다. 동시 연결을 지원하기 위해서는 1개 연결에 대한 기능을 N개의 연결로 확장하면 구현 가능할 것으로 판단하여 구현 복잡도는 M으로 선정하였다.

**QS\_07. 새로운 디바이스 클래스 추가 용이성은** 개발자 측면에서 새로운 디바이스 클래스가 추가되는 경우 이를 시스템에 쉽게 적용할 수 있어야 하며, 제품의 빠른 업데이트 및 확장성이 필요한 USB System을 고려하여 중요도는 H로 선정하였다. 클래스의 추가를 위해서는 별도의 추상화된 계층과 잘 정의된 Interface를 구현하면 될 것으로 판단하여 구현 복잡도는 M으로 선정하였다.

**QS\_08. 새로운 실행 환경(HW, OS) 지원 용이성은** 개발자 측면에서 새로운 실행환경에 제품을 적용하기 위한 개발 비용을 의미하며, 새로운 호스트 시스템으로 제품을 확장하기 위해서는 반드시 고려되어야 하는 요구사항으로 판단하여 중요도는 H로 선정하였다. 구현 측면에서는 OS나 HW를 위한 별도의 추상화 계층을 추가하거나 잘 정의된 Interface를 활용하는 경우 지원이 가능할 것으로 판단하여 M으로 선정하였다.

**QS\_09. 데이터 전송 오류 복구 비율은** 구현 측면에서는 다양한 오류 상황에 대한 복구 기능이 구현되어야 하기 때문에 H로 선정하였다. USB 스토리지나 USB 디바이스 동작에 영향을 줄 수 있는 품질 속성으로 사용자 측면에서 중요한 요인이지만, 데이터 전송 중 오류 발생 확률과 같은 다른 QS을 통해서도 향상할 수 있을 것으로 판단하여 품질 속성으로 선정하지는 않았다.

**QS\_10. 데이터 전송 과정의 보안성은** 구현 측면에서는 기준에 구현된 보안 레벨이 높은 암호화 알고리즘(혹은 Library)을 적용하는 것으로 보안성을 높일 수 있을 것으로 판단하여 M으로 선정하였다. 하지만, 대부분의 USB 디바이스의 사용환경이 개인화된 컴퓨팅 환경에서 사용되는 것을 고려하여 중요도가 높지 않다고 판단하여 품질 속성으로 채택하지는 않았다.

**QS\_11. 시스템이 사용하는 리소스 사용량을 줄이기 위해서는 시스템의 성능을 높이면서 리소스 사용량을 최소화하기 위해서는 다양한 최적화 기법과 알고리즘의 적용이 필요하고, 이에 영향을 받는 시스템 내부 모듈들도 다양할 것으로 판단하여 구현 복잡도는 M으로 선정하였다. 하지만, 전체 호스트 시스템 측면에서는 효율적인 자원 관리가 필요하지만, USB 데이터 전송에 필요한 시스템 자원의 절대량이 높지 않을 것으로 판단하여 중요도가 높지 않은 시나리오로 판단하여 품질 속성으로 선정하지는 않았다.**

## D. 후보 구조

### D1. NFR\_01 디바이스 연결 성공 비율

본 과제에서 대상으로 하는 모바일 장치의 경우 사용자가 빈번하게 USB 디바이스를 시스템에 연결하고 해제할 수 있다. 따라서, 디바이스 연결/해제 기능은 시스템이 제공해야 하는 기본적인 기능으로 99.99% 이상의 연결 성공률을 보장해야 한다. 다만, USB System의 자원은 제한되어 있기 때문에 동시에 지원할 수 있는 디바이스의 수 역시 제한될 수 있다. 따라서 디바이스가 실패하는 상황은 디바이스의 자원이 충분한 상황(동시 지원할 수 있는 디바이스의 수를 초과하지 않은 상태)에서 디바이스 연결이 실패하는 상황만을 고려한다.

연결 성공 비율을 높이기 위해서는 시스템 가용성(Availability)관점에서 오류 발생 가능성을 줄이고, 오류 발생 시 복구할 수 있는 방안에 대한 검토가 필요하다. 따라서, 먼저 디바이스 연결 과정에 발생할 수 있는 오류 요인들을 분석하고, 각 오류 요인을 완화할 수 있는 후보 구조들을 설계하였다. 다음은 디바이스 연결 성공 비율에 영향을 줄 수 있는 오류 요인과 이를 완화할 수 있는 후보 구조 분석을 요약하여 도식화한 것이다.

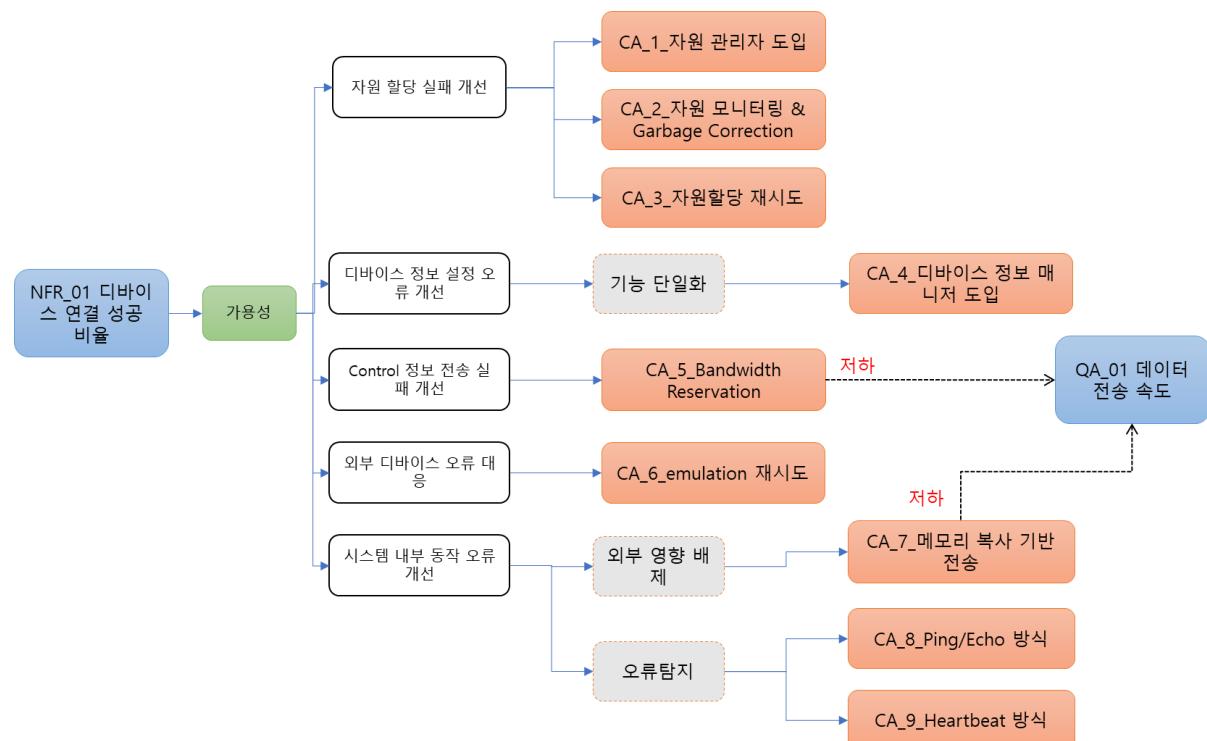


그림 59 디바이스 연결 성공 비율 향상을 위한 후보 구조 분석

시스템 내부의 디바이스 연결을 위한 대략적인 동작 과정은 다음과 같은 과정을 통해 수행된다.

1. 시스템이 디바이스 연결을 인지한다.
2. 시스템은 디바이스에 필요한 자원(Address, Bandwidth, Power등)을 할당한다.
3. 시스템은 디바이스에 Configuration을 요청한다.
4. 시스템은 디바이스 정보를 관리하는 저장소에서 디바이스와 연결된 Application을 확인하고 디바이스 연결을 통보한다.

위 과정에서 디바이스 연결 실패가 발생할 수 있는 가능성 있는 요인은 다음과 같다.

- **오류 요인 1.** 시스템에서 디바이스에 필요한 자원(Bandwidth, Power) 할당 실패
  - 디바이스의 자원 관리 오류로 디바이스 연결 해제 과정에서 자원에 대한 Leak이 이 발생하거나 시스템 내부의 컴포넌트가 동기화 문제가 발생할 수 있다.
- **오류 요인 2.** 디바이스 정보 설정 오류
  - 모바일 장치에서 디바이스 연결/해제는 빈번하게 발생할 수 있으며, 이 과정에서 이전 디바이스 정보가 남아 있는 경우 중복된 장치 등록으로 충돌이 발생할 수 있다.
  - 시스템 내부 컴포넌트가 디바이스 정보 접근에 대한 동기화 문제가 발생할 수 있다.
- **오류 요인 3.** 시스템에서 디바이스로 전달하는 Control 및 Configuration 정보 전달 지연
  - Transaction에 대한 Scheduling 과정에서 Control 정보 및 Configuration 정보 보다 높은 순위의 Transaction이 계속해서 발생하는 경우, Starvation이 발생할 수 있다.
- **오류 요인 4:** 외부 디바이스의 오류
  - 호스트 장치 외적인 요인으로, 연결 시도 중 외부 디바이스의 오류로 emulation 과정에 timeout등 오류가 발생할 할 수 있다.
- **오류 요인 5.** 시스템 내부 동작 오류
  - 시스템 내부의 디바이스 연결과 관련된 컴포넌트가 비정상 동작(특정 조건에서 코드 오류, 데이터 처리 오류 등)하는 경우가 발생할 수 있다.

<오류 요인 1>, <오류 요인 2>의 경우 디바이스 자원 및 디바이스 정보의 관리 측면에서 해제 할당 과정에서 오류가 발생할 가능성을 줄일 수 있는 구조에 대한 검토가 필요하다. <오류 요인 3>의 경우 Transaction의 스케줄링 관점에서 제한된 Bandwidth를 효율적으로 사용하면서도 특정 Transaction 요청에 대한 Starvation이 발생하지 않도록 보장할 수 있는 구조에 대한 검토가 필요하다. <오류 요인 4>의 경우 시스템 외부적인 요인으로 연결이 실패한 경우로 외부 장치를 reset하고 연결을 Retry 할 수 있는 구조에 대한 검토가 필요하다. <오류 요인 5>의 경우 시스템 내부의 오류 발생 요인을 줄일 수 있는 구조에 대한 검토가 필요하다. 또한, 주요 컴포넌트의 정상 동작 상태를 모니터링 하여 오류 상태를 확인하고 필요시 Recovery할 수 있는 구조에 대한 검토가 필요하다.

상기 설명된 오류 요인과 이에 대한 대응 방안을 바탕으로 도출된 후보 구조는 다음과 같다. <오류 요인 3>을 개선하기 위한 후보 구조 중 Transaction의 Scheduling에 관한 내용은 "D3. QA\_01 데이터 전송 속도"에 관한 후보 구조에서 자세히 다룬다.

#### D1.1. CA\_1. 자원 관리자를 통한 시스템 자원 관리 통합

<오류 요인 1>을 개선하기 위한 후보 구조로, 자원 관리를 통합 관리하는 구조를 고려할 수 있다. 시스템이 디바이스와 연결을 설정하기 위해서는 Power, Bandwidth 등의 USB BUS의 자원이 필요하다. 해당 자원을 시스템의 각 컴포넌트가 할당하고 해제하는 경우 자원을 관리하기 위한 코드의 복잡도가 높아지고, 각 컴포넌트 사이의 자원할당/해제시 동기화 문제등으로 인한 코드 오류 등이 발생할 가능성도 높아진다. 따라서 시스템 내부적으로 각 디바이스에 할당되는 자원을 관리하는 별도의 컴포넌트를 구성하고, 해당 컴포넌트를 통해서만 자원을 관리할 경우 자원관리에 대한 코드 복잡도와 동기화 문제등을 개선할 수 있다. 또한, 디바이스와 통신에 필요한 전체 자원을 별도의 Object로 생성하여 관리할 경우 디바이스 연결/해제시 해당 Object의 create/destroy를 통해 자원 관리를 연동 할 수 있는 장점도 있다.

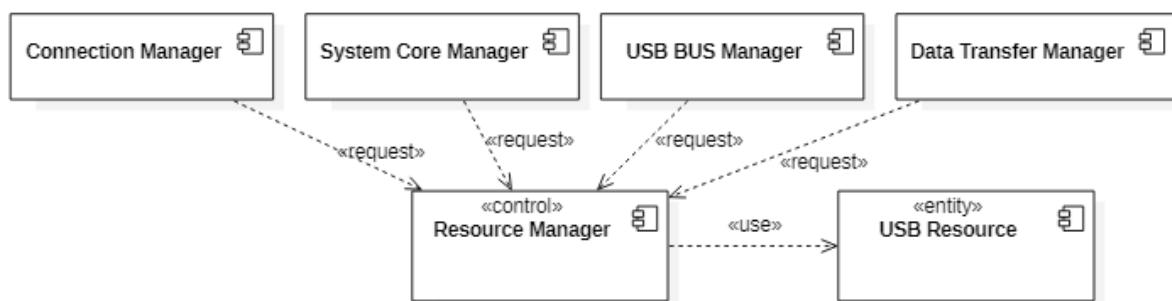


그림 60 단일 Resource Manager 구조

### D1.2. CA\_2. 낭비되는 자원에 대한 garbage correction

<오류 요인 1>을 개선하기 위한 후보 구조이다. Resource Monitor는 Endpoint에 할당된 자원을 주기적으로 모니터링 하여, 할당된 자원이 실제 사용되고 있는지를 확인한다. 할당된 자원이 임계치(threshold) 이상의 시간 동안 사용되지 않고 있는 경우 해당 디바이스와의 연결 오류나 자원 할당/해제 과정의 오류로 판단하고 자원을 회수하여 가용자원으로 다시 사용한다. Resource Monitor는 주기적으로 자원의 상태를 모니터링 해야 하기 때문에 별도의 thread로 구성한다. Resource Monitor는 Transaction Manager로부터 자원 사용에 대한 정보를 실시간으로 수집하고, 해당 정보와 Resource Manager에 할당된 정보를 비교하여 할당된 자원이 사용되지 않는 endpoint에 대한 자원을 회수한다.

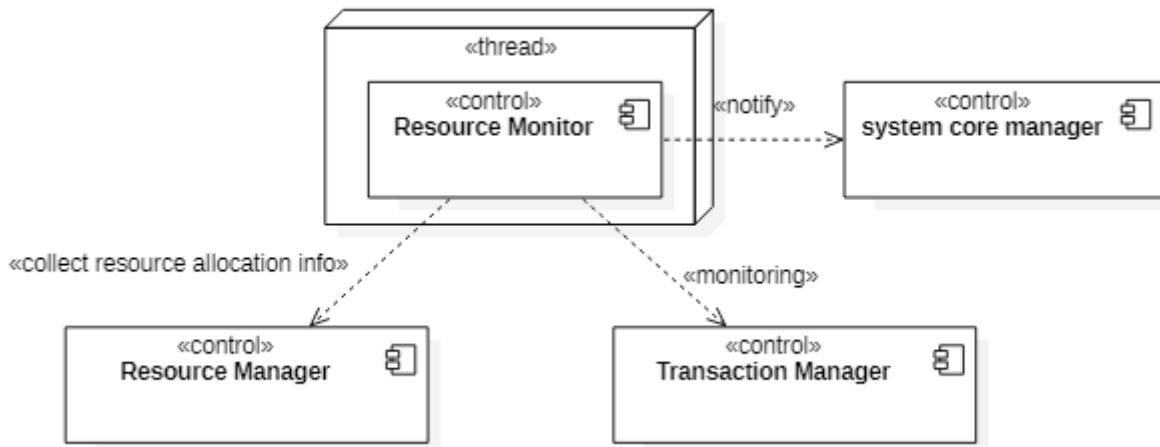


그림 61 Resource Monitoring 구조

### D1.3. 자원 할당 재시도

<오류 요인 1>을 개선하기 위한 후보 구조로, 가장 일반적인 오류 극복 방법으로 자원 할당을 재시도 하는 방법을 적용할 수 있다. 시스템의 일시적 오류, 자원의 일시적 부족 등의 상황에서 자원 할당을 재시도 하여 자원 할당을 받을 수 있다. 자원 할당 과정에서 일부 자원은 할당된 상태이면 할당되지 않은 자원에 대해서만 재할당 시도를 수행한다. 자원 할당 재시도시에도 반복적으로 실패하는 경우는 시스템의 가용 자원이 없는 상태일 가능성이 높고, 반복적인 자원 할당 재시도로 시스템 부하를 증가시킬 우려가 있기 때문에 자원 할당 재시도 횟수는 3회로 제한한다.

### D1.4. 디바이스 매니저를 통한 디바이스 정보 통합 관리

<오류 요인 2>를 개선하기 위한 후보 구조이다. 시스템이 디바이스와 연결하기 위해서는 디바이스에 할당된 Address, Description, Configuration, 디바이스 상태, 사용하는 Application 정보 등과 같은 USB Host Driver

많은 정보들이 관리되어야 한다. 이러한 정보들을 각각의 컴포넌트들이 개별 관리하는 경우 정보의 동기화 및 관리의 복잡성이 발생할 수 있다. 시스템 내부에 디바이스정보를 관리하는 별도의 컴포넌트를 적용하고 해당 컴포넌트를 통해서만 디바이스의 정보에 접근하도록 할 경우 디바이스 정보 관리 과정에서 발생하는 오류를 줄일 수 있다.

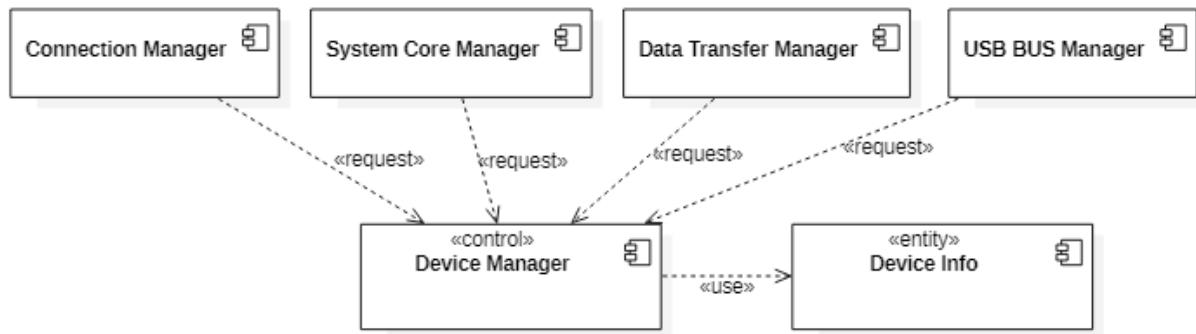


그림 62 Device Manager 구조

#### D1.5. CA\_5. Control 정보를 위한 Bandwidth Reservation

<오류 요인 3>을 개선하기 위한 후보 구조이다. 일반적인 상황에서 시스템은 Periodic transfer(Isochronous, Interrupt) 방식의 데이터를 전송에 우선 순위를 주고 자원을 할당 하기 때문에 USB BUS의 bandwidth 중 90%이상을 Periodic transfer를 위해 할당할 수 있다. Periodic transfer가 BUS bandwidth의 90%이상을 점유한 상황에서 Control 정보를 전달하기 위한 Transaction이 요청되는 경우 우선 순위에 따라 control 정보 전달의 지연, 혹은 전송 실패가 발생할 수 있다. 이러한 상황을 방지하기 위해 시스템은 새로운 디바이스 연결이 확인된 경우 Periodic transfer에 할당하는 Bandwidth의 최대 허용치를 설정하고 이를 넘는 경우 Bandwidth 할당을 제한하여 Control 정보 전달을 보장한다. 디바이스 연결이 완료된 경우 Threshold 설정을 해제한다.

#### D1.6. CA\_6. Emulation 과정을 재시도

<오류 요인 4>를 개선하기 위한 후보 구조이다. 시스템과 디바이스는 연결 과정에서 Attached → Powered → Default → Address → Configured의 과정을 거쳐 최종적으로 Configuration이 완료된 경우 시스템은 해당 디바이스를 사용할 수 있게 된다. 시스템이 이와 같은 디바이스와의 연결을 진행하는 과정에서 디바이스의 내부적 오류가 확인 되는 경우(protocol 오류, 전송 시간 timeout 등) 시스템은 디바이스와의 연결을 재시도 할 수 있다. Emulation 과정을 재시도 하는 것은 디바이스가 물리적으로 연결된 상태(연결은 유지된 상태를 가정)를 가정하고 있기 때문에, 디바이스를 SW 적으로 Reset 할 수 있도록 Reset 요청을 전달하는 과정부터 재시도 한다. 재시도 과정에서 디바이스에 이미 할당

된 자원(Address, Bandwidth, Power 등)이 있는 경우 해당 자원은 모두 회수한다. 재시도시에도 디바이스와 연결이 되지 않는 경우 SW적으로 복구 불가능한 상태일 가능성이 높고, 연속적인 재시도로 인한 시스템 부하가 발생할 가능성이 높기 때문에 재시도 횟수는 최대 3번까지만 허용한다.

#### D1.7. CA\_7. 메모리 복사 기반의 전송 데이터 전달

<오류 요인 5>를 개선하기 위한 후보 구조이다. Application에서 전달된 요청은 시스템 내부적으로 여러 개의 컴포넌트를 거쳐 최종적으로 Host Controller로 전달 된다. 이 과정에서 시스템과 시스템 외부와의 영향을 최소화 하기 위해 전달 되는 데이터를 모두 복사하여 사용한다. 전달되는 데이터에 대한 외부 메모리 참조가 제거 되기 때문에 시스템 외부에서 발생하는 오류로 인해 시스템이 영향 받는 부분을 최소화 할 수 있다. 예를 들어 Application의 오류로 데이터 buffer를 overflow 시키는 경우, 시스템과 Application이 동일한 데이터 buffer를 참조하고 있는 경우 시스템의 동작이 Application에 의해 영향을 받을 수 있다. 반면, 시스템과 Application이 사용하는 데이터 buffer가 분리되어 있는 경우는 Application 오류가 시스템에 직접 영향을 주지는 않기 때문에 시스템의 안정성을 향상 시킬 수 있다.

#### D1.8. CA\_8. Ping/Echo 기반 시스템 health check

<오류 요인 5>를 개선하기 위한 후보 구조이다. 시스템이 디바이스와 연결을 완료하기 위해서는 디바이스 연결과 관련된 다수의 컴포넌트(Core System Manager, Device Manager, Resource Manager 등)들이 정상적으로 동작 하여야 한다. 이러한 각 컴포넌트가 가용성을 높이기 위해 시스템 내부에 각 컴포넌트의 정상 동작 여부를 확인하는 Health Check 기능을 적용하여 디바이스 연결과 관련된 각 컴포넌트의 비정상 동작을 탐지하고 복구할 수 있다.

다음 그림은 Ping/Echo 기반으로 각 컴포넌트의 동작 상황을 모니터링하는 구조이다. Health Manager는 주기적으로 시스템의 상태를 모니터링 해야 하기 때문에 별도의 Thread로 구성하고 각 주기마다 각 컴포넌트에 ping 메시지를 전달하고 각 컴포넌트가 전달하는 echo 메시지를 통해 정상 동작 여부를 판단한다. 정상 동작하지 않는 컴포넌트는 Core System Manager에 통보하여 복구를 시도한다.

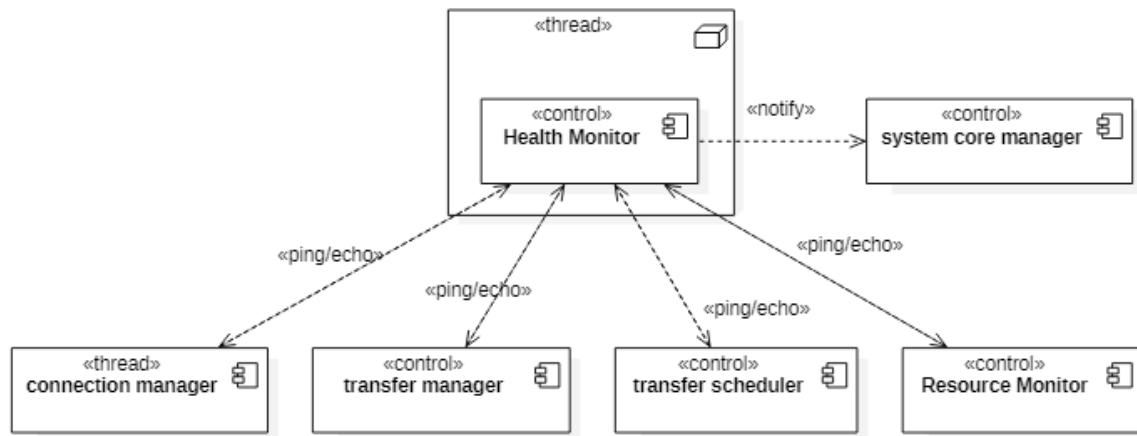


그림 63 ping/echo 기반 Health check 구조

#### D1.9. CA\_9. Heart bit 기반 시스템 health check

<오류 요인 5>를 개선하기 위한 후보 구조이다. 시스템 내부의 각 컴포넌트들이 일정 주기마다 Health Manager로 heart bit을 전달하는 방식으로 각 컴포넌트의 정상 동작 여부를 판단할 수 있다. Health Manager는 일정 시간이상 heart bit이 전달되지 않는 컴포넌트는 비정상 동작 상태로 판단하여 해당 컴포넌트의 복구를 core system manager에 요청한다.

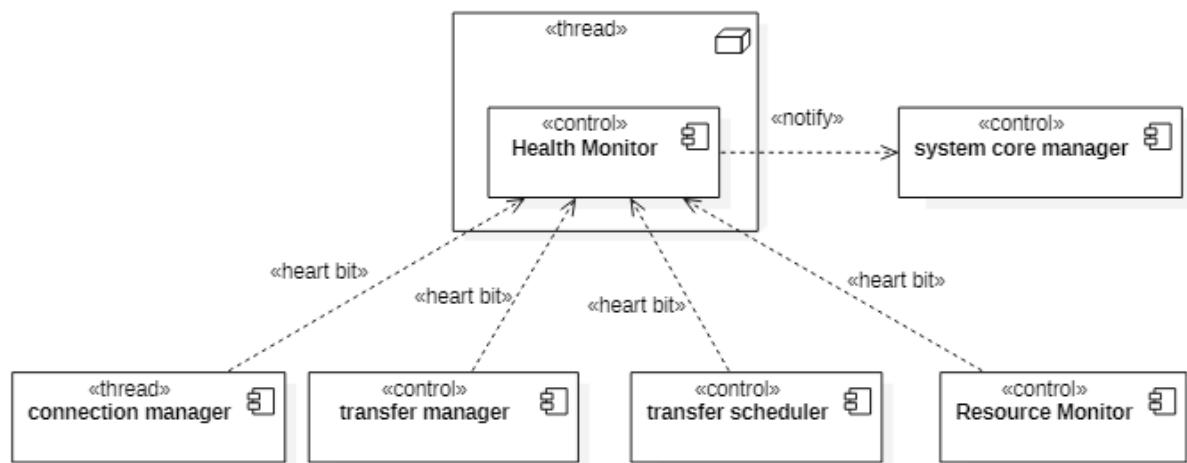


그림 64 heart bit 기반 Health check 구조

#### D2. NFR\_02 디바이스로부터 입력에 대한 처리 시간

디바이스로부터 입력에 대한 처리시간을 줄이기 위해서는 성능 관점에서 시스템 내부 동작 지연을

줄이고 Application과의 Communication 시간을 줄이는 방안 등에 대한 검토가 필요하다. 따라서, 먼저 디바이스 입력을 시스템이 처리하는 과정을 분석하고 각 과정에서 처리 시간을 줄일 수 있는 방안들을 후보 구조로 설계하였다. 다음은 디바이스 입력 처리 시간에 영향을 주는 요인과, 각 성능 요인을 개선하기 위한 후보 구조 분석을 요약하여 도식화한 것이다.

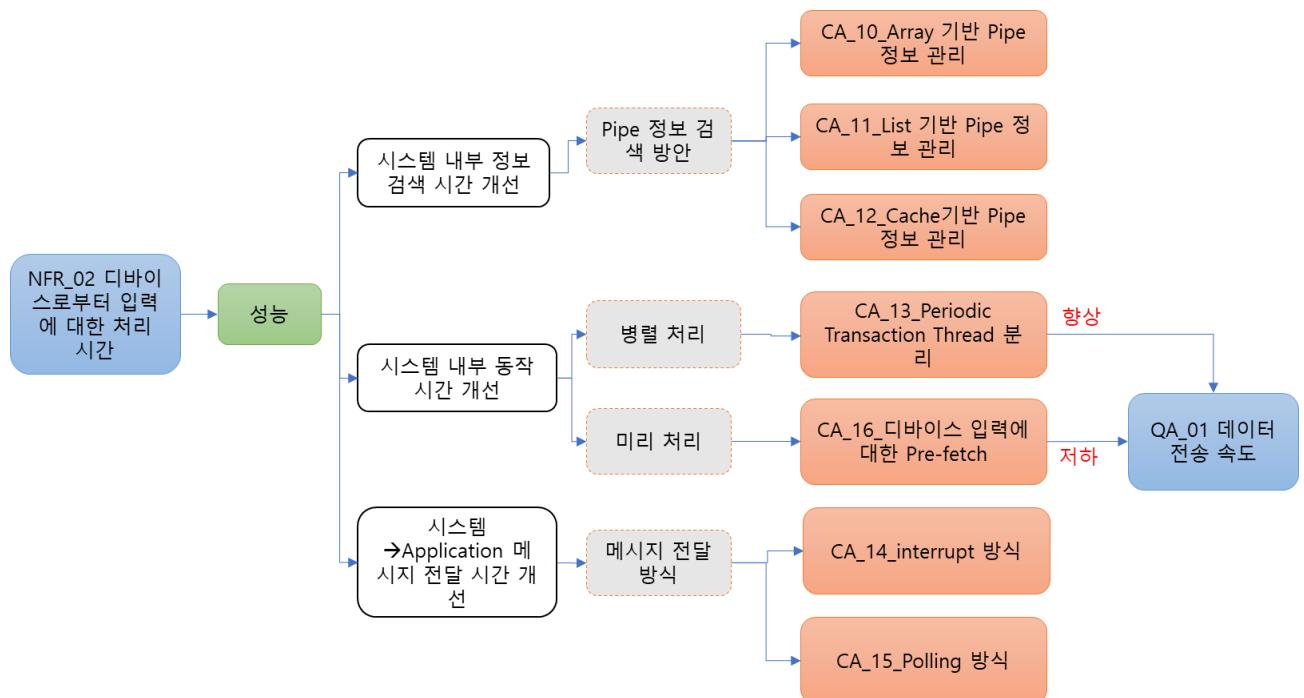


그림 65 디바이스 입력 처리 시간에 대한 후보 구조 분석

시스템에서 키보드 마우스와 같은 HID 디바이스의 입력을 처리하는 과정은 대략적으로 다음과 같다.

1. Application에서 시스템에 디바이스 입력 Read 요청
2. 시스템은 디바이스 정보 저장소에서 Application이 요청한 Pipe에 연결된 디바이스의 endpoint 검색
3. Endpoint 설정에 명시된 interrupt 주기 확인
4. 해당 주기마다 디바이스의 입력을 확인하여 Application에 전달

위 과정에서 디바이스로부터 입력 처리 시간에 영향을 주는 요인은 다음과 같다.

- **성능 요인 1:** 시스템에서 Application이 데이터 전송을 요청한 Pipe와 매칭되는 endpoint의 정보를 검색하는데 걸리는 시간
- **성능 요인 2:** 시스템에서 디바이스의 입력을 처리하는데 걸리는 시간
- **성능 요인 3:** 시스템에서 Application에 디바이스 입력을 전달하는데 걸리는 시간

<성능 요인 1>을 개선하기 위해서는 Pipe와 매핑되는 디바이스의 endpoint를 검색하는데 걸리는 시간을 최소화 할 수 있는 자료 구조와 검색 알고리즘에 대한 검토가 필요하다. <성능 요인 2>와 <성능 요인 3>은 디바이스에서 입력된 데이터를 빠르게 인식하고, 시스템 내부에서 처리하는 과정의 시간 지연을 최소화 할 수 있는 SW 구조 검토가 필요하다.

상기 설명된 성능 영향 요인을 바탕으로 도출된 후보 구조는 다음과 같다.

#### D2.1. CA\_10. Array 기반의 pipe 매핑 정보 관리

<성능 요인 1>을 개선하기 위한 후보 구조이다. 시스템은 Pipe에 연결된 Application과 디바이스의 endpoint를 확인하기 위한 매핑 정보를 관리해야 한다. 매핑 정보에는 기본적으로 <pipe, address, endpoint, application> 정보가 포함되어야 한다. 시스템에 연결될 수 있는 최대 디바이스의 수(127개)와 각 디바이스가 지원할 수 있는 최대 endpoint의 수(32개)를 고려하면 시스템은 최대 4064(최대 디바이스 수 \* 디바이스의 endpoint 수)개의 endpoint와 Application간의 매핑 정보를 관리해야 한다. 따라서 이러한 정보를 효율적으로 관리하면서 Application이 요청한 target endpoint를 빠르게 검색하는 것이 매우 중요하다.

시스템이 관리해야 하는 정보의 최대 크기는 4064개를 넘지 않기 때문에 해당 정보들을 배열에 저장하는 것을 우선 고려할 수 있다. 저장되는 정보는 아래와 같은 구조로, Pipe는 <address, endpoint> 정보에 대한 참조 번호 역할을 하며 순차적으로 증가한다.

Pipe num	Address(device id)	endpoint	Application id(process id)
0	0	0	1000
1	0	1	1000
2	0	2	1000
3	0	3	1000
⋮			
32	0	31	1000
33	1	0	3459
⋮			
4063	127	31	8889

그림 66 Array 기반 Pipe/Endpoint 매핑 정보 관리

배열로 저장할 경우 구현이 간단하고 Pipe를 배열을 index로 활용하여 검색 시간을 O(1)로 줄일 수 있다. 하지만, 불필요한 메모리 공간의 낭비가 발생할 수 있는 단점이 있다.

#### D2.2. CA\_11. List 기반의 pipe 매핑 정보 관리

<성능 요인 1>을 개선하기 위한 후보 구조이다. 메모리 낭비를 줄이기 위해 List 기반의 매핑 정보 관리 구조를 고려할 수 있다. ArrayList 기반의 방식과는 다르게 디바이스가 연결되는 경우 해당 디바이스에 할당되는 address를 기반으로 <pipe, address, endpoint, application> 정보를 생성하고 List에 추가하고 연결이 해제되는 경우 List에서 삭제한다. 해당 방식의 경우 현재 연결된 디바이스의 수에 필요한 메모리 공간만을 할당해서 메모리를 절약할 수 있지만, Pipe에 해당하는 address, endpoint 정보를 검색하는데 Worst case에 O(N) 만큼의 시간 지연이 발생할 수 있다.

#### D2.3. CA\_12. Cache 기반의 pipe 매핑 정보 관리

<성능 요인 1>을 개선하기 위한 후보 구조이다. List 방식의 검색 시간을 줄이기 위한 방식으로 요청 발생 가능성이 높은 매핑 정보를 저장할 별도의 table을 운영하여 매핑 정보 확인 시 해당 table을 1 차 검색하고 해당 table에 정보가 없는 경우 전체 List에서 검색한다. 캐시는 LRU 알고리즘을 적용하여 가장 최근에 사용된 pipe의 정보를 저장하도록 한다. 캐시 size가 커지면 검색 시간이 늘어나기 때문에 검색 성능을 고려하여 최대 32개의 entry만 저장하도록 한다.

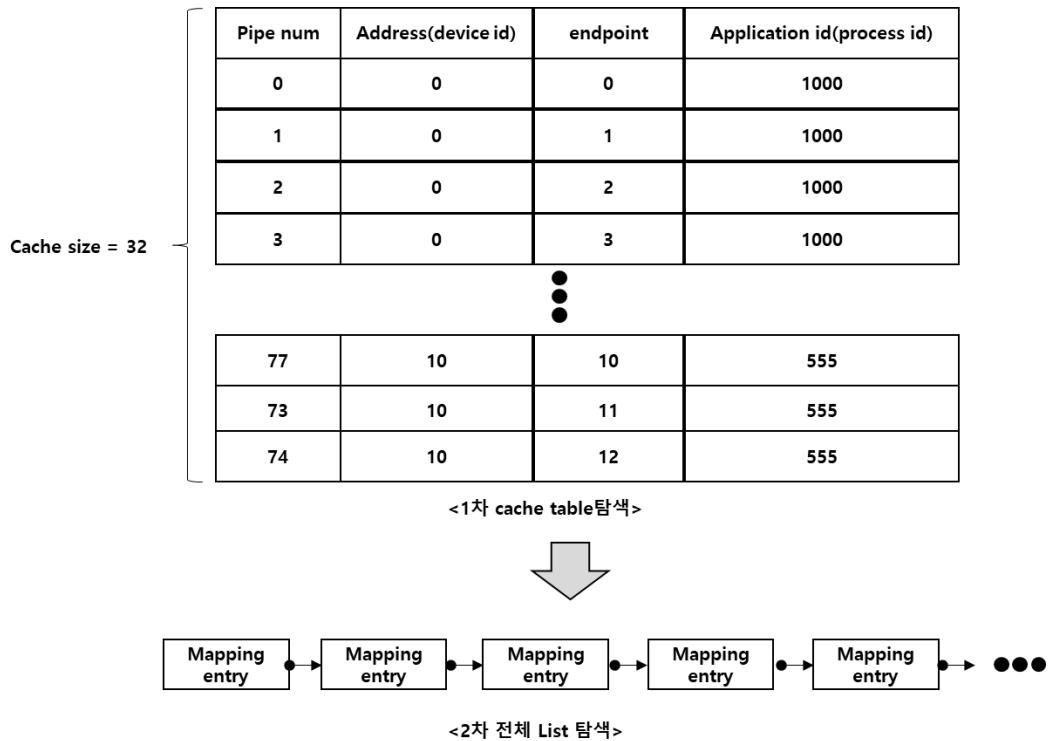


그림 67 Cache 기반 Pipe/Endpoint 매핑 정보 관리

#### D2.4. CA\_13. Periodic Transaction 처리를 위한 전용 Thread 운영 구조

<성능 요인 2>를 개선하기 위한 후보 구조이다. HID 입력과 같은 Interrupt 방식의 입력의 경우 임의의 시점에 입력이 발생할 수 있고, 해당 정보를 확인하기 위해서는 Polling 형태로 디바이스의 입력을 주기적으로 Read해야 한다. 따라서, HID에 대한 입력 요청과 같은 Periodic Transaction을 별도의 Thread로 분리하여 처리할 경우, 시스템의 다른 기능 수행을 blocking 하지 않고 병렬로 처리할 수 있다. Thread 내부적으로는 Periodic Transaction의 요청 목록을 관리하며, Read 주기가 도래한 경우 Transaction을 생성하여 Transaction Manager에게 전송을 요청한다.

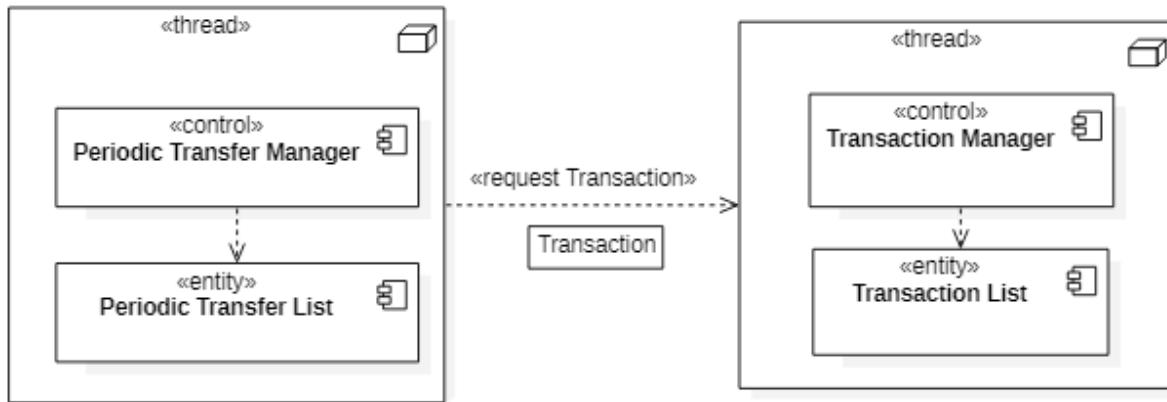


그림 68 주기적 Transaction 처리 구조

#### D2.5. CA\_14. Interrupt 방식의 시스템↔Application간 이벤트 전달 구조

<성능 요인 3>를 개선하기 위한 후보 구조이다. Interrupt 전달 구조는 시스템이 Application으로 디바이스 입력을 통보하는 방식이다. 시스템은 endpoint에 설정된 주기에 따라 디바이스의 입력을 읽어서 디바이스의 입력이 있는 경우 그 내용을 Application에 통보한다. Application과 시스템이 별도의 Process 혹은 Thread로 동작하는 경우 Polling으로 인한 오버헤드가 발생하는 반면, Event 발생시에만 통보하기 때문에 빈번한 IPC와 context switching으로 인한 시스템 오버헤드를 줄일 수 있다. 또한 일반적으로 ISR은 시스템의 실행 환경(i.e., OS)에서 높은 우선 순위를 가지기 때문에 디바이스에서 Read한 내용을 빠르게 Application으로 전달할 수 있다. Application의 구현과 관계없이 Event polling을 시스템 내부적으로 보장할 수 있어 신뢰성 측면 및 개발 유지 보수 측면에서도 이점을 가질 수 있다.

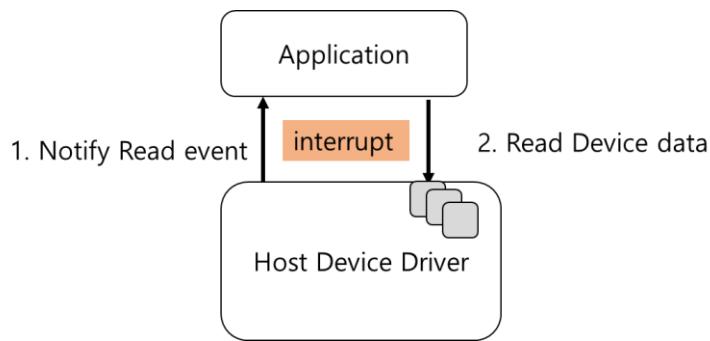


그림 69 시스템이 Application에 Read Event를 전달하는 구조

#### D2.6. CA\_15. Polling 방식의 시스템↔Application간 이벤트 전달 구조

<성능 요인 3>를 개선하기 위한 후보 구조이다. Application이 데이터 전달을 요청하는 구조는 USB Host Driver

Application이 Polling 주기를 관리하고 시스템은 별도의 주기를 관리 하지 않는다. Application은 endpoint의 설정 정보를 이용하여 디바이스의 입력을 Read할 주기를 설정하고, 해당 주기마다 시스템으로 디바이스에 대한 read 요청을 전달한다. 시스템은 Application이 요청한 Pipe에 해당하는 endpoint로부터 data를 read하여 Application에 전달한다.

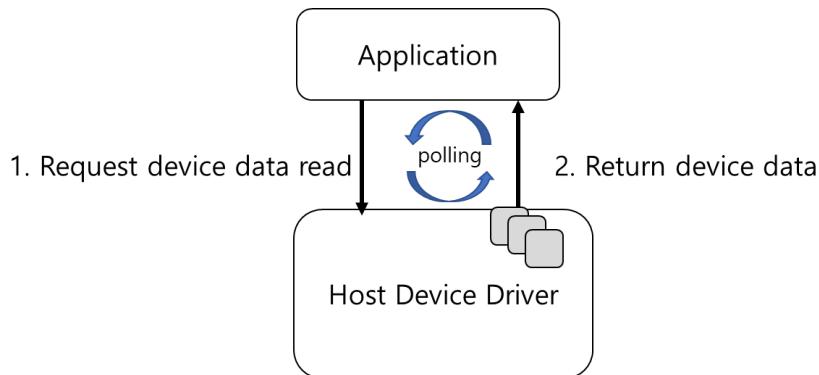


그림 70 Application이 시스템에 데이터 Read를 요청하는 구조

#### D2.7. CA\_16. 디바이스 입력에 대한 Pre-fetch 구조

<성능요인 2>를 개선하기 위한 후보 구조이다. CA\_15와 같이 Application의 요청에 따라 시스템이 데이터를 전달하는 방식이다. 추가적으로 데이터 Read 시간을 줄이기 위해 시스템에서 endpoint에 설정된 주기보다 짧은 주기로 디바이스의 입력을 미리 read하여 시스템 내부 buffer에 저장하고, Application에서 Read 요청 시 내부 Buffer에 저장된 값을 즉시 반환한다. 해당 구조의 경우 입력에 대한 반응 시간은 빨라지지만 USB BUS의 bandwidth가 늘어나는 단점이 있다.

### D3. QA\_01 데이터 전송 속도

데이터 전송 속도는 사용자 경험 측면에서 매우 중요하며, 동일한 데이터를 빠르게 전송 완료하여 디바이스가 suspend 상태로 진입할 수 있는 시간이 빨라 질수록 Power 효율 측면에서도 이점을 가질 수 있다. 하지만, USB System의 경우 가용한 물리적인 자원(ex, Bandwidth)가 제한되어 있다. 따라서 데이터 전송 속도를 높이기 위해서는 제한된 자원을 효과적으로 관리하고, SW적인 오버헤드를 줄이는 방안 등에 대해서 성능 측면에서 검토가 필요하다. 이를 위해 먼저 시스템의 데이터 전송 과정을 분석하고 각 과정에서 SW 오버헤드를 시간을 줄일 수 있는 방안들을 후보 구조로 설계하였다. 다음은 데이터 전송 속도에 영향을 주는 요인과, 각 성능 요인을 개선한기 위한 후보 구조 분석을 요약하여 도식화한 것이다.



그림 71 데이터 전송 성능에 대한 후보 구조 분석

시스템에서 데이터 전송을 처리하는 일반적인 과정은 다음과 같다.

1. 다수의 Application에서 Data를 전송(read or write)하기 위한 IRP를 Pipe를 통해 시스템에 전달
2. 시스템은 Pipe에 연결된 endpoint를 검색
3. 시스템은 전송에 필요한 정보를 설정하고 전송 data를 copy하여 Transaction을 생성
4. Transaction들을 우선 순위에 따라 scheduling하여 frame 생성
5. Application이 요청한 Transaction이 완료되는 경우 결과를 Application에 통보

위 과정에서 데이터 전송 속도에 영향을 주는 요인은 다음과 같다.

- **성능 요인 1:** 다수의 Application에서 발생하는 요청을 동시에 효율적으로 처리

- 시스템에 다수의 디바이스가 연결된 경우 시스템은 동시에 발생하는 다수의 데이터 전송 요청들을 효율적으로 처리할 수 있어야 한다.
- 시스템 내부의 컴포넌트 간의 메시지 교환에 소모되는 시간 delay를 최소화 해야 한다.
- **성능 요인 2:** IRP → Transfer 정보 → Transaction list → Frame 생성으로 변환되는 과정의 시간 지연
  - 데이터 전송을 위한 HW적인 bandwidth는 제한되어 있기 때문에 데이터 전송 과정에서 발생하는 SW적인 시간 지연을 최대한 줄여야 한다.
- **성능 요인 3:** Transfer 정보 관리 및 scheduling 효율성
  - 시스템이 다수의 디바이스와 데이터를 전송하고 있는 경우 제한된 자원을 최대한 사용 할 수 있도록 transaction을 처리 과정을 효율적으로 처리해야 한다.
- **성능 요인 4:** 자원(Bandwidth) 배분의 효율성
  - BUS 상태 변화 과정에서 발생할 수 있는 자원의 낭비를 최대한 줄여야 한다.

<성능 요인 1>, <성능 요인 2>을 개선하기 위해서는 동시에 발생하는 여러 전송 요청을 병렬적으로 처리하기 위한 SW 구조에 대한 검토가 필요하다. <성능 요인 3>은 다수의 transaction 발생 시 이들 사이의 우선 순위를 효율적으로 정할 수 있는 알고리즘 및 자료구조에 대한 검토가 필요하다. <성능 요인 4>는 데이터 전송에 필요한 자원을 효율적으로 분배하고, 낭비되는 자원을 방지하기 위한 SW 구조 검토가 필요하다.

상기 설명된 성능 영향 요인을 바탕으로 도출된 후보 구조는 다음과 같다.

#### D3.1. CA\_17. 데이터 Transfer와 Transaction을 단일 Thread에서 운영 하는 구조

<성능 요인 1>, <성능 요인 2>을 개선하기 위한 후보 구조이다. 시스템에 디바이스가 다수 연결되어 여러 개의 Application이 동시에 USB를 통한 데이터 전송을 요청하는 경우 시스템은 IRP → Transfer → Transaction → Frame으로 변환되는 과정을 관리해야 하고 처리된 Frame이 어떤 Transfer에 대한 처리 등 인지를 Trace할 수 있어야 한다. 시스템은 해당 과정들을 독립된 task로 분할하고 이를 병렬로 처리할 경우 동시에 발생하는 처리 요청에 대한 처리 효율을 높일 수 있다.

다음 그림은 데이터 Transfer 요청을 처리하는 Transfer Manager와 Transaction Manager를 하나의 독립된 Thread로 구성하여 시스템의 다른 기능과 병렬적으로 처리할 수 있는 구조에 대한 설명이다.

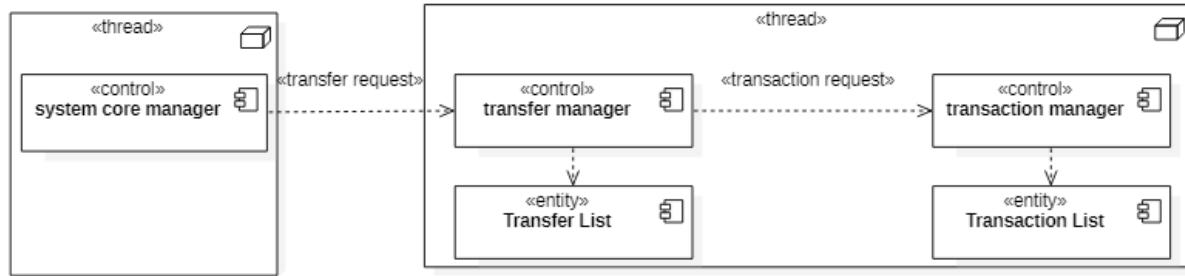


그림 72 단일 Thread 기반의 데이터 Transfer 구조

### D3.2. CA\_18. 데이터 Transfer와 Transaction을 개별 Thread에서 운영 하는 구조

<성능 요인 1>, <성능 요인 2>를 개선하기 위한 후보 구조이다. 다음 그림은 데이터 전송을 처리하는 과정을 ①Application의 request를 처리하는 과정, ② Application의 요청을 Transfer 단위로 관리는 과정, ③Transfer 요청을 Host Controller가 처리할 수 있는 Frame단위로 생성하는 과정으로 세분화 하고 각각을 처리하는 독립된 Thread를 운영하는 구조이다. 각각의 과정이 Message를 기반으로 독립적으로 동작 하기 때문에 다수의 데이터 전송 요청을 병렬 적으로 처리할 수 있다.

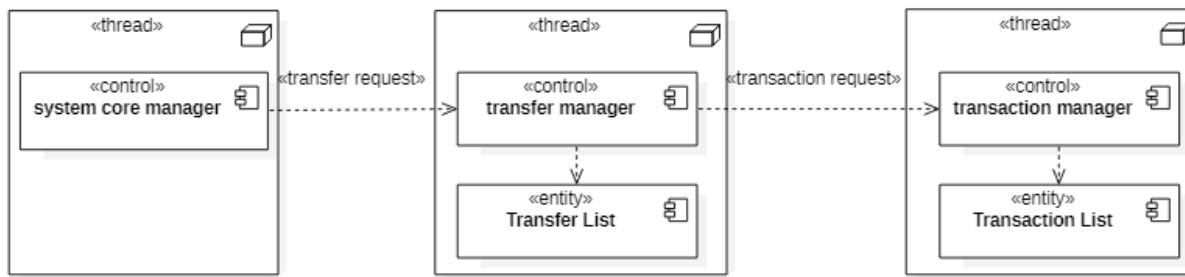


그림 73 독립된 Thread 기반 데이터 Transfer 관리 구조

### D3.3. CA\_19. Transfer 요청에 대한 Thread 생성(Creation) 구조

<성능 요인 1>, <성능 요인 2>를 개선하기 위한 후보 구조이다. 다수의 Application에서 Transfer 요청이 발생하는 경우 시스템은 각 Transfer 요청을 처리할 별도의 Thread는 생성하고 각각을 병렬 처리 하는 구조이다. 각각의 Transfer 처리가 독립적으로 동작하기 때문에 병렬 처리를 통해 Transfer 요청 중간에 발생할 수 있는 대기 시간을 hiding 할 수 있는 장점이 있다. 하지만, Transfer가 전달되는 시점에 Thread를 생성하는 등이 부가적인 시스템 오버헤드가 발생할 수 있는 단점도 있다. 다음 그림은 Transfer 요청별로 Thread를 생성하는 구조에 대한 설명이다.

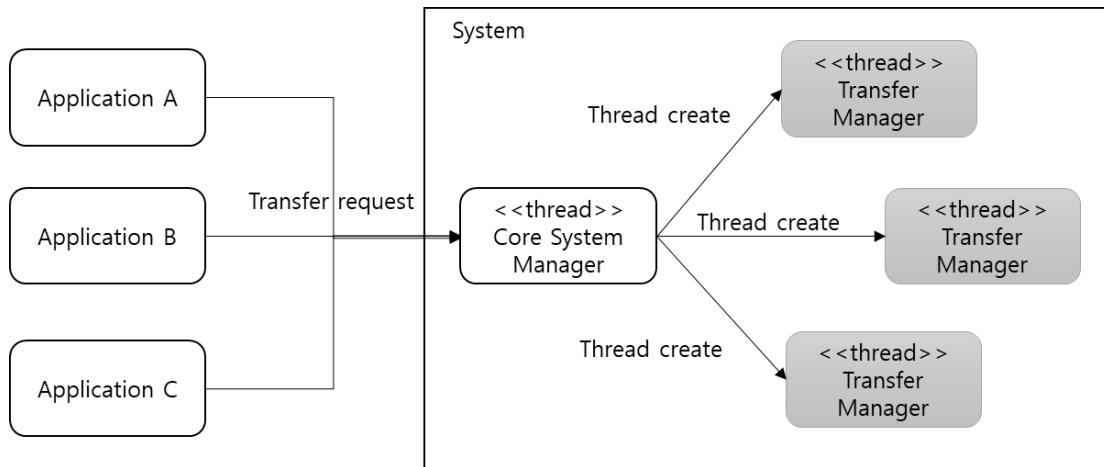


그림 74 Transfer 요청별 Thread 생성 구조

#### D3.4. CA\_20. Transfer 요청 속성별 Thread 할당 구조

<성능 요인 1>, <성능 요인 2>를 개선하기 위한 후보 구조이다. 시스템은 Transfer 요청을 처리할 Thread를 각 전송 Type별로 미리 생성해 두고, Application 요청 전달시 해당 요청을 처리할 Thread에 Transfer 처리를 위임 하는 구조이다. Transfer 요청을 처리할 Thread가 미리 생성되어 있기 때문에 Thread 생성에 대한 시스템 오버헤드가 발생하지 않는 장점이 있다. Transaction Type별로 Thread가 할당 되기 때문에 Transfer 할당을 위한 별도의 정렬등이 필요하지 않지만, 동일한 type의 Transfer가 동시에 발생하는 경우 해당 Type을 처리하는 Thread에서 병목현상이 발생할 가능성도 있다. 다음 그림은 Transfer Type별로 Thread를 운영하는 구조에 대한 설명이다.

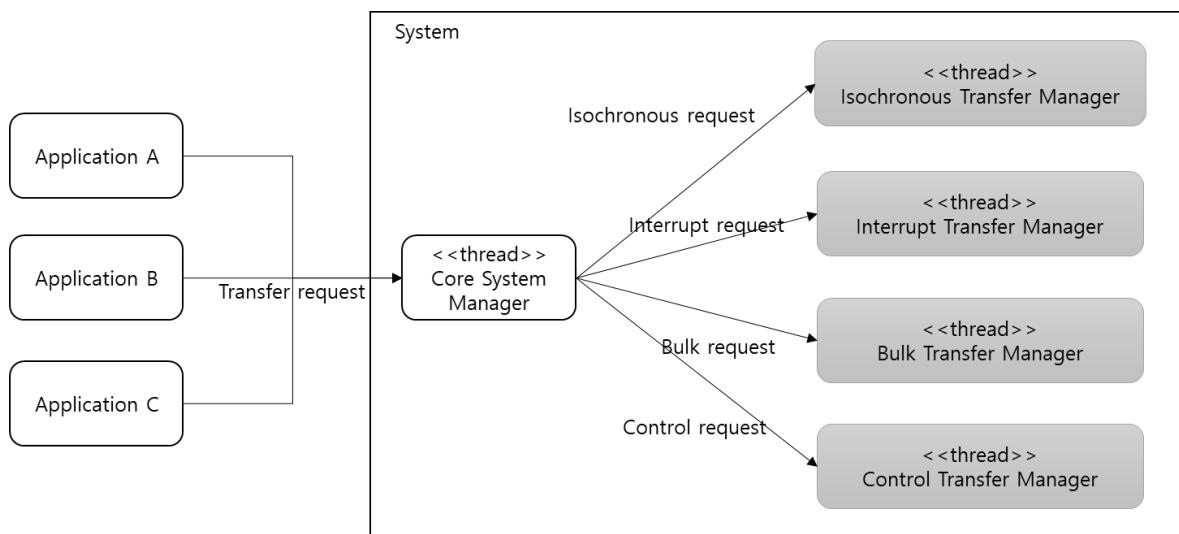


그림 75 Transfer Type별 Thread 할당 구조

### D3.5. CA\_21. Transfer 요청에 대한 Thread 임의 할당(Random Dispatch) 구조

<성능 요인 1>, <성능 요인 2>를 개선하기 위한 후보 구조이다. 시스템은 Transfer 요청을 처리할 Thread를 N개 생성해 두고, Application 요청이 전달되는 시점에 해당 요청을 처리할 Thread를 선택하여 Transfer 요청을 위임하는 구조이다. Transfer 요청이 전달되는 시점에 해당 요청을 처리할 Thread가 결정되기 때문에 이를 처리할 중계자(Dispatcher)가 필요하다. 중계자는 각 Thread의 Transfer 처리 상황을 판단하여 추가적인 Transfer 처리 요청을 할당한다. 요청 전달 시점에 Thread가 생성하는 오버헤드를 제거할 수 있고 Transfer Type별로 Thread가 고정되지 않아 여러가지 전송 상황에 flexible하게 대응할 수 있는 이점이 있다. 반면에, 중계자에서 Thread 할당 방식과 적절한 Thread의 수를 결정하는 추가적인 고려가 필요하다. 다음 그림은 Thread 임의 할당 방식의 Transfer 처리 구조에 대한 설명이다.

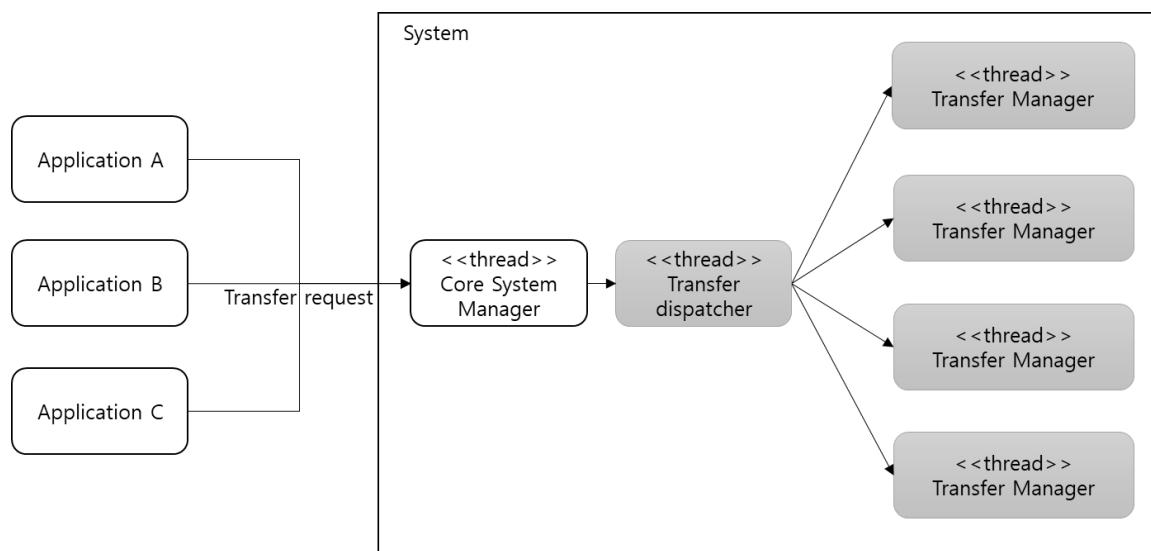


그림 76 Transfer 요청에 대한 Thread 임의 할당 구조

### D3.6. CA\_22. Round Robin 방식의 Transfer 할당 스케줄링

<성능 요인 1>, <성능 요인 2>를 개선하기 위한 후보 구조이다. Thread 임의 할당 후보 구조에서 Transfer Dispatcher를 통해 각 Transfer 요청에 대한 Thread 할당시 Round Robin 형태로 각 Transfer 요청을 할당하는 방식이다. Round Robin으로 처리되기 때문에 scheduling을 위한 별도의 오버헤드가 필요하지 않고 SW구현 복잡도가 줄어드는 이점이 있지만, Transfer를 처리하는 각 Thread 상황의 상황에 맞는 Load balance등은 어려울 수 있다.

### D3.7. CA\_23. Priority Queue 방식의 Transfer 할당 스케줄링

<성능 요인 1>, <성능 요인 2>를 개선하기 위한 후보 구조이다. Transfer 요청을 처리할 Thread 할당 시 Thread의 Load 상황을 판단하여 가장 Load가 적은 Thread도 Transfer 요청을 전달하는 구조이다. Thread의 Load를 판단하는 기준은 각 Thread가 보유한 Message Buffer의 현재 Length, 전송 message의 크기 정보를 이용하여 예상되는 Load가 적은 Thread를 선택한다. Thread간의 Load가 분산되기 때문에 효율적인 전송 처리가 가능하지만 SW 구현의 복잡도가 높아질 수 있다.

### D3.8. CA\_24. On-demand 방식의 Transaction 생성

<성능 요인 3>을 개선하기 위한 후보 구조이다. Application에서 요청한 전송 데이터의 크기가 Transaction으로 전송 가능한 크기 보다 큰 경우 시스템은 데이터 전송 요청을 transaction 단위로 분할(Fragment)한다. 분할된 각각의 transaction은 transaction list에 저장되어 우선 순위에 따라 처리된다. On-demand 방식의 transaction 생성 방식은 Transfer list에 남아 있는 요청 중 우선 순위에 따라 전송할 transfer를 선택하고 가용한 자원 만큼의 transaction을 생성하여 transaction list에 추가하는 방식이다. 즉, Host Controller가 하나의 frame 전송 완료를 통보하는 시점에 새로운 frame을 생성 하여 Host Controller에게 전달하게 된다. Frame 생성 시점에 필요한 만큼의 transaction 데이터를 생성하기 때문에 시스템의 메모리 사용을 최소화 할 수 있고, 현재 할당 가능한 bandwidth 만큼 frame 데이터를 생성하기 때문에 허용된 bandwidth를 100% 사용할 수 있다. 반면 transaction 생성에 소모되는 시간을 hiding 하기 어려울 수 있다.

다음 그림은 On-demand 방식으로 transaction을 처리하는 경우에 대한 시스템 내부 동작 흐름이다.

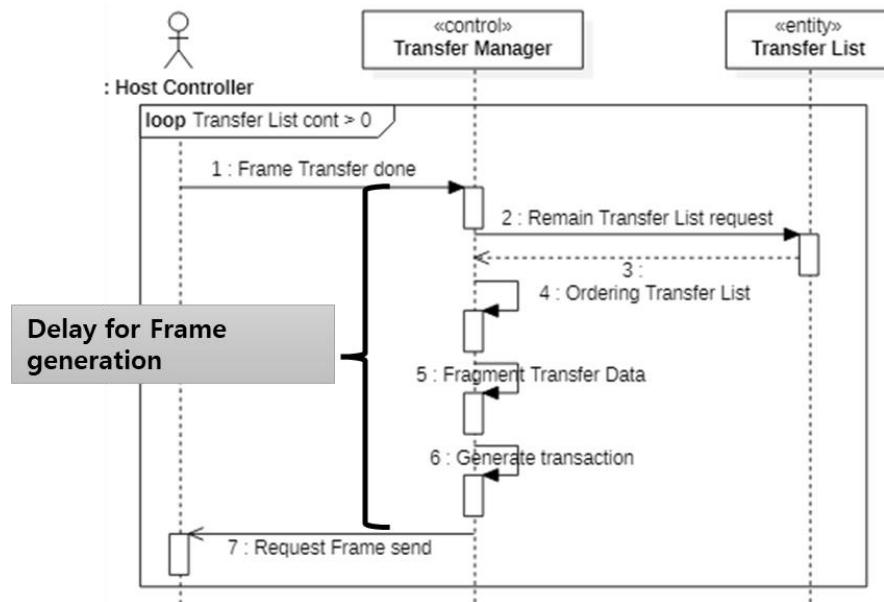


그림 77 on-demand 방식의 Frame generation 구조

### D3.9. CA\_25. Pre-generation 방식의 Transaction 생성

<성능 요인 3>을 개선하기 위한 후보 구조이다. 시스템은 Transfer 요청이 전달되면 Transfer Manager는 Transfer 요청을 transaction 단위로 분할하여 Transaction List에 저장한다. Host Controller로부터 전송 완료 명령이 전달되면 transaction List에 저장된 1개의 Frame을 Host Controller로 전달한다. Transfer 요청에서부터 transaction 생성까지 소모되는 시간을 hiding 할 수 있기 때문에 전송에 필요한 시간 delay를 줄일 수 있다. 하지만, Bulk 전송과 같이 대량의 Data를 전송해야 하는 경우 전체 Data를 미리 transaction 단위로 생성하기 위해서는 메모리 낭비가 발생할 수 있다.

다음 그림은 Pre-generation 방식으로 transaction을 처리하는 경우에 대한 시스템 내부 동작 흐름이다.

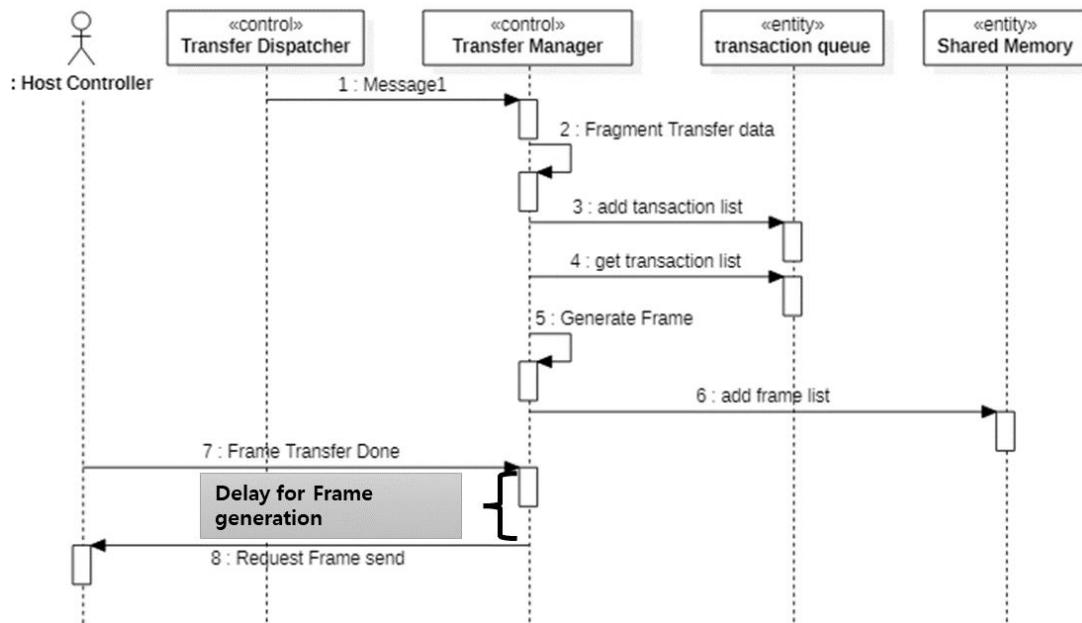


그림 78 Pre-generation 방식의 Frame 생성 구조

### D3.10. CA\_26. Buffered 방식의 Transaction 생성 구조

<성능 요인 3>을 개선하기 위한 후보 구조이다. Transfer Manager는 Transfer 요청이 전달되면 Transfer data를 Host Controller가 전송 가능한 transaction 단위로 분할하고 이를 임시 Buffer인 Transaction Queue에 저장한다. Transfer Manager는 Host Controller와 공유하는 Shared Buffer를 모니터링 하여 queue에 빈 공간이 있는 경우 Transaction queue에 임시 저장된 transaction을 Shared Buffer에 저장한다. Host Controller에서 Frame 전송 완료 event가 발생하면, Transfer Manager는 별도의 Transaction 생성 과정 없이 다음 Frame 전송을 Host Controller에 요청한다.

Transaction queue에 저장할 수 있는 공간을 일정 크기로 제한하는 경우 Pre-Generation 구조에서 발생하는 Bulk data와 같은 큰 data 전송 과정에서 발생하는 메모리 공간 낭비 문제를 해결할 수 있다. 또한, Transfer Manager는 Frame 전송을 위한 추가적인 transaction 생성 시간을 hiding 할 수 있는 장점도 있다.

다음 그림은 Buffered방식의 Transaction 생성 구조에 대한 시스템 내부 동작 흐름이다.

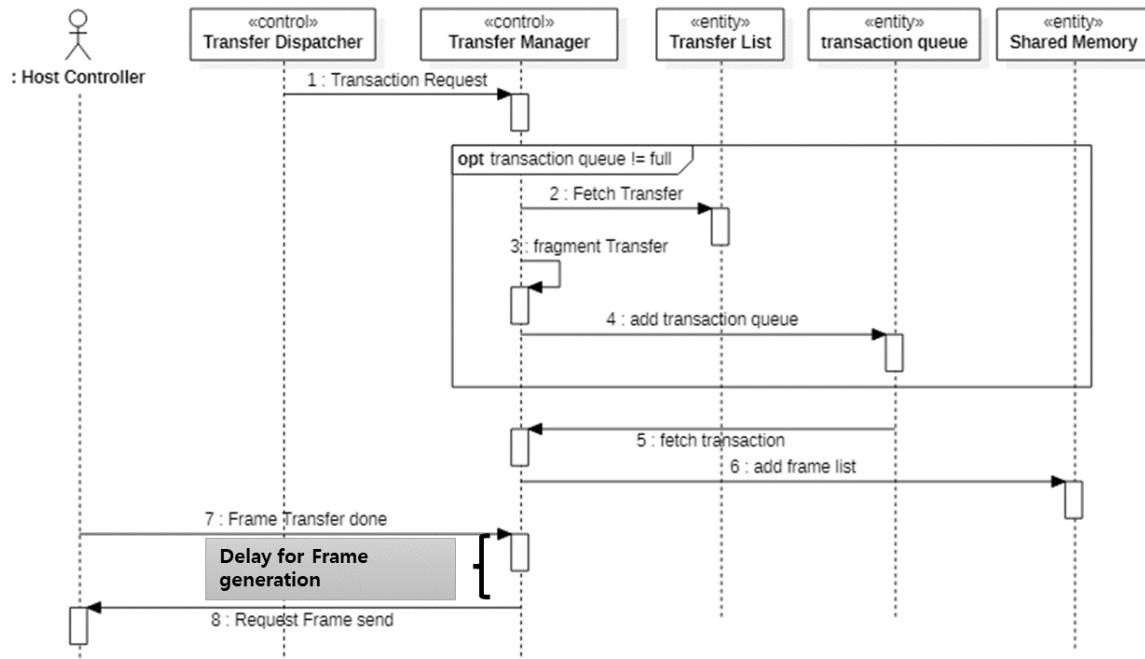


그림 79 Buffered 방식의 Transaction 생성 구조

### D3.11. CA\_27. 단일 List 기반 Transfer 목록 관리 구조

<성능 요인 3>를 개선하기 위한 후보 구조이다. 시스템은 다수의 디바이스가 동시에 전송 요청을 하는 것 지원하기 때문에 전체 Transfer 목록에 추가 삭제하는데 시간을 줄이고, 필요한 경우 전체 전송 목록에서 가장 우선 순위가 높은 Transfer 요청을 빠르게 선택해야 하다.

그림은 Transfer Dispatcher에서 요청된 전체 Transfer를 하나의 list로 통합해서 관리하는 구조이다. Transfer Dispatcher는 Transfer 요청이 추가될 경우 우선 순위에 따라 정렬하여 List에 신규 Transfer를 추가한다. Transfer추가 시 전체 list에 대해서 우선 순위 비교가 필요할 수 있기 때문에 O(n) 만큼의 시간 지연이 발생할 수 있다.

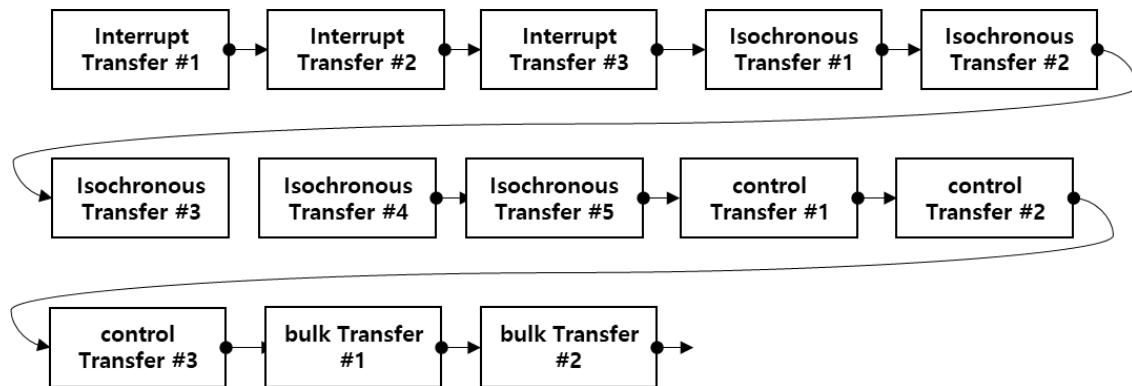


그림 80 List 기반 Transfer 목록 관리

### D3.12. CA\_28. Transfer type별 Transfer목록 관리 구조

<성능 요인 3>를 개선하기 위한 후보 구조이다. USB의 Transfer은 전송 Type별로 우선 순위를 가질 수 있다. 따라서 시스템 내부적으로 전송 type에 대한 우선 순위를 정하고 각 type별로 별도 Transfer list를 관리할 수 있다. 그림은 전송 type별로 목록을 별도로 관리하는 구조에 대한 설명이다. 시스템은 Interrupt → isochronous → control → bulk 순으로 우선 순위를 정하고 각각의 list에는 FIFO형태의 queue로 운영한다. 전송 우선 순위가 개발 시점에 정해져 있기 때문에 Transfer을 추가 삭제시 O(1) 만큼의 시간이 필요하다. 따라서, 데이터 전송 과정에 발생하는 Transfer 목록 관리에 소모되는 시간을 줄일 수 있다.

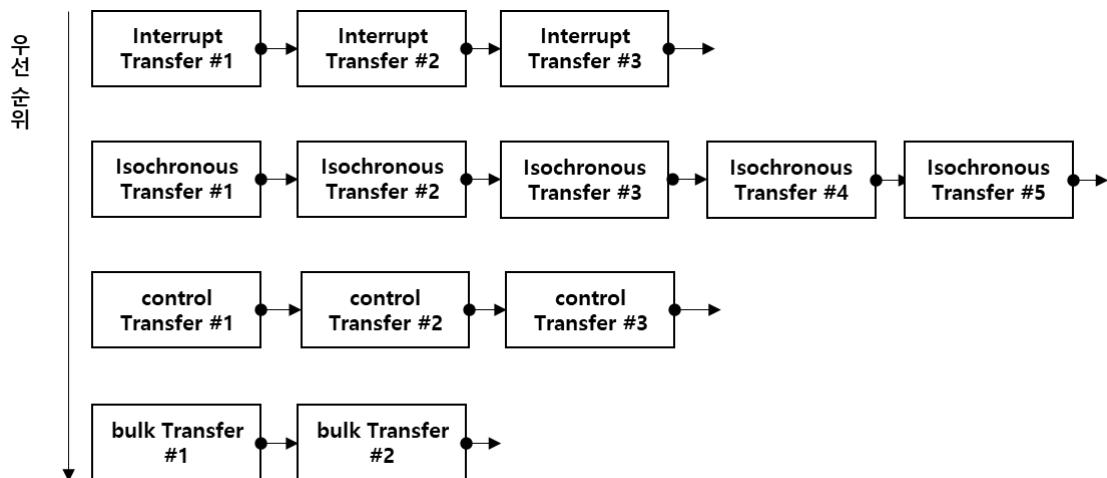


그림 81 Transfer type별 목록 관리

### D3.13. CA\_29. FIFO 기반 Transfer scheduling

<성능 요인 3>을 개선하기 위한 후보 구조이다. Transfer의 type에 따른 우선 순위는 Interrupt → USB Host Driver

isochronous → control → bulk 순으로 우선 순위를 정한다. Transaction type이 동일한 경우 동일한 우선 순위로 처리하여 먼저 요청된 순으로 Transfer 요청을 처리한다.

#### D3.14. CA\_30. Priority 기반 Transfer scheduling

<성능 요인 3>을 개선하기 위한 후보 구조이다. 1차로 Transfer의 type에 따른 우선 순위는 Interrupt → isochronous → control → bulk 순으로 정한다. 동일한 type내에서는 별도의 우선 순위 정책을 적용하여 Transfer의 우선 순위를 정한다. 예를 들어 Power 소모를 줄이기 위한 정책을 적용하는 경우 동일한 type내에서는 Power 소모가 많은 디바이스에 대한 Transfer를 먼저 처리하여 해당 디바이스를 suspend 상태로 빨리 만드는 것이 효율적이다. 따라서 power 소모가 많은 디바이스에 대한 Transfer에 우선 순위를 줄 수 있다. 아래 예의 경우 전류 소모가 많은 B 디바이스에 대한 Transfer를 먼저 처리하는 것이 효율적이다.

A 디바이스 Transfer: <예상 전송 시간: 5초>, <초당 전류 소모 10mA>

B 디바이스 Transfer: <예상 전송 시간: 5초>, <초당 전류 소모 20mA>

A Transfer을 먼저 처리하는 경우:  $5(A \text{ 디바이스 동작 시간}) * 10\text{ms} + 10(B \text{ 디바이스 동작시간}) * 20\text{mA} = 250\text{mA}$

B Transfer을 먼저 처리하는 경우:  $10(A \text{ 디바이스 동작 시간}) * 10\text{ms} + 5(B \text{ 디바이스 동작시간}) * 20\text{mA} = 200\text{mA}$

#### D3.15. CA\_31. 단일 Message Queue 기반 컴포넌트간 메시지 전달 구조

<성능 요인 2>을 개선하기 위한 후보 구조이다. 시스템 내부에서 각 컴포넌트 간에 처리되는 메시지의 우선 순위를 관리하는 별도의 컴포넌트를 통해 우선 순위를 관리하는 구조이다. 시스템을 구성하는 각 컴포넌트는 전체 메시지를 처리하는 컴포넌트에 메시지를 전달하고, 시스템 작업의 우선 순위에 따라 처리할 메시지를 선택하고 해당 메시지를 처리할 컴포넌트에 메시지 처리를 요청한다. 이 경우 시스템 내부의 전체 동작 중 디바이스 입력과 관련된 메시지의 우선 순위를 높일 수 있어 디바이스 입력에 대한 반응 시간을 개선할 수 있다. 반면 전체 Message가 하나의 컴포넌트로 집중되는 점에 따른 오버헤드도 발생할 수 있어 이에 대한 고려도 필요하다.

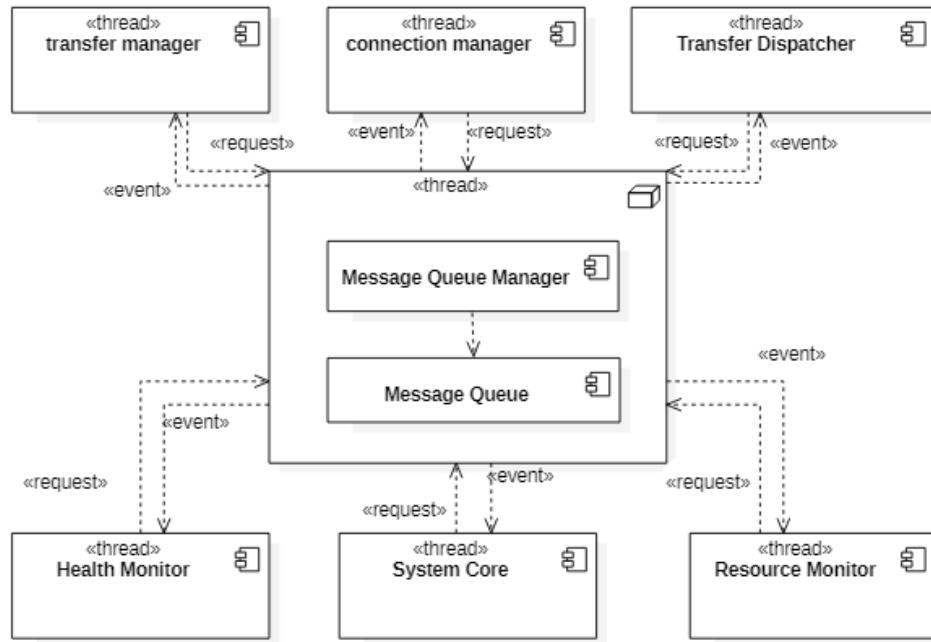


그림 82 단일 메시지 큐를 이용한 시스템 내부 메시지 교환 구조

### D3.16. CA\_32. 컴포넌트별 Message Queue 관리 구조

<성능 요인 2>을 개선하기 위한 후보 구조이다. 단일 메시지 큐와 달리 시스템 내부의 각 컴포넌트는 독립된 메시지 큐를 가지고 있으며 각 메시지 큐에 저장된 메시지를 우선 순위에 따라 처리한다. 시스템 내부의 다양한 메시지 중 데이터 전송과 관련된 메시지의 우선 순위를 높여 처리하여 데이터 전송을 위해 시스템 내부적으로 발생하는 시간 지연을 최소화할 수 있다.

그림에서 각 컴포넌트가 독립된 메시지 큐를 가지고 있으며 새로운 요청이 전달될 경우 메시지 큐를 우선 순위에 따라 정렬되어 저장된다. 해당 컴포넌트는 메시지 큐에서 하나씩 처리하고, 처리 결과에 따라 필요한 요청을 다른 컴포넌트의 메시지 큐에 전달한다. 각각의 컴포넌트는 다른 컴포넌트와 독립적으로 실행될 수 있기 때문에 시스템에 다수의 요청이 발생시 병렬 처리를 통해 시스템 내부적인 시간 지연을 최소화할 수 있고, 데이터 전송에 필요한 메시지의 우선 순위를 높이 경우 데이터 전송 시간 지연을 추가적으로 줄일 수 있다. 아래 그림의 경우 데이터 전송과 관련된 요청보다 Health Manager에서 전달된 요청은 낮은 우선 순위로 처리한다.

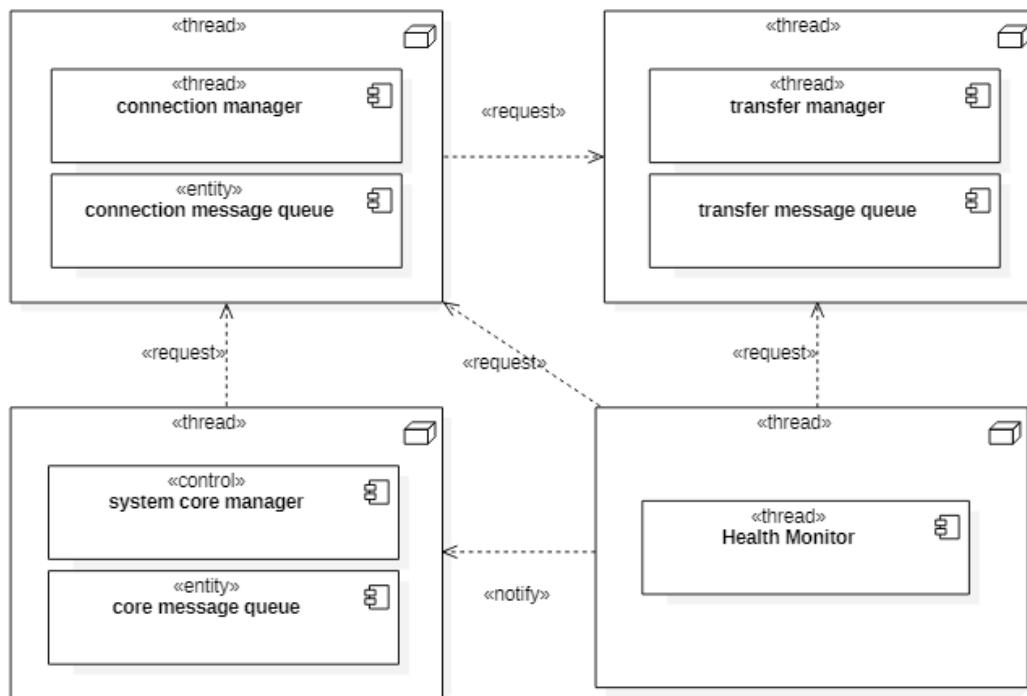


그림 83 독립된 메시지 큐 기반 시스템 내부 메시지 교환 구조

### D3.17. CA\_33. 메모리 참조 기반의 전송 데이터 전달 구조

<성능 요인 2>를 개선하기 위한 후보 구조이다. Application에서 전달된 데이터는 시스템 내부의 여러 컴포넌트를 거쳐 최종적으로 Host Controller에 frame 형태로 전달된다. 이 과정에서 Application에서 전달된 데이터를 복사하지 않고 참조 형태로만 전달하는 구조이다. 데이터를 참조 형태로만 전달할 경우 메모리 복사에 필요한 시간을 줄일 수 있기 때문에 데이터 전송 과정에 발생하는 시스템 내부의 시간 지연을 최소화할 수 있는 장점이 있다.

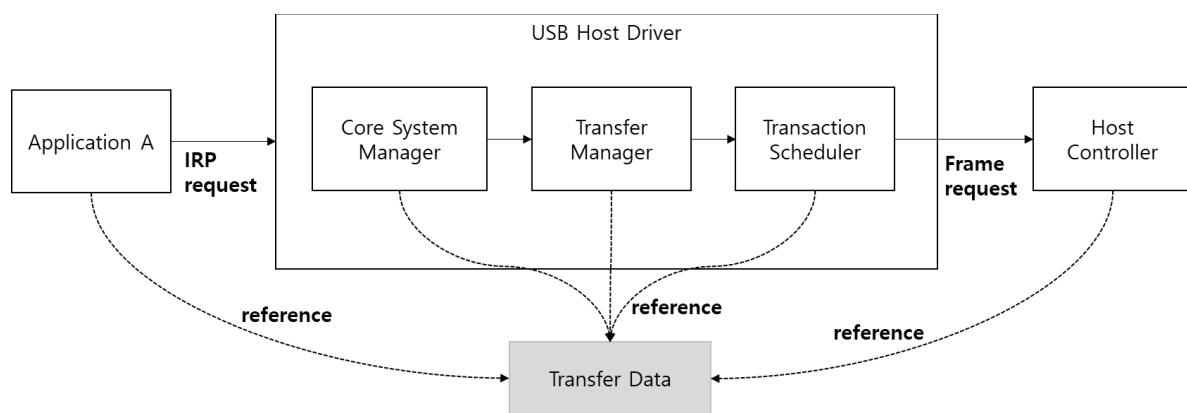


그림 84 메모리 참조 기반의 데이터 전달 구조

### D3.18. CA\_34. Tree 기반의 Configuration 정보 관리 구조

<성능 요인 4>를 개선하기 위한 후보 구조이다. USB 버스 상에서 특정 디바이스가 Idle 상태로 변경된 경우 해당 디바이스에 현재 할당된 자원을 빠르게 회수하기 위해 디바이스에서 자원이 할당된 endpoint를 빠르게 검색할 수 있어야 한다. 이를 위해 시스템 내부적으로 디바이스에 대한 Configuration과 endpoint 매핑 정보를 Tree 형태로 관리하여 특정 디바이스의 endpoint에 할당된 정보를 4단계 만에 확인 할 수 있다.

아래 그림은 Tree 형태로 구성된 각 디바이스에 대한 configuration 정보 관리 구조이다.

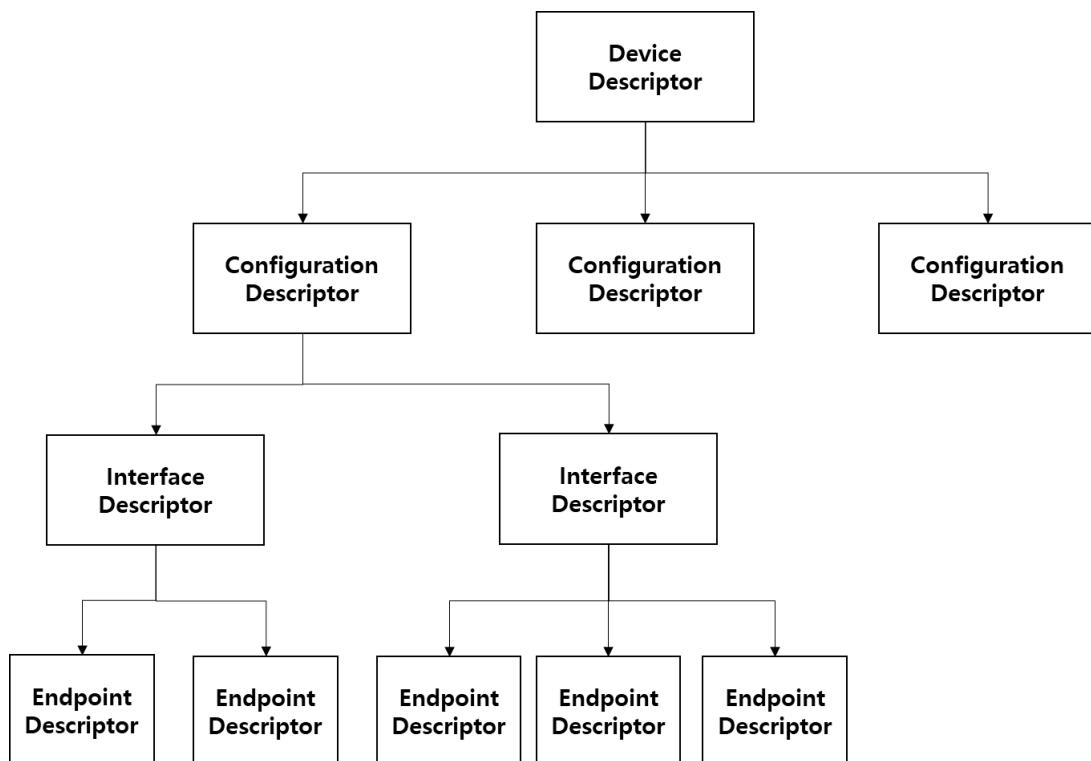


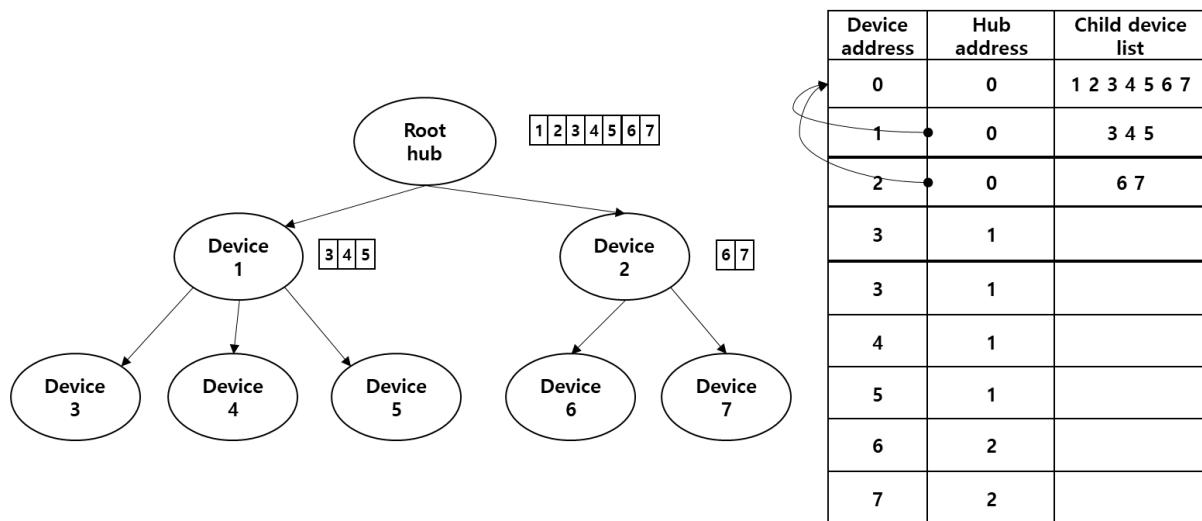
그림 85 Tree 기반 디바이스 Configuration 관리 구조

### D3.19. CA\_35. Tree 기반의 BUS topology 정보 관리 구조

<성능 요인 4>를 개선하기 위한 후보 구조이다. BUS 상에서 HUB로 동작하는 디바이스가 idle 상태로 변경된 경우 HUB에 포함된 전체 device의 자원을 회수하기 위해 HUB에 포함된 전체 디바이스에 대한 확인이 필요하다. 따라서 시스템에서 HUB에 포함된 전체 디바이스의 검색 시간을 줄이는 경우 HUB에 포함된 전체 자원에 대한 회수 시간을 줄일 수 있어 효율적인 자원 활용이 가능하다.

다음 그림은 HUB에 포함된 디바이스를 Tree 형태로 관리하는 구조이다. 시스템은 전체 디바이스를 배

열 형태로 관리하면 각 디바이스 번호를 배열의 index로 사용한다. 배열은 각 디바이스 별로 <device ID, HUB address, Child device list>를 포함하고 있다. HUB address는 디바이스가 연결된 HUB에 대한 Address(device ID)이며, Child device list는 디바이스가 HUB인 경우 HUB에 연결된 전체 디바이스의 목록을 저장하고 있다. 시스템에 연결될 수 있는 디바이스의 최대 수는 127개를 넘지 않기 때문에 시스템은 최대 127개의 배열을 만들어서 전체 디바이스 목록을 관리한다. 시스템에 디바이스가 새로 추가되는 경우 시스템은 해당 디바이스가 연결된 HUB에 해당 정보를 업데이트 하고, Root HUB까지 Tree를 따라가면서 상위 HUB에 연결된 디바이스 정보를 업데이트 한다. 특정 HUB가 idle 상태로 변경되는 경우 시스템은 해당 HUB의 디바이스 ID를 index로 하여 배열을 검색하고, 해당 HUB에 포함된 device 목록을 확인할 수 있기 때문에 O(1)의 시간 복잡도로 검색이 가능하다.



&lt;USB Topology의 논리적 구성&gt;

&lt;USB Topology의 자료 구조&gt;

그림 86 Tree 기반 USB BUS topology 관리 구조

### D3.20. CA\_36. 데이터 Cache 관리 구조

<성능 요인 2>를 개선하기 위한 후보 구조이다. 각 endpoint에 대한 데이터 Cache를 운영하여 디바이스로부터 미리 data를 read하여 Cache에 저장할 수 있다. 이 경우 Application에서 IRP를 전달받았을 때 IRP → Transfer 정보 → Transaction list → Frame 정보를 변환하는 과정과 디바이스로부터 Data를 Read해 오는 과정을 생략할 수 있어 이 과정의 시간 지연을 줄일 수 있다. 하지만, Bulk 전송 방식과 같이 대량의 data를 Read하는 경우 디바이스로부터 데이터를 Read하여 cache를 채우는 속도가 bottleneck이 되기 때문에 Cache로 인한 이득이 없어지는 단점이 있다.

#### D4. QA\_02 디바이스의 연결/해제시 인식 지연 시간

USB 시스템은 제한된 자원을 다수의 디바이스가 공유하여 사용하기 때문에 물리적인 제약 사항이 존재한다. 따라서 제한된 환경에서 디바이스의 빠른 인식을 위해서는 디바이스 이 SW적인 오버헤드를 줄이는 방안 등에 대해서 성능 측면에서 검토가 필요하다. 이를 위해 먼저 시스템의 디바이스 인식 과정을 분석하고 각 과정에서 SW 오버헤드를 시간을 줄일 수 있는 방안들을 후보 구조로 설계하였다. 다음은 디바이스 인식 속도에 영향을 주는 요인과, 각 성능 요인을 개선하기 위한 후보 구조 분석을 요약하여 도식화한 것이다.

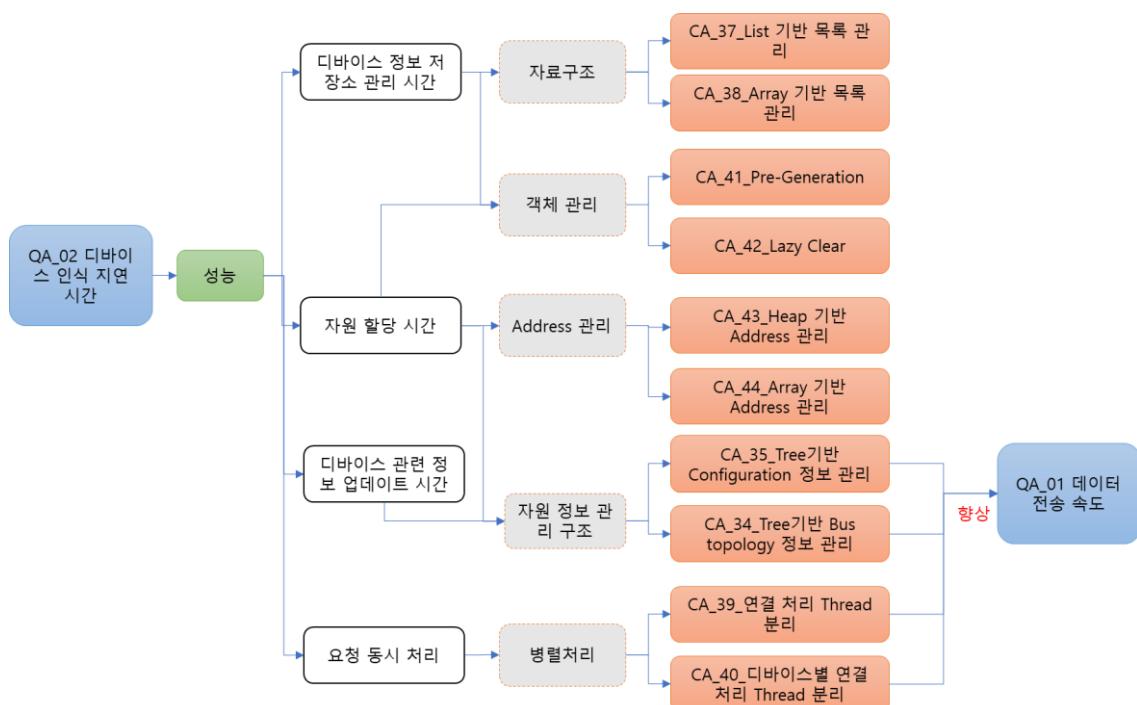


그림 87 디바이스 인식 시간에 대한 후보 구조 분석

시스템에서 디바이스 연결 해제를 처리하는 일반적인 과정은 다음과 같다.

1. Host Controller가 시스템에 디바이스 연결 상태 변화를 통보.
  2. 시스템은 디바이스에 Reset 명령을 전달하고 디바이스 Description을 요청
  3. 시스템은 디바이스에 Address를 할당하여 디바이스에 전달
  4. 시스템은 디바이스의 Configuration에 필요한 리소스(Bandwidth, Power 등)를 확인하여 할당
  5. 시스템은 추가된 디바이스 정보를 디바이스 정보 저장소, BUS Topology 저장소 등에 업데이트

6. 시스템은 추가된 디바이스를 사용할 Application을 검색하여 디바이스 추가를 통보

위 과정에서 디바이스 연결 해제 시간 지연에 영향을 주는 요인은 다음과 같다.

- **성능 요인 1:** 디바이스 정보 저장소에서 추가/삭제하는데 필요한 시간
  - 새로운 디바이스가 연결되는 경우 디바이스 정보 저장소에 추가하는데 필요한 시간을 최소화 해야 한다.
  - 디바이스가 분리 되는 경우 디바이스 정보 저장소에서 분리된 디바이스를 검색하여 정보를 삭제하는 필요한 시간을 최소화 해야 한다.
- **성능 요인 2:** 디바이스에 필요한 자원 할당/해제 시간
  - 디바이스에 대한 Address 할당/해제 시간을 최소화 해야 한다.
  - 디바이스에 필요한 자원(Bandwidth, Power 등)을 할당/해제 하는 시간을 최소화 해야 한다.
- **성능 요인 3:** 시스템 내의 디바이스와 관련된 정보를 업데이트하는데 필요한 시간
  - BUS Topology에 추가된 디바이스의 정보를 추가/삭제하는데 필요한 시간을 최소화 해야 한다.
- **성능 요인 4:** 시스템에 동시에 여러 디바이스가 접속된 경우에 대한 처리 시간
  - 시스템에서 데이터 전송과 새로운 디바이스 연결을 동시에 처리해야 하는 경우 데이터 전송으로 인해 새로운 디바이스 연결이 지연되는 시간은 최소화 해야 한다.

<성능 요인 1>을 개선하기 위해서는 디바이스 정보를 저장하고 검색하는데 효율적인 자료구조 및 알고리즘 검토가 필요하다. <성능 요인 2>를 개선하기 위한 후보 구조는 "D1. NFR\_01. 디바이스 연결 성공 비율"에 포함된 내용을 일부 포함하며, 본 장에서는 추가적인 성능 개선 요인에 대해서만 다룬다. <성능 요인 3>에 대한 후보 구조는 "D3. QA\_01 데이터 전송 속도"의 "CA\_29. Tree 기반의 BUS topology 정보 관리 구조"와 동일하다.

상기 설명된 성능 영향 요인을 바탕으로 도출된 후보 구조는 다음과 같다.

#### D4.1. CA\_37. List 기반 디바이스 목록 관리

<성능 요인 1>을 개선하기 위한 후보 구조이다. 시스템은 디바이스 정보를 List 형태의 자료 구조로 관리한다. 시스템에 디바이스가 추가 되는 경우 디바이스 정보를 저장하기 위한 메모리 공간을 할당하여 디바이스 정보 List에 추가한다. 시스템에서 디바이스가 분리 되는 경우 List에서 디바이스 정보를 제거하고 할당된 메모리를 해제한다. 디바이스가 HUB인 경우 HUB에 연결된 전체 디바이스를 검색하여 연결을 해제한다.

#### D4.2. CA\_38. Array 기반 디바이스 목록 관리

<성능 요인 1>을 개선하기 위한 후보 구조이다. 시스템은 디바이스 정보를 Array 형태의 자료 구조로 관리한다. 시스템에 연결될 수 있는 디바이스의 최대 개수는 127개 이기 때문에 127개의 Entry를 가지는 배열을 생성하여 디바이스 정보를 생성한다. 디바이스의 Address 정보(디바이스 ID)를 1~127 까지 할당하는 경우 해당 값을 배열의 Index로 사용할 수 있다. 즉, 디바이스 추가 삭제에 필요한 시간 복잡도를 O(1)로 줄일 수 있다.

#### D4.3. 디바이스 연결 처리를 위한 독립 Thread 구조

<성능 요인 1>, <성능 요인 4>을 개선하기 위한 후보 구조이다. 시스템은 동시에 여러 개의 디바이스가 연결되어 동작하는 지원한다. 따라서 이미 연결된 디바이스를 통해 데이터를 전송 중인 상태에서 새로운 디바이스가 연결 되는 경우 데이터 전송 처리와 디바이스 연결 처리를 동시에 진행 해야 한다. 시스템 내부적으로 두 개의 처리 기능을 분리하여 병렬로 처리하여 시간 지연을 줄일 수 있다.

다음 그림은 두개의 처리 기능을 별도로 분리하여 병렬로 처리하는 구조이다. 시스템에 새로운 Event 가 발생하면 Event를 처리하는 Main는 요청된 Event에 따라 데이터 전송 처리 컴포넌트 혹은 디바이스 연결 처리 컴포넌트에 요청을 전달한다. 디바이스 연결 요청은 연결 처리 컴포넌트가 관리하는 큐에 저장되어 순차적으로 처리된다.

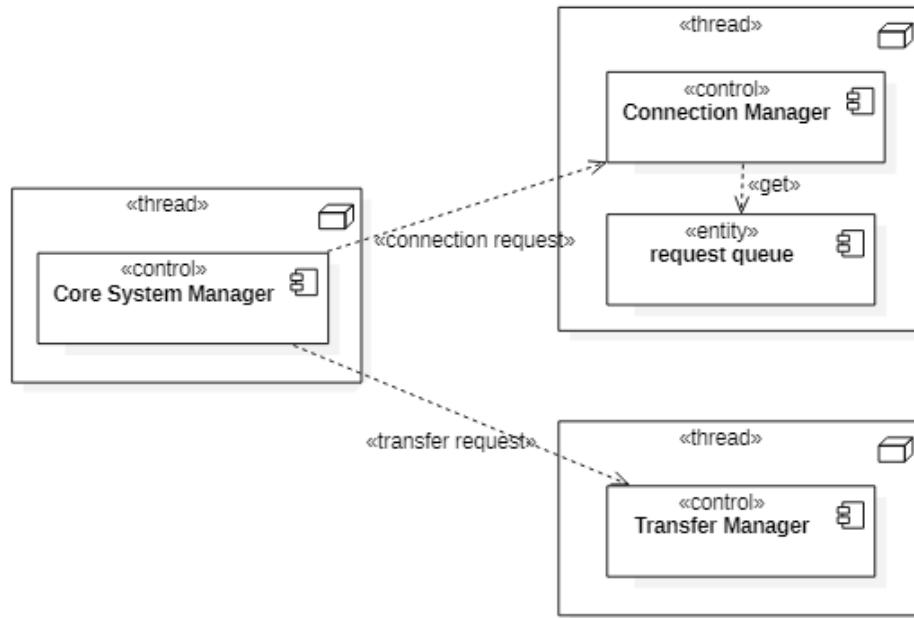


그림 88 디바이스 연결 처리를 위한 Thread 분리 구조

#### D4.4. CA\_40. 디바이스 연결 요청 별 독립된 Thread 구조

<성능 요인 1>, <성능 요인 4>을 개선하기 위한 후보 구조이다. 시스템은 동시에 여러 개의 디바이스가 연결되는 경우 각각의 디바이스 연결을 처리하기 위한 독립된 Thread를 생성하여 처리를 위임한다. 각각의 디바이스 연결 처리가 병렬로 처리될 수 있으며 연결 처리를 위한 별도의 대기 큐가 존재하지 않는다. 즉, 디바이스 연결이 필요한 경우 시스템은 디바이스 연결을 처리할 별도의 Thread를 생성하고 디바이스 연결 처리 과정을 해당 Thread를 통해 처리한다.

다음 그림은 연결 요청 별 독립된 Thread를 가지는 구조이다.

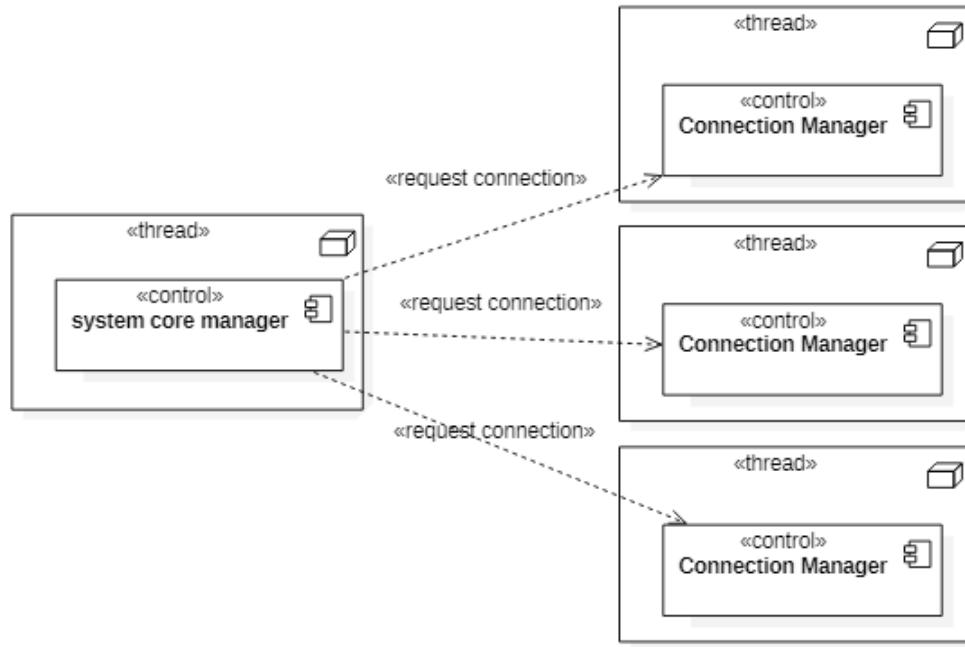


그림 89 연결 요청 별 독립된 Thread 실행 구조

#### D4.5. CA\_41. 연결 관리 객체의 Pre-Generation

<성능 요인 1>, <성능 요인 2>, <성능 요인 3>을 개선하기 위한 구조이다. 시스템에 새로운 디바이스가 연결되는 경우 시스템은 해당 디바이스를 관리하기 위한 객체를 생성해야 한다. 시스템은 하나의 디바이스에 대해 <device address, device configuration, application mapping, device resource> 등의 여러가지 정보를 관리하는 객체를 생성해야 하기 때문에 객체 생성에 필요한 시간 지연도 디바이스 연결 시간에 영향을 줄 수 있다. 시스템이 연결 관리에 필요한 객체를 미리 생성해 두고 새로운 연결 요청 발생시 이용하는 구조를 적용할 경우 객체 생성에 필요한 시간을 hiding 할 수 있어 디바이스 연결 시간을 줄일 수 있다.

#### D4.6. CA\_42. 연결 관리 객체의 Lazy Clear

<성능 요인 1>, <성능 요인 2>, <성능 요인 3>을 개선하기 위한 구조이다. 시스템에서 디바이스가 분리되는 경우 시스템은 해당 디바이스와 관련된 객체를 삭제해야 한다. 하지만 동일한 디바이스의 분리가 빈번하게 발생하는 경우 매번 디바이스를 위한 객체를 생성하고 삭제하는 것은 비효율적이다. 시스템에서 디바이스가 분리된 상황에서 일정 시간 동안 해당 디바이스의 정보를 유지하는 경우 동일한 디바이스가 다시 연결될 때 객체 생성에 필요한 시간과 디바이스 정보 설정에 필요한 시간을 줄일 수 있다.

#### D4.7. CA\_43. 가용 Address 목록 관리 구조(Heap)

<성능 요인 2>를 개선하기 위한 구조이다. 시스템에 새로운 디바이스가 연결되는 경우 시스템 가용 Address 목록에서 해당 디바이스에 사용할 Address를 하나 선택하여 해당 디바이스에 할당 한다. Address 할당에 필요한 시간을 줄이기 위해 시스템 내부적으로 별도의 가용 Address 목록을 관리할 경우 Address 할당에 대한 시간 복잡도를 O(1)로 줄일 수 있다. Heap을 이용하여 가용 Address 목록을 관리할 경우 가용한 Address를 순차적으로 사용할 수 있지만, Heap을 통한 정렬에 필요한 시간이 들어날 수 있다.

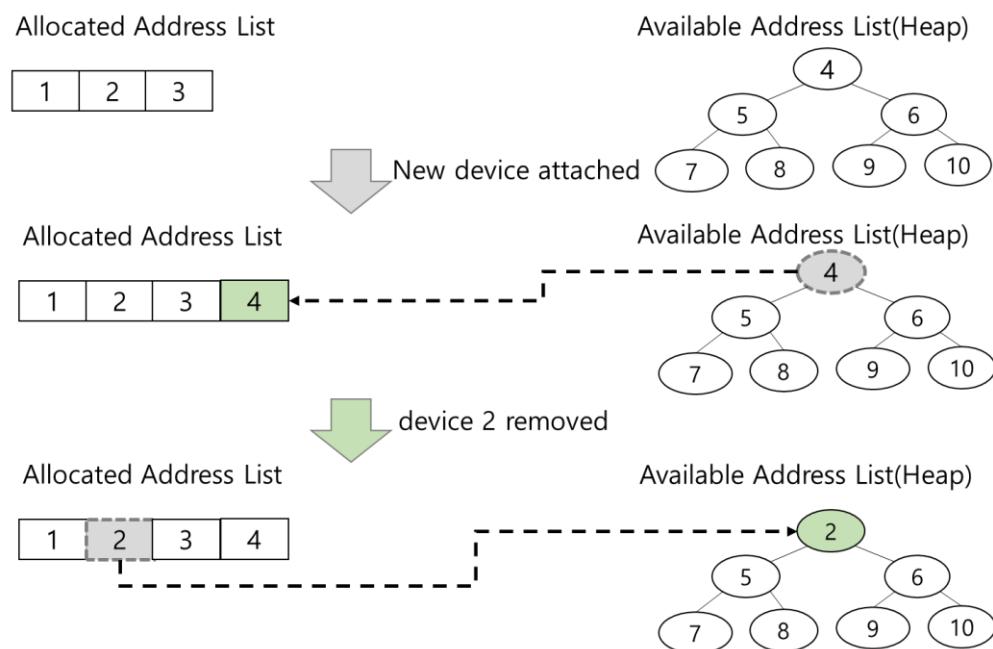


그림 90 Heap 기반 가용 Address 관리 구조

#### D4.8. CA\_44. 가용 Address 목록 관리 구조(Array)

<성능 요인 2>를 개선하기 위한 구조이다. 가용한 Address 목록을 FIFO 형태의 Array로 관리하는 방식이다. FIFO 형태로 관리되기 때문에 디바이스 Address가 순차적으로 정렬되어 사용되지는 않지만 Address 할당에 필요한 시간 복잡도를 O(1)로 줄일 수 있는 장점이 있다.

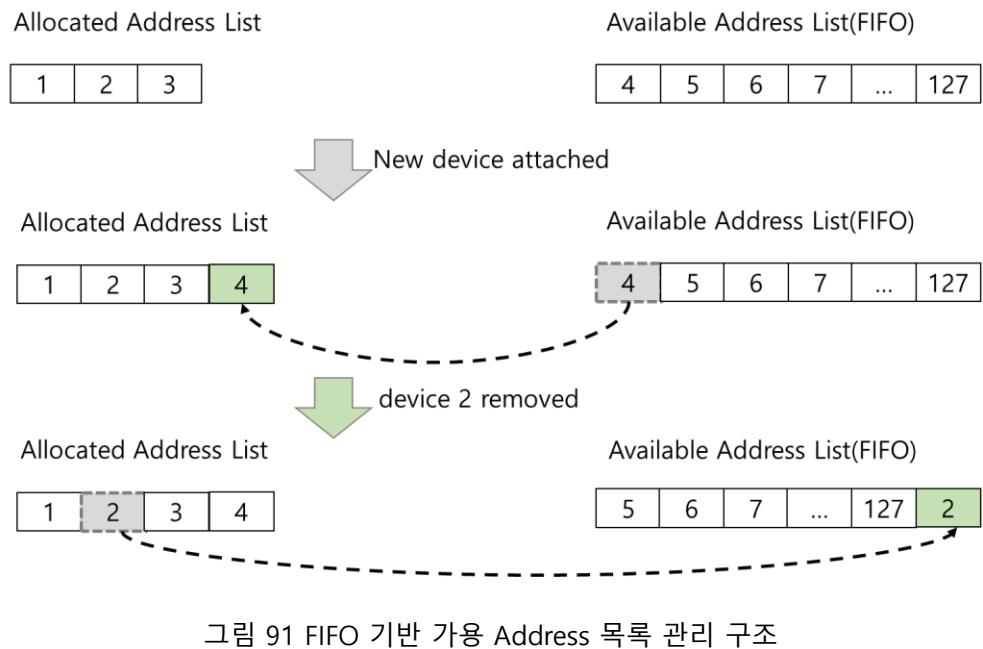


그림 91 FIFO 기반 가용 Address 목록 관리 구조

## D5. QA\_03 동시에 연결할 수 있는 디바이스의 수

본 과제에서 대상으로 하는 모바일 장치의 경우 사용자가 다수의 USB 장치를 시스템에 동시에 연결하여 사용할 수 있으며, HUB를 이용하여 디바이스 연결을 추가적으로 확장하여 사용할 수 있다. 따라서, 시스템은 동시에 여러 개의 장치 연결을 지원해야 하며, 동시에 지원되는 장치의 수가 많을수록 좋다. 다음은 시스템이 동시에 지원할 수 있는 디바이스의 수에 영향을 주는 후보 구조 분석을 요약하여 도식화한 것이다.

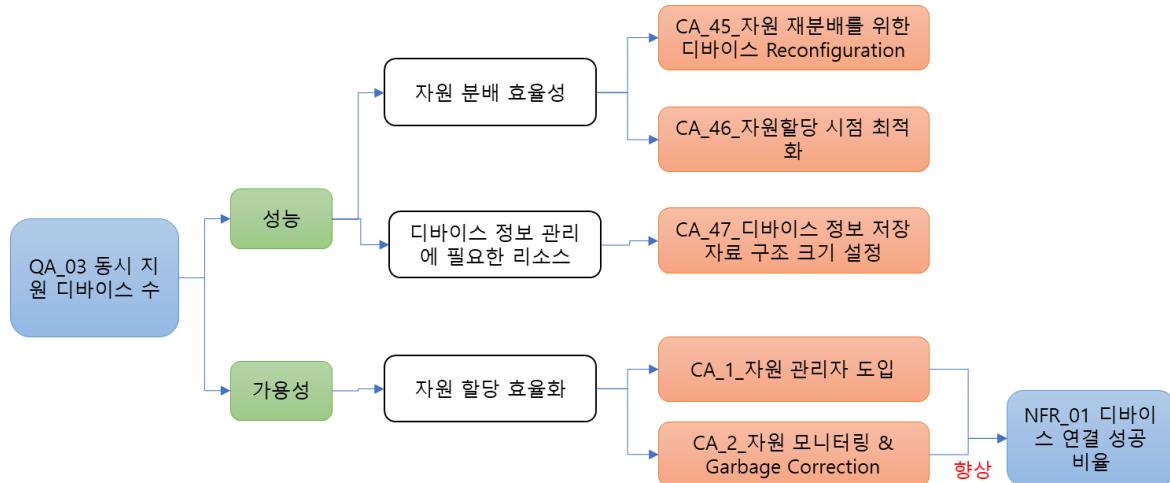


그림 92 동시 지원 디바이스 수 증가를 위한 후보 구조 분석

시스템에서 지원하는 시스템에서 디바이스가 연결/분리되는 과정은 “[QA\\_02 디바이스의 연결/해제시 인식 지역 시간](#)”에 설명된 과정과 동일하다. 이 과정에서 동시에 지원할 수 있는 디바이스 수에 영향을 주는 요인은 다음과 같다. USB 시스템은 지원되는 전체 자원의 수가 제한되어 있기 때문에, 본 과제에서는 제한된 자원 범위 내에서 동시에 지원하는 디바이스의 수를 늘릴 수 있는 요인에 대해서만 고려한다.

- **성능 요인 1:** 낭비되는 자원(Bandwidth, Power 등) 방지
  - 디바이스 자원이 연결 해제 과정에서 낭비되지(Leak) 않도록 효율적인 관리가 필요하다.
- **성능 요인 2:** 디바이스에 대한 자원 배분의 효율성
  - 유휴 상태인 디바이스에 대해서는 자원을 빠르게 회수하여 가용 자원으로 재 할당해야 한다.
  - 각 디바이스에 할당된 자원을 허용 범위 내에서 줄일 수 있는 경우 디바이스에 할당된 자원을 재 분배하여 새로운 디바이스 연결을 지원할 수 있다.
- **성능 요인 3:** 디바이스 정보 저장 자료 구조의 크기
  - USB Spec에서는 최대 127개의 디바이스를 지원할 수 있는 것을 명시하고 있기 때문에 본 시스템에서도 최대 127개의 디바이스를 동시에 지원할 수 있는 자료 구조를 고려해야 한다.

<성능 요인 1>은 시스템 내부적으로 USB 시스템 자원에 대한 관리 오류를 제거하는 것과 동일하다. 따라서 "NFR\_01 디바이스 연결 성공 비율"에 대한 후보 구조에서 다루었던 <오류 요인 1> 시스템에서 디바이스에 필요한 자원(Bandwidth, Power) 할당 실패에 대한 후보 구조와 동일하다. <성능 요인 2>에 대한 후보 구조 중 유휴 상태의 디바이스 자원을 빠르게 회수하기 위한 후보 구조는 "QA\_01 데이터 전송 속도"에 대한 후보 구조에서 다루었기 때문에 본 절에서는 디바이스 별 자원 재분배 구조에 대해서만 설명한다.

상기 설명된 성능 영향 요인을 바탕으로 도출된 후보 구조는 다음과 같다.

#### D5.1. CA\_45. 디바이스 Reconfiguration을 통한 자원 재 분배 구조

<성능 요인 2>를 개선하기 위한 후보 구조이다. USB 시스템은 가용한 자원이 한정되어 있기 때문에 시스템에서 동시에 많은 수의 디바이스를 지원하기 위해서는 가용한 자원을 디바이스별로 효율적으로 분배하여야 한다. 하지만 데이터 전송 효율성을 위해 이미 연결된 디바이스에 가용한 자원을 모두 할당한 상황에서는 새로운 디바이스에 필요한 자원을 할당하지 못하는 경우가 발생할 수 있다. 따라서, 이러한 상황에서 새로운 디바이스 연결을 지원하기 위해 기존에 연결된 디바이스에 할당된 자원을 재분배하는 구조를 고려할 수 있다.

다음 그림은 디바이스 Reconfiguration을 통한 자원 재분배 과정에 대한 설명이다. 시스템은 새로운 디바이스가 연결된 경우 디바이스에 필요한 자원을 확인한다. 디바이스에 필요한 자원이 부족한 경우 현재 연결된 디바이스를 사용중인 Application에 자원의 사용량을 낮출 것을 요청한다. Application에서 자원 사용량을 낮출 수 있는 경우, Application은 디바이스에 대한 Reconfiguration을 시스템에 요청하고, 시스템은 Reconfiguration을 통해 확보한 자원을 새로운 디바이스에 할당한다.

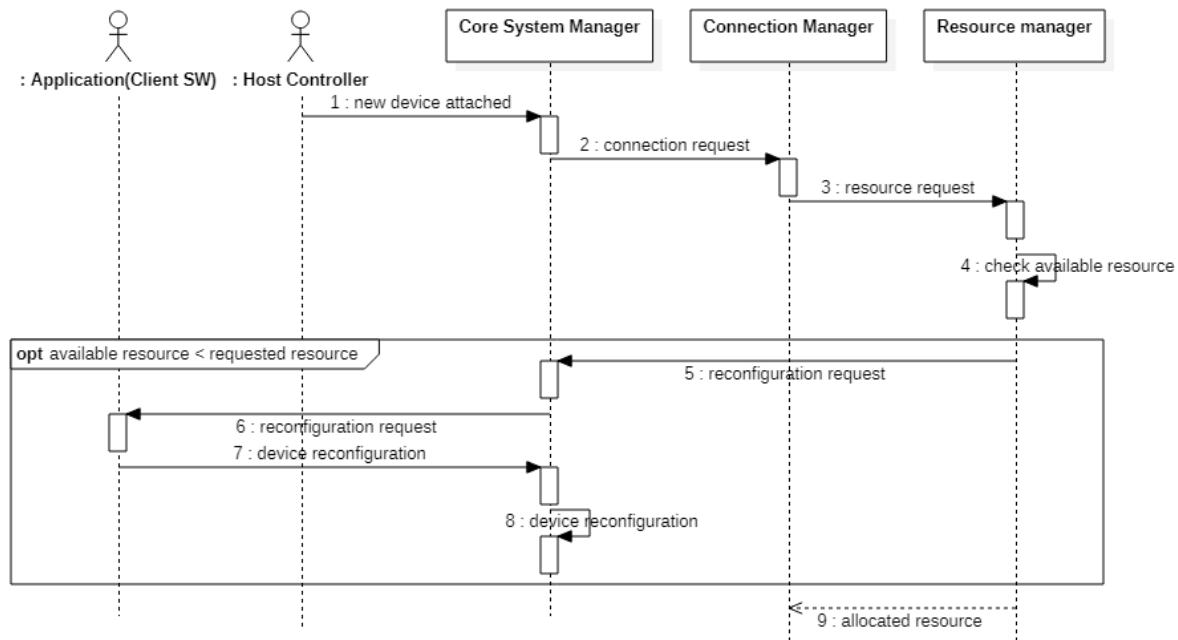


그림 93 USB 자원 재 분배를 위한 디바이스 Reconfiguration Sequence

#### D5.2. CA\_46. Delayed Resource Allocation 구조

<성능 요인 2>를 개선하기 위한 후보 구조이다. 시스템에서 디바이스에 자원(Bandwidth)을 할당하는 시점을 디바이스 연결 시점이 아닌 디바이스와 통신하는 시점으로 지연시키는 구조이다. 시스템이 디바이스와 통신하는 과정은 일반적으로 “디바이스 인식” → “디바이스 연결(Enumeration)” → “디바이스 Idle” → “디바이스와 통신”的 과정으로 진행된다. 따라서, 디바이스 연결 시점부터 자원을 할당할 경우 “디바이스 연결 시점”부터 “디바이스 통신” 시점 사이에는 할당된 자원이 낭비되는 구간이 발생한다. 만약 디바이스 구현(혹은 디바이스 동작 오류)에 따라 디바이스 연결 후 디바이스 통신까지 시간이 지연되는 경우 자원이 낭비는 더 커질 수 있다. 따라서, 디바이스에 대한 자원 할당 시점을 디바이스와 통신하는 시점으로 지연시킬 경우 불필요한 자원 낭비를 막을 수 있으며, 디바이스 연결 시점에 자원을 할당하지 않기 때문에 동시에 연결 가능한 디바이스의 수도 늘릴 수 있다.

#### D5.3. CA\_47. 디바이스 정보 저장 자료 구조의 크기

<성능 요인 3>을 개선하기 위한 후보 구조이다. 본 과제에서 대상으로 하는 모바일 장치는 시스템에 직업 연결되는 장치와 HUB를 통해 확장 연결되는 장치를 모두 지원한다. 따라서 일반적으로 확장되는 장치의 수에 제한이 없기 때문에 USB Spec에서 정의한 최대 디바이스 연결 수인 127개 디바이스가 연결될 수 있는 환경을 가정한다. 이를 위해 시스템 내부적으로 관리되는 entity(Device Info table, Configuration Table 등)는 최대 127개를 동시에 지원할 수 있도록 고려되어야 한다.

## D6. QA\_04 새로운 디바이스 클래스 추가 용이성

본 과제에서 대상으로 하는 모바일 장치는 기본적으로 Mass Storage와 HID 클래스를 지원하고 있지만, 이외에도 다양한 USB 장치가 연결되는 상황을 고려하여 새로운 디바이스 클래스가 추가되는 경우에 대한 고려가 필요하다. 다음은 디바이스 클래스 추가 용이성에 영향을 주는 후보 구조 분석을 요약하여 도식화한 것이다.

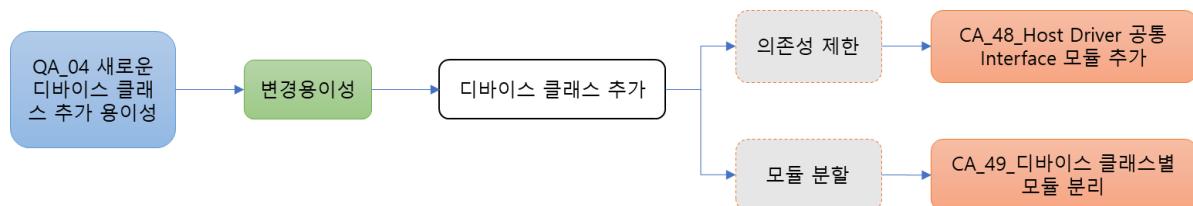


그림 94 디바이스 클래스 추가 용이성 향상을 위한 후보 구조 분석

새로운 디바이스가 추가될 경우에 발생할 수 있는 변경 요인은 다음과 같다.

- **변경 요인 1:** 새로운 디바이스 기능 추가
  - 디바이스 클래스 별로 동작하는 방식이 다르기 때문에 신규 디바이스 클래스를 위한 새로운 Interface가 추가될 수 있다.

<변경 요인 1>을 개선하기 위해서는 새로운 디바이스 클래스에서 사용할 수 있는 범용적인 interface 구조에 대한 고려가 필요하다. 또한 새로운 Class 추가로 인해 시스템 내부 구현이 변경되지 않도록 변경 범위를 제한할 수 있는 구조에 대한 검토가 필요하다.

상기 설명된 변경 요인을 바탕으로 도출된 후보 구조는 다음과 같다.

### D6.1. CA\_48. USB Host Driver Interface 모듈 분리

<변경 요인 1>을 개선하기 위한 후보 구조이다. 시스템의 기능에 따라 변경되는 부분과 공통적인 기능 부분으로 분리하고 Application은 Interface Layer를 통해서만 시스템에 접근할 수 있도록 한다. USB Host Driver의 공통적인 기능은 USB Host Driver Layer에 적용하며, 디바이스 별로 변경되어야 하는 부분은 USB Host Driver API에 적용한다. 따라서 공통적인 기능과 외부 Interface 기능을 분리하여 디바이스 클래스 추가에 따라 변경되어야 하는 부분을 Interface 부분으로 제한할 수 있다.

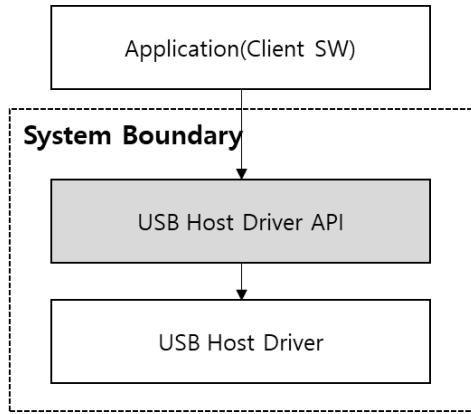


그림 95 USB Host Driver API 분리

#### D6.2. CA\_49. 디바이스 클래스별 Interface 적용

<변경 요인 1>을 개선하기 위한 후보 구조이다. 디바이스 클래스에 필요한 Interface를 별도의 Interface로 분리할 경우 디바이스 추가로 인한 영향 범위를 해당 Interface로 제한할 수 있다. 다음 그림은 디바이스 클래스별 Interface를 분리한 구조이다. 특정 디바이스 클래스를 사용하는 Application은 해당 클래스를 지원하는 Interface를 사용하며, 이외 별도의 클래스로 구분되지 않는 Custom 디바이스의 경우 범용적인 USB Host Driver API를 이용한다. 따라서 디바이스 클래스별 Interface를 분리할 경우 디바이스 클래스에 대한 수정이 발생할 경우 USB Host Driver API 전체를 변경하지 않고 변경 사항을 적용할 수 있다.

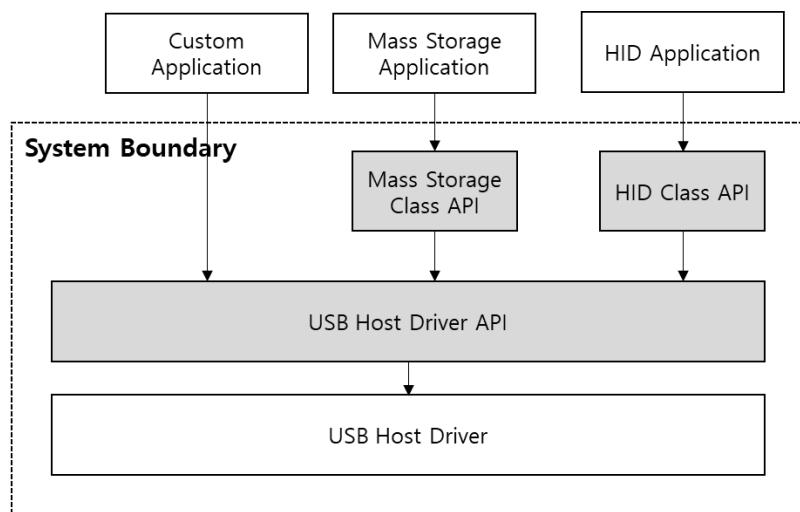


그림 96 디바이스 Class별 Interface 분리 구조

## D7. QA\_05 새로운 실행 환경(HW, OS) 지원 용이성

본 과제에서 대상으로 하는 USB Host Driver는 모바일 장치의 실행 환경(HW, OS등)이 변경된 경우 변경된 실행 환경을 지원하기 위한 추가적인 SW 구현이 필요하다. 실행 환경의 변화에 대한 시스템 구현을 최소화하기 위해서는 외부 실행환경에 영향이 있는 모듈을 분리하고, 외부 의존성을 최소화하기 위한 Interface등에 대한 검토가 필요하다. 다음은 실행환경 변화에 대한 영향 최소화하기 위한 후보 구조 분석을 요약하여 도식화한 것이다.

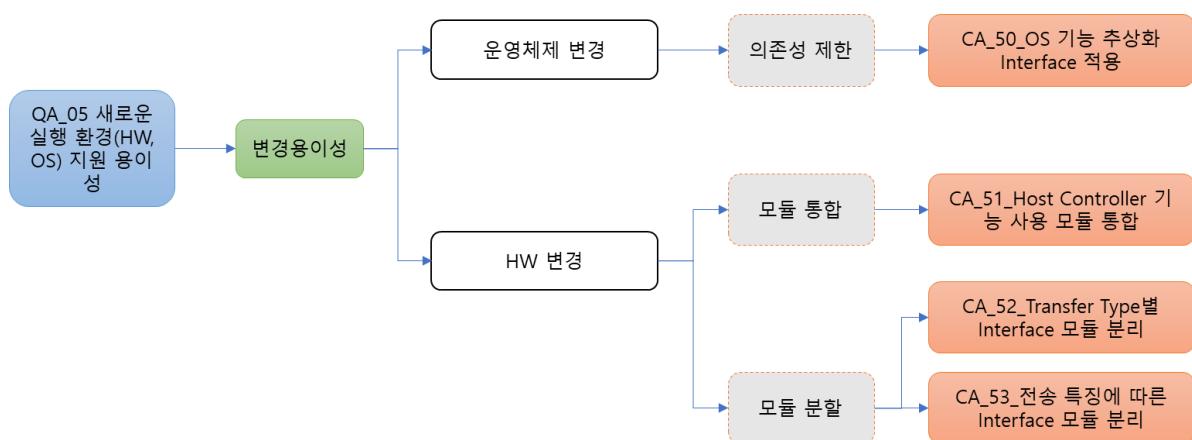


그림 97 실행환경에 대한 변경용이성 후보 구조 분석

실행 환경의 변경 요인은 다음과 같다.

- **변경 요인 1: 운영체제의 변경**
  - 시스템은 운영체제의 Driver 형태로 동작하기 때문에 새로운 운영체제에 적용하거나, 기존 운영체제의 기능이 변경되는 경우 구현이 변경되어야 할 수 있다.
- **변경 요인 2: HW 변경**
  - 기존 Host Controller에 기능이 변경 되거나 새로운 USB Spec 지원을 위해 신규 기능이 추가되는 경우 Host Controller를 관리하기 위한 SW 기능 변경이 필요할 수 있다.

<변경 요인 1>, <변경 요인 2>를 개선하기 위해서는 OS의 기능 및 HW 기능을 추상화 할 수 있는 SW Interface 구조에 대한 검토가 필요하다.

상기 설명된 변경 요인을 바탕으로 도출된 후보 구조는 다음과 같다.

### D7.1. CA\_50. Layer별 OS 기능 추상화 Interface 구조

<변경 요인 1>을 개선하기 위한 후보 구조이다. 시스템에서 사용하는 운영체제와 관련된 Interface를 별도의 모듈 분리하고 시스템에서는 해당 모듈을 이용해서 운영체제의 기능을 사용한다. 운영체제의 기능이 변경되는 경우 해당 모듈로 변경을 제한할 수 있어 변경에 따른 영향을 최소화할 수 있다. 다음 그림은 시스템이 사용하는 OS관련 기능을 별도의 Interface로 분리한 구조이다. 시스템의 전체 모듈은 OS 기능이 필요한 경우 OS Abstraction Interface를 이용하기 때문에 해당 모듈은 Vertical한 구조를 가진다.

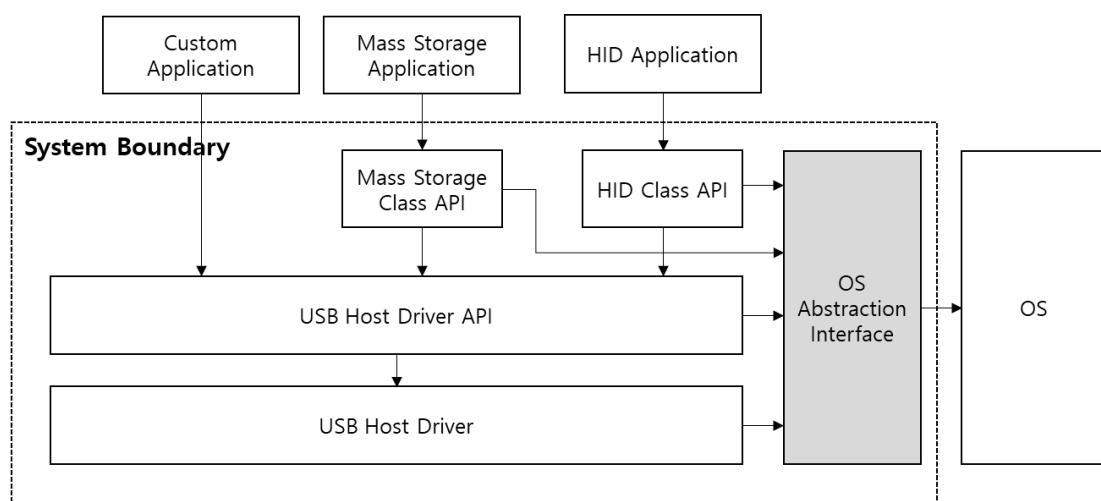


그림 98 OS 기능 추상화 Interface 구조

### D7.2. CA\_51. Host Controller 기능 사용 모듈 통합

<변경 요인 2>를 개선하기 위한 후보 구조이다. 시스템의 기능 중 Host Controller의 기능을 직접 사용하는 모듈들을 통합하여 별도의 패키지로 관리하고 Host Controller를 직접 사용하지 않는 기능들과 분리하여 Host Controller에 대한 구현 측면의 의존성을 감소시킨다. 아래 그림은 시스템에서 Host Controller를 직접 사용하는 디바이스 연결 관리 모듈, Transaction 처리 모듈 등을 별도의 패키지로 관리하여 시스템 내부의 다른 기능과 분리한 구조에 대한 설명이다.

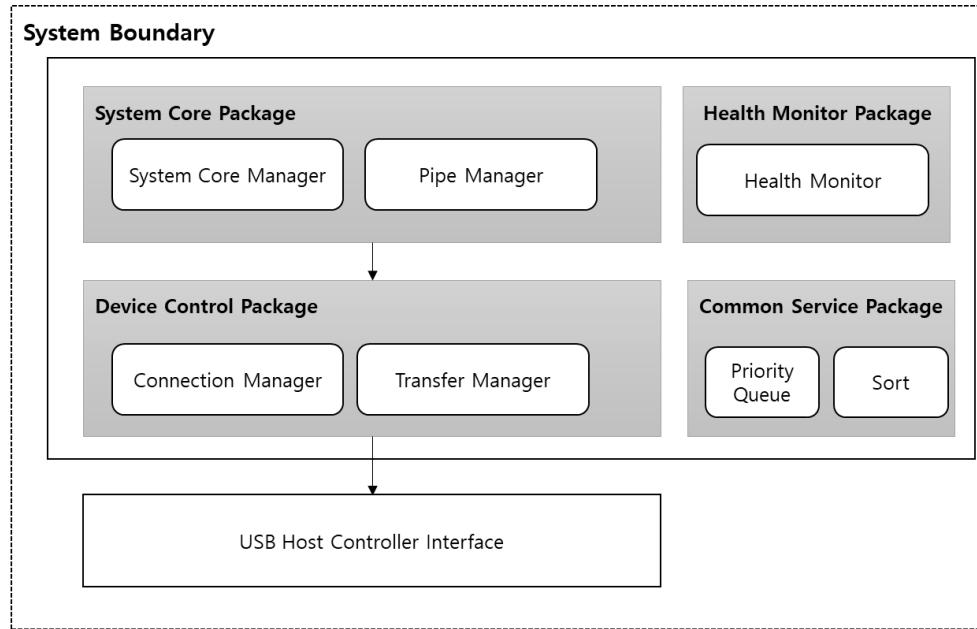


그림 99 Host Controller에 의존적인 모듈을 독립된 Package로 분리한 구조

#### D7.3. CA\_52. Transfer type별 Interface 모듈 분리 구조

<변경 요인 2>를 개선하기 위한 후보 구조이다. Host Controller의 전송은 4가지 Type으로 구분되며, 전송 Type에 따른 transaction 구성 방법과 Host Controller 설정 방법도 달라진다. 따라서, 전송 type 별로 HW 제어 모듈을 분리할 경우 HCI 변경시 Host Controller 설정 방식이 전송 Type별로 변경되는 경우에 대한 수정 범위를 최소화할 수 있다. 다음 그림은 Transfer Type별 Host Controller 설정 모듈을 분리한 구조에 대한 설명이다.

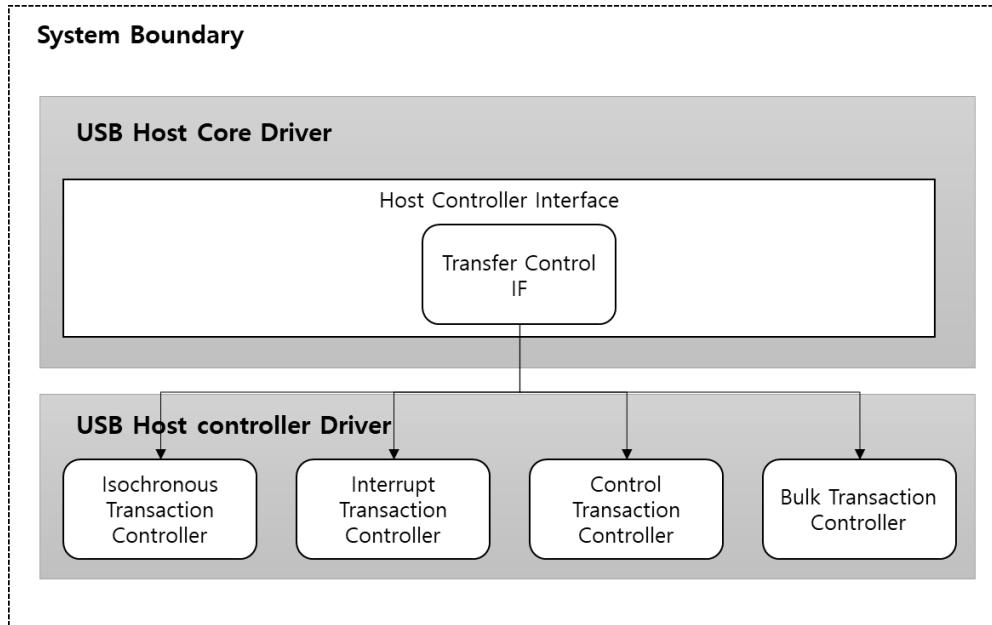


그림 100 Transfer Type별 HW control 모듈 분리 구조

#### D7.4. CA\_53. 전송 특징(Async와 Periodic)에 따라 Interface 모듈 분리 구조

<변경 요인 2>를 개선하기 위한 후보 구조이다. Host Controller의 전송 기능은 전송 특징에 따라 Periodic 방식과 Asynchronous 방식으로 구분되며, 각각의 처리 방식도 달라진다. EHCI의 경우 Periodic 방식의 전송은 Array와 Link list를 조합하여 전송 데이터를 관리하고, Asynchronous 전송의 경우 ring buffer 형태로 전송 데이터를 관리한다. 따라서 HCI 종류에 따라 전송 데이터를 관리하는 방식이 달라질 수 있기 때문에 전송 방식에 따라 이를 처리하는 모듈을 통합하여 다른 모듈에 대한 영향성을 줄일 수 있다. 다음 그림은 전송 방식에 따라 모듈을 분리한 구조에 대한 설명이다.

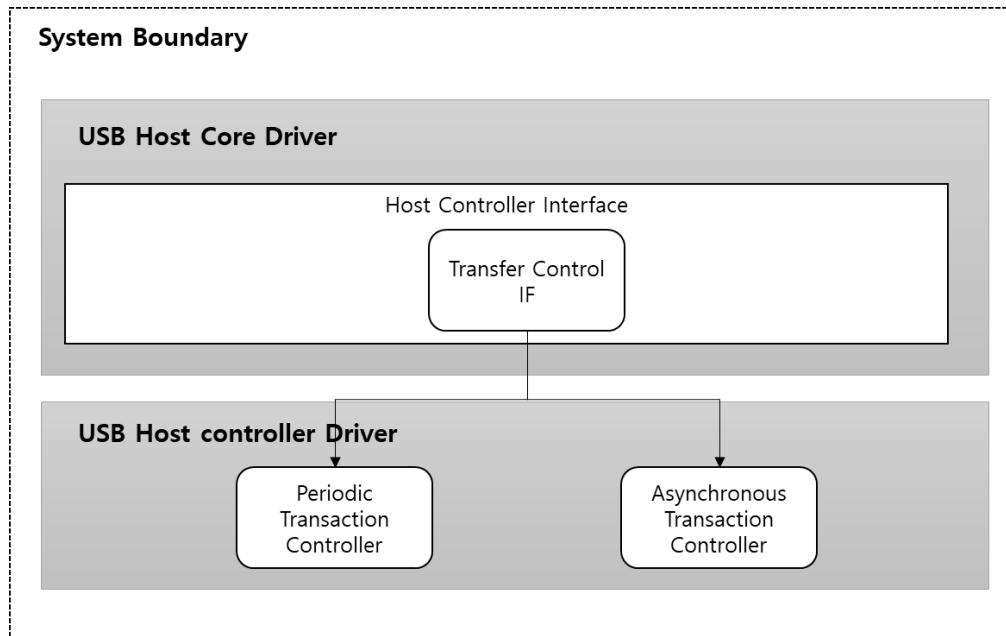


그림 101 전송 특징에 따라 HW control 모듈 분리 구조

## E. 후보 구조 평가

본 장에서는 도출된 후보 구조에 대해서 품질 속성을 바탕으로 비교 평가하였고, 평가된 결과는 다음과 같다.

### E1. NFR\_01 디바이스 연결 성공 비율

**CA\_1. 자원 관리자를 통한 시스템 자원 관리 통합 구조(채택)**

**CA\_2. 낭비되는 자원에 대한 garbage correction(채택)**

품질 요구 사항	영향(CA_1/CA_2)
NFR_01. 디바이스 연결 성공 비율	(++) 디바이스 연결에 필요한 자원을 통합 관리하여 오류 발생 가능성은 낮춘다. USB BUS에서 낭비되는 자원을 방지하여 디바이스 연결 시 자원이 부족한 상황을 예방한다.
NFR_02. 디바이스 입력 처리 시간	(+) 자원을 효율적으로 관리하여 디바이스 입력 처리에 필요한 자원이 부족한 상황을 방지할 수 있다.
QA_01. 데이터 전송 속도	(+) 자원을 효율적으로 관리하여 데이터 전송에 필요한 USB BUS 자원을 최대한 사용할 수 있다.
QA_02. 동시에 연결할 수 있는 디바이스의 수	(+) 자원을 효율적으로 관리하여 다수의 디바이스 연결에 필요한 자원을 확보할 수 있다.

**[채택 근거]** USB BUS의 자원(Bandwidth 등)은 한정되어 있기 때문에 이를 효율적으로 관리하는 것이 중요하다. 시스템의 각 컴포넌트가 개별적으로 자원을 관리할 경우 오류 발생 가능성이 높고, 자원관리의 동기화 문제등이 발생할 가능성이 높기 때문에 자원을 관리하는 별도의 컴포넌트가 있는 것이 효율적이라 판단하였다. USB BUS 상에서 낭비되는 자원이 없는지 주기적으로 모니터링 하여 낭비되는 자원을 회수할 경우 자원이 부족하여 디바이스 연결이 거부되는 상황을 방지할 수 있기 때문에 Garbage correction 컴포넌트도 함께 채택하였다. 또한, 자원을 효율적으로 관리할 수 있게 되면, 필요한 디바이스에 적절한 자원을 분배할 수 있을 것으로 판단하여 디바이스 입력 처리 시간, 데이터 전송 속도, 동시에 연결 가능한 디바이스의 수와 같은 품질 속성에도 플러스 요인으로 작용할 것으로 판단하였다.

**CA\_3. 자원 할당 재시도 (채택)**

**CA\_6. Emulation 과정을 재시도 (채택)**

품질 요구 사항	영향(CA_3/CA_6)
NFR_01. 디바이스 연결 성공 비율	(+) 일시적인 오류 상황에 대한 복구가 가능 하다.

**[채택 근거]** 시스템의 일시적인 오류/자원 부족 등의 상황에서 가장 간단하게 복구를 시도할 수 있는 방법으로 자원할당이나 Emulation 과정을 재시도 하는 방법을 적용할 수 있다. 예를 들어 다수의 디바이스가 연결된 상황에서 특정 디바이스가 idle 상태로 변경되는 경우 해당 디바이스에 할당된 자원을 새로 연결되는 디바이스에 사용할 수도 있다. 따라서 간단한 재시도를 통해 일시적인 자원 부족 등으로 인한 연결 실패 문제가 해결될 가능성이 있기 때문에 재시도 구조를 채택하였다.

**[Risk/단점]** 오류나 자원 할당이 복구되지 않는 상황에서도 재시도 반복하는 경우 시스템의 안정성과 성능을 저하시킬 수도 있고, 수회 반복 시에도 동일한 상황이 발생하는 경우 복구되지 않는 문제일 가능성이 높기 때문에 재시도 횟수는 3회로 제한하여 적용한다.

**CA\_4. 디바이스 매니저를 통한 디바이스 정보 통합 관리(채택)**

품질 요구 사항	영향(CA_4)
NFR_01. 디바이스 연결 성공 비율	(+) 디바이스 정보를 통합 관리하여 디바이스 정보의 오류로 인한 연결 실패 가능성을 낮춘다.
QA_03. 동시에 연결할 수 있는 디바이스의 수	(+) 디바이스 정보 오류로 인해 디바이스 연결이 실패할 가능성을 낮춘다.

**[채택 근거]** 시스템이 디바이스 와 연결하기 위해서는 디바이스에 할당된 Address, Description, Configuration, 디바이스 상태, 사용하는 Application 정보 등과 같은 많은 정보들이 관리되어야 한다. 해당 정보들을 시스템 내의 각각의 컴포넌트가 직접 접근하는 경우 내부 구현이 복잡성이 높아지고, 동기화 문제 등으로 인해 시스템 동작 오류를 발생시킬 가능성도 높아진다. 따라서, 디바이스 정보를 관리하는 별도의 컴포넌트를 적용하고 해당 컴포넌트를 통해서만 디바이스 정보를 접근하는 구조를 채택하였다.

**CA\_5. Control 정보를 위한 Bandwidth Reservation(채택 안함)**

품질 요구 사항	영향(CA_5)
NFR_01. 디바이스 연결 성공	(+) 디바이스 연결과정에서 control 정보 전달이 보장되기

비율	때문에 디바이스 연결을 위한 Control 정보 전달 실패 가능성을 제거할 수 있다.
NFR_02. 디바이스 입력 처리 시간	(-) USB BUS 자원 중 일부를 활용하지 못하기 때문에 디바이스 입력 처리를 위한 자원 할당에 영향을 줄 수 있다.
QA_01. 데이터 전송 속도	(-) control 정보가 전달되지 않는 상황에서는 USB BUS 자원 중 일부를 활용하지 못하기 때문에 자원 사용의 효율성이 떨어진다.
QA_02. 디바이스의 연결/해제시 인식 지연 시간	(+) 디바이스 연결을 위한 control 정보를 위한 자원이 reserve 되어 있기 때문에 디바이스 연결/해제시 자원 할당을 대기하는 시간이 줄어 든다.

[채택하지 않은 근거] 디바이스 연결에 필요한 자원은 항상 보장되어 있어 디바이스 연결 오류 가능성은 낮아 지지만, 디바이스 연결 해제 시점이외에는 해당 자원을 활용할 수 없기 때문에 데이터 전송 속도 및 디바이스 입력 처리 시간에 마이너스 요인이 된다. 따라서, 별도의 자원을 reserve 하는 방식 대신 transaction의 우선 순위에 따라 scheduling하는 후보 구조가 더 적절하다고 판단하였다.

#### CA\_7. 메모리 복사 기반의 전송 데이터 전달(채택 안함)

#### CA\_33. 메모리 참조 기반의 전송 데이터 전달 구조(채택)

품질 요구 사항	영향	
	CA_7	CA_33
NFR_01. 디바이스 연결 성공 비율	(+) 시스템 메모리를 외부로부터 isolation 할 수 있어 외부 오류로 인한 영향을 줄일 수 있다.	(-) 시스템 메모리가 외부와 공유되기 때문에 외부 오류에 시스템이 영향을 받을 수 있다.
NFR_02. 디바이스 입력 처리 시간	(-) 데이터를 복사하는데 필요한 시간 지연이 추가된다.	(++) 데이터 복사를 위한 시간 지연이 없다.
QA_01. 데이터 전송 속도	(-) 데이터를 복사하는데 필요한 시간 지연이 추가된다	(++) 데이터 복사를 위한 시간 지연이 없다.
QA_02. 디바이스의 연결/해제시 인식 지연 시간	(-) 데이터를 복사하는데 필요한 시간 지연이 추가된다	(++) 데이터 복사를 위한 시간 지연이 없다.

[채택 근거] 시스템이 외부와 주고받는 데이터를 복사하여 사용하는 경우 외부 메모리와 isolation되어 외부의 동작 오류 등으로 인해 시스템이 영향 받을 가능성이 줄어드는 장점이 있다. 하지만, 메모리 복사를 위한 추가적인 시간이 필요하기 때문에 가장 우선 순위가 높은 QA인 데이터 전송 속도 등의 성능에는 마이너스 요인이 된다. 특히 Bulk data 전송과 같이 전송 데이터의 크기가 큰 경우에는

메모리 복사로 인한 시간 지연이 데이터 전송 속도를 저하시킬 가능성이 매우 높기 때문에 성능을 향상시킬 수 있는 메모리 참조 기반의 방식을 채택하였다.

**[Risk/단점]** 공유된 메모리가 시스템이 참조하는 동안 변경되는 경우 시스템에 예기치 않은 오류가 발생할 가능성이 있다. 시스템에서 메모리 참조로 접근하는 외부 메모리는 대부분 전송 데이터로 시스템의 동작에 영향을 줄 가능성이 높지는 않지만, 시스템의 안정성을 위해 해당 메모리에 대한 접근을 데이터를 사용하는 Application과 본 시스템으로 제한할 필요가 있다. 시스템 외부적으로 OS에서 제공하는 Memory Access Control(MMU등 활용) 기능과 메모리 Sanitizer 기능 등을 활용하여 비정상적인 메모리 접근에 대한 Risk를 보완할 수 있다. 시스템 내부적으로는 CA\_8의 Health Monitor를 통해 시스템의 비정상 상황을 탐지하고 복구할 수 있도록 하여 risk를 감소시킬 수 있다.

#### CA\_8. Ping/Echo 기반 시스템 health check(채택)

#### CA\_9. Heartbeat 기반 시스템 health check(채택 안함)

품질 요구 사항	영향	
	CA_8	CA_9
NFR_01. 디바이스 연결 성공 비율	(++) 시스템 내부 오류 발생 시 복구를 통해 연결 성공 비율을 높일 수 있다.	(++) 시스템 내부 오류 발생 시 복구를 통해 연결 성공 비율을 높일 수 있다.

**[채택 근거]** Ping/Echo 방식과 Heartbeat 방식 모두 컴포넌트의 비정상 동작을 감시할 수 있기 때문에 디바이스 연결 성공 비율을 높이는데 효과가 있다. 하지만, 시스템의 경우 컴포넌트의 수가 제한적이고 확장 가능성이 크지 않기 때문에 확장성 보다는 SW 구현 복잡도와 효율성 측면에서 Ping/Echo 방식을 선택하였다. Ping/Echo 방식의 경우 시스템 내부의 각 컴포넌트는 Health Manager 컴포넌트에서 요청하는 메시지에 대해서만 응답을 전달하면 되지만, Heartbeat 방식의 경우 각 컴포넌트가 Heartbeat 생성을 위한 별도의 주기를 관리해야 한다. 즉, Heartbeat 생성을 위한 별도의 Timer 기능을 각 컴포넌트가 구현해야 하기 때문에 이로 인한 SW 복잡도가 높아질 수 있다. 또한, Timer interrupt를 처리하는 과정에서 발생하는 오버헤드가 Message를 처리하는 방식에 비해 상대적으로 클 것으로 판단하여 본 시스템의 경우에는 성능 측면에서도 Ping/Echo 방식이 유리할 것으로 판단하였다.

**[Risk/단점]** Ping/Echo 메시지가 반복 주기가 짧은 경우 각 컴포넌트가 Ping/Echo를 처리하는데 불필요한 오버헤드가 발생할 수 있다. 따라서 시스템에 Overhead를 발생시키지 않는 수준의 주기 설정이

필요하다. 또한, 각 컴포넌트가 가지고 있는 메시지 큐에서 Ping/Echo 메시지의 우선순위가 데이터 전송메시지에 우선 순위가 밀려 Starvation이 되지 않도록 Ping에 대한 Echo를 시간 범위 내에서 전달할 수 있도록 보장하는 메시지 스케줄링이 필요하다.

## E2. NFR\_02 디바이스로부터 입력에 대한 처리 시간

**CA\_10. Array 기반의 pipe 매핑 정보 관리(채택 안함)**

**CA\_11. List 기반의 pipe 매핑 정보 관리(채택 안함)**

**CA\_12. Cache 기반의 pipe 매핑 정보 관리(채택)**

품질 요구 사항	영향		
	CA_10	CA_11	CA_12
NFR_02. 디바이스 입력 처리 시간	(+) Pipe 번호를 배열의 index로 하여 O(1)의 시간에 검색 가능하다. 디바이스 정보로 검색시에는 Worst case에 배열 전체를 검색해야 한다.	(0) 입력을 처리할 Application 확인시 List를 전체 검색하는 시간이 소모된다.	(++) Cache에 포함된 매핑 정보를 빠르게 검색 가능하다.
QA_01. 데이터 전송 속도	(+) Pipe로 검색시에는 O(1), 디바이스 정보로 검색 시에는 O(N) 시간이 걸린다.	(0) Pipe로 검색, 디바이스로 검색하는 두 경우 모두 O(N)시간이 걸린다.	(++) Cache에 포함된 정보로 데이터를 전송할 디바이스와 Application 검색 시간을 단축할 수 있다.
QA_02. 디바이스의 연결/해제시 인식 지연 시간	(+) Pipe로 검색시에는 O(1), 디바이스 정보로 검색 시에는 O(N) 시간이 걸림	(0) Pipe로 검색, 디바이스로 검색하는 두 경우 모두 O(N)시간이 걸린다.	(++) Cache에 포함된 정보로 디바이스 연결을 통보할 Application 검색 시간을 단축할 수 있다.

[채택 근거] 자주 사용되는 Pipe 매핑 정보는 Application에서 디바이스로 데이터를 보낼 때 와 디바이스에서 발생한 입력을 Application으로 통보할 때 검색이 필요하다. Pipe 매핑 정보에 대한 캐시를

적용하는 경우 Application 기준 검색과, 디바이스 기준 검색 모두 일정한 수준의 시간으로 검색이 가능하다.

### CA\_13. Periodic Transaction 처리를 위한 전용 Thread 운영 구조(채택 안함)

품질 요구 사항	영향(CA_13)
NFR_01. 디바이스 연결 성공 비율	(-) 시스템 내부 Thread 가 많아 질수록 Thread 간 동기화 및 SW 복잡도 높아져 오류 가능성성이 높아진다.
NFR_02. 디바이스 입력 처리 시간	(+) 디바이스의 입력 처리와 다른 데이터 처리 Thread 가 병렬로 처리될 수 있어 디바이스 입력 처리 시간을 줄일 수 있다.
QA_01. 데이터 전송 속도	(0) 디바이스 입력 처리에 필요한 별도의 자원을 할당하게 되어 시스템 전체 데이터 전송 속도에 필요한 자원을 소모할 수 있다.
QA_02. 디바이스의 연결/해제시 인식 지연 시간	(+) 디바이스 입력 처리와 디바이스 연결/해제 처리 Thread 가 분리되어 병렬 처리가 가능한다.

[채택하지 않은 근거] 디바이스 입력 처리에 필요한 별도의 시스템 자원을 할당하게 되어 시스템 전체적인 데이터 처리 성능에 영향을 줄 수 있다. 또한, 데이터 전송 속도와 관련된 가장 중요한 후보 구조인 “CA\_21. Thread 임의 할당 방식”과 conflict 이 발생하여 후보 구조로 채택하지 않았다.

### CA\_14. Interrupt 방식의 시스템<->Application간 이벤트 전달 구조(채택)

### CA\_15. Polling 방식의 시스템<->Application간 이벤트 전달 구조(채택 안함)

품질 요구 사항	영향	
	CA_14	CA_15
NFR_02. 디바이스 입력 처리 시간	(+) 디바이스 입력 발생시 Application 으로 즉시 통보 가능 하다.	(0) polling 주기 동안 시간 지연이 발생한다.
QA_02. 디바이스의 연결/해제시 인식 지연 시간	(+) 디바이스 연결/해제 발생시 Application 으로 즉시 통보 가능 하다.	(0) polling 주기 동안 시간 지연이 발생한다.
QA_05. 새로운 실행 환경(HW, OS) 지원 용이성	(-) OS 별 Interrupt 처리 방식 등의 차이가 발생할 수 있어 SW 추가 구현이 필요할 수 있다.	(0) OS 환경 등에 영향이 없다.

**[채택 근거]** 시스템에서 디바이스 입력 혹은 디바이스 연결 이벤트를 인식하는 시점에 Interrupt를 통해 Application에 즉시 통보할 수 있어 시간 지연을 최소화할 수 있다. 또한, Application과 시스템이 별도의 Process 혹은 Thread로 동작하는 경우 Polling을 위한 빈번한 context switching 시간을 줄일 수 있어 시스템 전체의 성능 측면에서도 이점이 있다.

**[Risk/단점]** Interrupt 처리 방식은 OS나 실행 환경에 따라 차이가 날 수 있기 때문에, OS별로 별도 구현이 필요할 수 있다. OS별로 차이 나는 구현은 CA\_45를 통해 보완할 수 있다.

### CA\_16 디바이스 입력에 대한 Pre-fetch 구조(채택 안함)

품질 요구 사항	영향(CA_16)
NFR_02. 디바이스 입력 처리 시간	(++) 디바이스 입력을 정해진 주기보다 자주 읽어 오기 때문에 입력 인식 시간을 최소화할 수 있다.
QA_01. 데이터 전송 속도	(-) 과도한 USB BUS의 bandwidth 사용으로 시스템의 전체 성능은 저하시킬 수 있다..
QA_02. 디바이스의 연결/해제시 인식 지연 시간	(-) 과도한 USB BUS의 bandwidth 사용으로 시스템의 전체 성능은 저하시킬 수 있다.

**[채택하지 않은 근거]** 디바이스 입력을 정해진 주기보다 빨리 읽어오는 경우 디바이스 입력 처리 시간은 줄일 수 있지만, USB Bus의 bandwidth를 과도하게 사용하여 전체 데이터 전송 성능과 디바이스 인식/해제 시간을 저하시킬 수 있는 문제가 있어 채택하지 않았다.

### E3. QA\_01 데이터 전송 속도

#### CA\_17 데이터 Transfer와 Transaction을 단일 Thread에서 운영 하는 구조(채택 안함)

#### CA\_18 데이터 Transfer와 Transaction을 개별 Thread에서 운영 하는 구조(채택)

품질 요구 사항	영향	
	CA_17	CA_18
NFR_01. 디바이스 연결 성공 비율	(-) 리소스 및 디바이스 정보등의 공유 정보 접근시 동기화 문제 등으로 SW 복잡도 및 시스템 내부 오류 가능성	(-) 리소스 및 디바이스 정보등의 공유 정보 접근시 동기화 문제 등으로 SW 복잡도 및 시스템 내부 오류

	증가할 수 있다.	가능성 증가할 수 있다.
NFR_02. 디바이스 입력 처리 시간	(+) 디바이스 입력 처리와 데이터 전송 처리를 병렬로 할 수 있어 디바이스 입력 시간 지연을 줄일 수 있다.	(+) 디바이스 입력 처리와 데이터 전송 처리를 병렬로 할 수 있어 디바이스 입력 시간 지연을 줄일 수 있다.
QA_01. 데이터 전송 속도	(+) 디바이스 입력 처리와 데이터 전송 처리를 병렬로 할 수 있어 전송 속도를 개선할 수 있다.	(++) Transfer 정보를 Transaction으로 fragment 하는 기능과 Transaction을 Frame으로 조합하는 기능을 병렬 처리하여 전송 속도를 향상시킨다.
QA_02. 디바이스의 연결/해제시 인식 지연 시간	(+) 디바이스 입력 처리와 데이터 전송 처리를 병렬로 할 수 있어 디바이스 연결 해제 인식 지연 시간을 줄일 수 있다.	(+) 디바이스 입력 처리와 데이터 전송 처리를 병렬로 할 수 있어 디바이스 연결 해제 인식 지연 시간을 줄일 수 있다.

**[채택 근거]** 데이터 전송을 전담하는 기능을 별도의 Thread로 분리함으로써 연결 처리, Health check 등 시스템의 다른 기능과 병렬 처리가 가능해져 디바이스 입력 처리 시간을 줄일 수 있고, 데이터 전송 속도와 같은 다른 성능을 향상시킬 수 있는 이점도 있다. 또한, Transfer에 대한 Fragment 기능과 Transaction을 Frame으로 생성하는 기능을 병렬 처리하여 성능을 더욱 개선할 수 있다.

**[Risk/단점]** 시스템의 동작 Thread간의 공통 리소스 접근에 대한 동기화 문제등이 발생할 가능성성이 높아지기 때문에 공유 자원 관리를 안전하게 할 수 있는 방안에 대한 검토가 필요하다. 자원 관리의 안정성을 높일 수 있는 CA\_1/CA\_2 후보 구조로 단점을 보완할 수 있다.

#### CA\_19. Transfer 요청별 Thread 생성(Creation) 구조(채택 안함)

#### CA\_20. Transfer 요청 type별 Thread 할당(Dispatch) 구조(채택 안함)

#### CA\_21. Transfer 요청에 대한 Thread 임의 할당(Random Dispatch) 구조(채택)

품질 요구 사항	영향		CA_21
	CA_19	CA_20	
NFR_01. 디바이스 연결 성공 비율	(--) 다수의 Transfer 요청이 발생시 SW 복잡도	(-) 전송 Thread에 대한 동기화 문제로 내부 오류	(-) 전송 Thread에 대한 동기화

	및 공유 자원에 대한 동기화 문제로 내부 오류 발생 가능성이 증가할 수 있다. 특히, Thread 수에 제한이 없는 단점이 있다.	발생 가능성이 증가할 수 있다.	문제로 내부 오류 발생 가능성이 증가 할 수 있다.
NFR_02. 디바이스 입력 처리 시간	(0) 디바이스 입력은 Scheduling에 따라 처리되어 특별한 영향이 없다.	(+) 디바이스 입력에 대한 우선 순위를 관리하기 용이하다.	(0) 디바이스 입력은 Scheduling에 따라 처리되어 특별한 영향이 없다.
QA_01. 데이터 전송 속도	(+) 입력에 대한 병렬 처리 효율은 좋지만, thread 생성을 위한 오버헤드가 증가할 수 있다..	(+) 동일한 type의 전송이 동시에 발생할 경우, 해당 type을 처리하는 thread에 부하가 발생할 수 있다.	(++) 다수의 Thread가 미리 생성되어 있어 Thread 생성을 위한 오버헤드가 제거되고, Dispatcher를 통해 thread 간 Load 관리가 가능하다.

**[채택 근거]** 데이터 전송을 전담하는 Thread를 미리 생성하여 전송 시점에 Thread를 생성하는 오버헤드를 줄일 수 있다. 또한, Dispatcher가 각 전송 Thread의 오버헤드를 확인하여 Thread간 load를 적절히 분해하여 전송 요청을 처리하기 때문에 특정 Type이 전송이 동시에 몰리는 경우에도 전송을 효과적으로 처리할 수 있다.

**[Risk/단점]** 전송을 Thread간에 공유하는 공통 리소스 접근에 대한 동기화 문제등이 발생할 가능성이 높아지기 때문에 공유 자원 관리를 안전하게 할 수 있는 방안에 대한 검토가 필요하다. 자원 관리의 안정성을 높일 수 있는 CA\_1/CA\_2 후보 구조로 단점을 보완할 수 있다.

#### CA\_22. Round Robin 방식의 Transfer 할당(채택 안함)

#### CA\_23. Priority Queue 방식의 Transfer 할당(채택)

품질 요구 사항	영향	
	CA_22	CA_23

NFR_02. 디바이스 입력 처리 시간	(0) 입력 데이터에 대한 처리 우선 순위를 관리하기 어렵다.	(+) Transfer Thread 간 우선 순위 설정을 위한 별도의 오버헤드가 필요 없다. 처리 상황에 따라 우선 순위를 배정하기는 어렵다.
QA_01. 데이터 전송 속도	(+) 디바이스 입력 처리 요청이 몰리는 경우에도 Transfer Thread 의 우선 순위에 따라 Load 분산이 가능하다.	(++) Transfer Thread 의 우선 순위에 따라 Load 분산 되어 병렬 처리 효율이 높아진다.

[채택 근거] Transfer 요청에 따라 각 Thread에 필요한 처리 시간 등은 다를 수 있다. 따라서 단순히 Round Robin으로 Transfer 요청을 Thread에 배정할 경우 처리 시간이 오래 걸리는 Transfer 요청이 특정 Thread에 집중되어 병렬 처리 효과가 저하될 가능성이 있다. 따라서, 각 Thread의 처리 상황을 모니터링 하여 가장 빠른 시간에 Transfer 요청을 처리할 수 있는 Thread를 선택하여 Transfer 요청을 전달하는 것이 병렬처리 효율을 높일 수 있는 구조로 판단하였다.

#### CA\_24. On-demand 방식의 Frame 생성 (채택 안함)

#### CA\_25. Pre-generation 방식의 Frame 생성(채택 안함)

#### CA\_26. Buffer Queue 방식의 frame 생성 구조(채택)

품질 요구 사항	영향		
	CA_24	CA_25	CA_26
QA_01. 데이터 전송 속도	(0) Frame 생성 시간도 전송 시간에 포함된다.	(++) 전송할 Frame 을 미리 생성해 두기 때문에 전송 시간을 줄일 수 있다.	(++) 전송할 Frame 을 미리 생성해 두기 때문에 전송 시간을 줄일 수 있다.
QA_02. 디바이스의 연결/해제시 인식 지연 시간	(0) Frame 생성 시간도 디바이스 인식 해제 시간에 포함된다.	(+) 디바이스 인식을 위한 Control 정보 Frame 을 미리 생성하여 인식 시간을 줄일 수 있다.	(+) 디바이스 인식을 위한 Control 정보 Frame 을 미리 생성하여 인식 시간을 줄일 수 있다.

[채택 근거] Pre-Generation 방식과 Buffer Queue 방식 모두 전송할 Frame을 미리 생성해 두기 때문에 데이터 전송 시간을 줄일 수 있는 장점이 있다. 하지만, Pre-Generation 방식의 경우 Bulk data 전송과 같은 전송 데이터의 크기가 큰 경우 Frame을 미리 생성하기 위한 메모리 낭비가 과도하게 발생

할 수 있다. 반면에 Buffer Queue 형태로 미리 생성한 Frame을 유지하는 경우 적은 양의 메모리 추가로 동일한 성능 향상을 달성할 수 있기 때문에 Buffer Queue를 이용하는 방식을 채택하였다.

**[Risk/단점]** Buffer Queue의 커지는 경우 메모리 낭비가 발생할 수 있다. 반면 너무 작은 경우 Host controller에서 Buffer Queue에 있는 데이터를 사용하는 속도가 SW가 Buffer Queue를 채우는 속도가 따라 가지 못하여 전송 효율이 떨어질 수 있기 때문에, 실제 동작환경에서 실험 등을 통해 적절한 Buffer size를 결정하여야 한다. 또한, 동적으로 Buffer의 Size를 조절하는 방안도 검토해 볼 수 있다.

#### CA\_27. 단일 List 기반 Transaction 목록 관리 구조(채택 안함)

#### CA\_28. Transfer type별 Transfer목록 관리 구조(채택)

품질 요구 사항	영향	
	CA_19	CA_20
QA_01. 데이터 전송 속도	(0) Transfer 의 우선 순위를 정하기 위해 Worst case 에 O(N) 만큼의 시간 지연이 필요하다.	(++) Transfer 을 추가하고 우선 순위를 정하는데 O(1) 만큼의 시간만 필요하다.
QA_02. 디바이스의 연결/해제시 인식 지연 시간	(0) 디바이스 연결을 위한 Control 정보 처리 시간도 Transfer 의 처리 지연 시간에 영향을 받는다.	(++) Transfer 을 추가하고 우선 순위를 정하는데 O(1) 만큼의 시간만 필요하다.

**[채택 근거]** 시스템에 다수의 디바이스가 연결된 경우 우선 순위가 다른 여러 요청이 동시에 시스템에 전달될 수 있다. 따라서, 다수의 Transfer목록 중 우선 순위가 높은 메시지를 빠르게 정렬하고 처리하는 것은 전체 데이터 전송 성능에 영향을 준다. USB의 4가지 전송 type에 대한 우선 순위를 고정해 두고 각 type별로 추가되는 Transfer만을 관리하는 경우 Transfer 추가시 별도의 우선 순위 정렬이 필요 없기 때문에 Transfer 목록에 추가하는 시간을 줄일 수 있다.

**[Risk/단점]** 4가지 전송 type에 대한 우선 순위가 개발 시점에 고정되기 때문에 우선 순위가 높은 요청이 계속해서 발생할 경우 우선 순위가 낮은 type은 starvation이 발생할 가능성이 있다. 따라서 우선 순위가 낮은 요청에 대한 starvation을 방지할 수 있는 동적 우선 순위 조정 방법도 검토가 필요하다.

**CA\_29. FIFO 기반 Transfer scheduling (채택 안함)****CA\_30. Priority 기반 Transfer scheduling(채택)**

품질 요구 사항	영향	
	CA_29	CA_30
NFR_02. 디바이스 입력 처리 시간	(0) FIFO로 처리되기 때문에 디바이스 입력이 우선 순위가 밀릴 수 있다.	(+) 디바이스 입력과 같은 우선 순위가 높은 작업을 먼저 처리할 수 있다.
QA_01. 데이터 전송 속도	(+) FIFO로 처리되기 때문에 Transaction 간 우선 순위를 정하는데 소요되는 시간이 없다.	(0) 같은 type 내에서 우선 순위를 정하는데 $O(\log N)$ 시간 복잡도를 가진다.

**[채택 근거]** 동일 전송 type 내에서도 디바이스 입력과 같은 우선 순위가 높은 Transfer를 먼저 처리할 필요가 있는 경우 이에 대한 처리가 필요하기 때문에 우선 순위 기반 scheduling 정책을 채택하였다.

**[Risk/단점]** 동일 전송 type 내에서 우선 순위 정렬을 위한 시간 지연이 추가적으로 필요하기 때문에 시간 지연을 최소화할 수 있는 알고리즘 적용이 필요하다. Priority Queue와 같은 우선 순위 알고리즘을 사용할 경우  $O(\log N)$ 의 시간에 정렬이 가능하다.

**CA\_31. 단일 Message Queue 기반 컴포넌트간 메시지 전달 구조(채택 안함)****CA\_32. 컴포넌트별 Message Queue 관리 구조(채택)**

품질 요구 사항	영향	
	CA_31	CA_32
QA_01. 데이터 전송 속도	(0) 전체 메시가 하나의 메시지 큐로 관리되기 때문에 메시지의 우선 순위 정렬에 대한 시간 지연이 크다.	(+) 메시지 큐가 각각의 컴포넌트 별로 분산되어 있어 메시지 정렬이 병렬 처리되어 시간 지연이 줄어 듈다.

**[채택 근거]** 시스템 내부의 메시지 처리 성능 측면에서 컴포넌트별로 독립된 메시지 큐를 가지는 경우 메시지 큐에 대한 우선 순위 정렬을 병렬로 처리할 수 있어 단일 메시지 큐에 비해 이점을 가진다. 또한, 컴포넌트별로 메시지에 대한 우선 순위를 정할 수 있어 우선 순위 적용을 상황에 따라 유연하게 적용할 수 있는 이점도 있다.

**[Risk/단점]** 컴포넌트 간에 메시지 교환이 가능한 조합을 개발 시점에 미리 정해야 하기 때문에 컴포넌트의 수가 늘어날 경우 확장성에서는 단점을 가질 수 있다. 하지만, 본 과제의 시스템의 경우 명시된 기능을 제공하기 때문에 시스템 내부 컴포넌트의 확장 가능성은 높지 않은 것으로 판단하였다.

#### CA\_34. Tree 기반의 Configuration 정보 관리 구조(**채택**)

#### CA\_35. Tree 기반의 BUS topology 정보 관리 구조(**채택**)

품질 요구 사항	영향(CA_34/CA_35)
NFR_02. 디바이스 입력 처리 시간	(+) 시스템 자원 관리를 위한 정보 처리 시간을 줄여 USB Bus 자원 사용의 효율성을 높일 수 있기 때문에, 디바이스 입력 처리를 위해 필요한 자원(Bandwidth) 할당 비율도 높아질 수 있다.
QA_01. 데이터 전송 속도	(+) 시스템 자원 관리를 위한 정보 처리 시간을 줄여 USB Bus 자원 사용의 효율성을 높일 수 있기 때문에, 시스템의 데이터 전송 효율도 함께 향상된다.
QA_02. 디바이스의 연결/해제시 인식 지연 시간	(+) 시스템 자원 관리를 위한 정보 처리 시간을 줄여 USB Bus 자원 사용의 효율성을 높일 수 있기 때문에, 디바이스 인식에 필요한 Control 메시지 처리를 위한 자원 할당 비율도 높아질 수 있다.

**[채택 근거]** 유휴 자원이 회수시 디바이스에 할당된 자원 정보를 빠르게 확인할수록 유휴 자원을 회수하여 가용 자원으로 활용하는 효율성이 높아진다. 디바이스별 Configuration 정보와 BUS Topology 정보 모두 계층적인 구조를 가지고 있기 때문에 Tree 형태의 자료 구조를 적용하여 검색 시간을 줄일 수 있다.

#### CA\_36. 데이터 Cache 관리 구조(**채택 안함**)

**[채택하지 않은 이유]** Bulk data와 같은 대량의 데이터 Read에는 Cache의 효용성이 떨어진다. 또한 Interrupt 방식의 read에서는 Cache를 운용할 수 없기 때문에 채택하지 않았다.

#### E4. QA\_02 디바이스의 연결/해제시 인식 지연 시간

**CA\_37. List 기반 디바이스 목록 관리(채택 안함)**

**CA\_38. Array 기반 디바이스 목록 관리(채택)**

품질 요구 사항	영향	
	CA_37	CA_38
QA_01. 데이터 전송 속도	(0) 데이터 전송을 위한 디바이스 정보 검색에 $O(N)$ 의 시간이 걸릴 수 있다.	(+) 데이터 전송을 위한 디바이스 정보 검색에 $O(1)$ 의 시간으로 가능 하다.
QA_02. 디바이스의 연결/해제시 인식 지연 시간	(+) 디바이스 해제시 기존 디바이스 정보 삭제에 $O(N)$ 의 시간이 걸릴 수 있다.	(+) 디바이스 연결/해제 모두 $O(1)$ 의 시간으로 디바이스 정보 검색이 가능하다.
QA_03. 동시에 연결할 수 있는 디바이스의 수	(++) 동시에 연결할 수 있는 디바이스의 수에 제한이 없다.	(-) 동시에 연결할 수 있는 디바이스의 수가 배열의 크기로 제한된다.

**[채택 근거]** 배열로 디바이스 정보를 관리할 경우, 디바이스 Address를 index로 사용하여  $O(1)$ 의 시간 복잡도로 디바이스 정보를 추가/삭제/검색이 가능하다.

**[Risk/단점]** 배열로 관리할 경우 동시에 지원할 수 있는 디바이스의 수가 배열의 크기로 제한되는 단점이 있다. 하지만, USB Spec상 최대 127개 까지의 디바이스를 동시에 연결할 수 있으면 되기 때문에, Worst case에 127개의 배열을 만들어도 메모리 낭비가 크지 않다.

**CA\_39. 디바이스 연결 처리를 위한 독립 Thread 구조(채택)**

**CA\_40. 디바이스 연결 요청 별 독립된 Thread 구조(채택 안함)**

품질 요구 사항	영향	
	CA_39	CA_40
NFR_01. 디바이스 연결 성공 비율	(-) 시스템 내부 Thread 증가로 인한 SW 복잡도와 공유 자원에 대한 동기화 문제 발생 가능성성이 높아짐	(--) 디바이스 연결 마다 새로운 Thread 가 생성되어 동기화 문제 발생 가능성성이 높아지고, Thread 생성/삭제 반복으로 인한 SW 복잡도 증가함.

QA_02. 디바이스의 연결/해제시 인식 지연 시간	(+) 디바이스 연결을 별도 Thread에서 처리하여 시스템의 다른 기능과 병렬로 처리 가능하다.	(++) 동시에 여러 개의 디바이스가 접속하는 경우에도 모두 병렬 처리 가능하다.
QA_03. 동시에 연결할 수 있는 디바이스의 수	(0) 동시에 연결할 수 있는 디바이스의 수에 제한이 없다.	(-) 동시에 여러 개의 디바이스가 연결되는 경우 실행 환경에 따라 Thread 생성 수가 제약 될 수 있다.

**[채택 근거]** 디바이스 연결을 처리하는 별도의 Thread가 있는 경우, 시스템의 다른 컴포넌트와 병렬로 동작할 수 있어 디바이스 연결/해제 시간을 줄일 수 있다. 매 연결 마다 새로운 Thread를 만드는 경우 병렬 처리 효율은 높일 수 있지만 Thread 생성을 위한 시스템 Overhead 발생 가능성이 있고, Thread 생성/삭제를 반복하는 것에 대한 SW 복잡도 증가 등의 단점이 있어 한 개의 Thread를 이용하는 방안을 채택하였다.

**[Risk/단점]** 연결 처리를 위한 별도 Thread를 운영하는 경우 시스템내의 공유 자원 접근시 동기화 문제등이 발생할 가능성이 높아질 수 있지만, 후보 구조 CA\_1, CA\_2, CA\_4를 통해 완화시킬 수 있다.

#### CA\_41. 연결 관리 객체의 Pre-Generation(채택)

품질 요구 사항	영향(CA_41)
QA_02. 디바이스의 연결/해제시 인식 지연 시간	(+) 디바이스 연결에 필요한 정보를 포함한 객체 생성 시간을 디바이스 연결 시간에서 제외시킬 수 있다.

**[채택 근거]** 시스템은 디바이스 연결시 <device address, device configuration, application mapping, device resource>등의 여러가지 정보를 관리하는 객체를 생성해야 하기 때문에 객체 생성 시간이 디바이스 연결 시간에 영향을 줄 수 있다. 따라서, 객체를 미리 생성해 두는 경우 디바이스 연결 시간에서 객체 생성시간을 제외할 수 있는 이점이 있다.

#### CA\_42. 연결관리 객체의 Lazy Clear(채택 안함)

**[채택하지 않은 이유]** 동일 디바이스가 짧은 시간내에 다시 연결되지 않는 경우 Lazy Clear로 인한 효과가 없기 때문에 Lazy Clear로 인한 시스템 메모리 낭비 및 SW 복잡도 대비 효과가 크지 않은 것으로 판단하였다.

**CA\_43. 가용 Address 목록 관리 구조(Heap)(채택 안함)****CA\_44. 가용 Address 목록 관리 구조(List)(채택)**

품질 요구 사항	영향	
	CA_43	CA_44
QA_02. 디바이스의 연결/해제시 인식 지연 시간	(+) 가용 Address 목록을 별도로 관리하여 Address 할당을 위한 검색 시간을 줄일 수 있다. 가용 Address 를 정렬하기 위한 시간이 추가 된다.	(++) 가용 Address 목록을 별도로 관리하여 O(1)의 시간 복잡도로 가용 Address 를 할당 할 수 있다.
QA_03. 동시에 연결할 수 있는 디바이스의 수	(-) 배열을 기반으로 가용 Address 목록을 관리하기 때문에 지원할 수 있는 디바이스의 수가 제한된다..	(-) 배열을 기반으로 가용 Address 목록을 관리하기 때문에 지원할 수 있는 디바이스의 수가 제한된다.

**[채택 근거]** 별도의 가용 Address 목록을 관리하여 전체 Address 목록에서 가용 Address를 검색하는 데 필요한 시간을 줄일 수 있다. 또한 USB Spec에서 정의한 최대 디바이스의 수(127개)를 고려할 경우 배열로 관리하는데 문제가 없는 수준이고, 별도 정렬 없이 배열 기반 FIFO로 할당할 경우 O(1)의 시간 복잡도로 사용 가능한 Address를 할당할 수 있다.

**[Risk/단점]** 지원할 수 있는 디바이스의 수가 배열을 크기로 제한되지만, USB Spec에서 정의한 최대 디바이스 수를 지원해도 메모리 낭비가 크지 않을 것으로 판단하였다.

**E5. QA\_03 동시에 연결할 수 있는 디바이스의 수****CA\_45. 디바이스 Reconfiguration을 통한 자원 재 분배 구조(채택)**

품질 요구 사항	영향(CA_45)
QA_03. 동시에 연결할 수 있는 디바이스의 수	(+) 새로운 디바이스를 위한 자원이 부족한 상황에서도 자원 재분배를 통해 디바이스 연결이 가능하다.
QA_04. 새로운 디바이스 클래스 추가 용이성	(-) Application 에 자원 재분배를 요청하는 구조로, 새로운 디바이스 Class 가 추가되는 경우 Application 을 위한 Interface

	정의 및 구현이 필요하다.
--	----------------

**[채택 근거]** 디바이스 연결을 위한 자원이 부족한 상황에서도 자원을 재 분배하여 디바이스 연결이 가능하게 하는 구조이다. 자원의 재분배를 시스템이 직접 하는 대신 각 Application에서 상황에 맞게 조절 할 수 있도록 하였기 때문에 자원 재분배로 인한 Application 동작 오류의 가능성도 낮다.

**[Risk/단점]** 새로운 디바이스 Class가 추가되는 경우 자원 재분배를 위한 Interface를 정의하고 구현하는 작업이 추가적일 필요 하다.

#### CA\_46. Delayed Resource Allocation 구조(채택 안함)

**[채택하지 않은 이유]** 디바이스에 필요한 자원의 할당을 자원 사용 시점으로 미뤄 동시에 여러 개의 디바이스를 연결할 수 있는 가능성은 높아 지지만, 자원 할당 구조가 복잡해진다. 또한, 디바이스가 연결된 상태에서 자원할당에 실패하여 동작하지 않는 등의 문제가 발생할 수 있어 채택 하지 않았다.

#### CA\_47. 디바이스 정보 저장 자료 구조의 크기(채택)

품질 요구 사항	영향(CA_47)
QA_03. 동시에 연결할 수 있는 디바이스의 수	(+) USB Spec에서 정의한 127 개를 지원한다.

**[채택 근거]** USB Spec상 최대 시스템이 지원할 수 있는 최대 디바이스의 수를 설계 단계에서 정의하기 위해 채택하였다.

### E6. QA\_04 새로운 디바이스 클래스 추가 용이성

#### CA\_48. USB Host Driver Interface 기능 분리(채택)

#### CA\_49. 디바이스 클래스별 Interface 적용(채택)

품질 요구 사항	영향(CA_48/CA_49)
QA_04. 새로운 디바이스 클래스 추가 용이성	(+) 새로운 디바이스 클래스 추가시 정의된 Interface를 맞추면 시스템 내부의 구조 변경 없이 지원 가능하다.

**[채택 근거]** 시스템과 Application 사이에 Interface 모듈 추가할 경우 새로운 디바이스 클래스가 추가

될 경우에도 시스템 내부의 구현 변경 없이 필요한 수정 사항을 Interface 모듈로 제한할 수 있다. 또한, 디바이스 클래스용 별도 Interface 모듈을 적용하는 경우 클래스 동작 변경으로 인한 영향을 해당 클래스에 대한 Interface로 제한할 수 있는 정점이 있다.

## E7. QA\_05 새로운 실행 환경(HW, OS) 지원 용이성

### CA\_50. Layer별 OS 기능 추상화 Interface 구조 (채택)

품질 요구 사항	영향(CA_50)
QA_05. 새로운 실행 환경(HW, OS) 지원 용이성	(+) OS 기능이 변경되어도 시스템 내부 구현에 영향을 주지 않는다.

**[채택 근거]** 시스템은 OS의 커널 드라이버 형태로 동작하기 때문에 OS의 여러 가지 기능들(ex, interrupt, memory 등)을 사용한다. 따라서 OS에서 해당 기능에 대한 사용 interface가 변경되는 경우 시스템의 구현도 영향을 받게 될 수 있다. 시스템에서 사용하는 OS의 기능들을 별도 모듈로 구현하여 interface를 단일화하는 경우 OS 변경으로 인한 시스템 변경을 해당 Interface로 제한할 수 있다.

### CA\_51. Host Controller 기능 사용 모듈 통합 (채택)

품질 요구 사항	영향(CA_51)
QA_05. 새로운 실행 환경(HW, OS) 지원 용이성	(+) HW(Host Controller)의 변경 사항에 따른 시스템 영향 범위를 Host Controller 기능 사용 패키지로 제한할 수 있다.

**[채택 근거]** 시스템은 Host Controller가 제공하는 기능을 사용하기 때문에 Host Controller의 기능이 추가되거나 Controller 방식이 변경되는 경우 시스템 구현도 변경이 필요하다. 시스템 기능 중 Host Controller를 사용하는 모듈들을 별도의 패키지로 통합하고 다른 모듈들과 분리할 경우 Host Controller의 변경에 따라 영향 받는 범위를 해당 패키지로 제한할 수 있다.

### CA\_52. Transfer type별 Interface 모듈 분리 구조 (채택)

### CA\_53. Transfer type 별 Interface 모듈 분리 구조 (채택 안함)

품질 요구 사항	영향	
	CA_52	CA_53

QA_05. 새로운 실행 환경(HW, OS) 지원 용이성	(++) 전송 type 별로 변경되는 범위를 각 type에 대한 처리 모듈로 제한할 수 있다.	(+) 전송 특징에 따라 변경 범위를 제한할 수 있다.
---------------------------------	--	--------------------------------

[채택 근거] Host Controller의 기능 중 HCI 버전에 따라 변경이 큰 부분은 전송 방식에 대한 것이다. 따라서 각 전송 방식에 따라 구현 모듈을 분리할 경우 HCI 변경에 따른 변경 범위를 제한할 수 있다. 특히, 각 전송 type별로 처리 모듈을 분리할 경우 4가지 전송 타입별로 변경에 대한 영향 범위를 각 모듈로 제한할 수 있는 이점이 있다.

#### CA\_54. Transaction Queue(Shared Memory) 관리 기능을 별도 모듈로 분리 (채택)

품질 요구 사항	영향(CA_54)
QA_05. 새로운 실행 환경(HW, OS) 지원 용이성	(+) Host Controller 별로 변경이 큰 Shared Memory 관리기능을 별도 모듈로 분리하여 영향 범위를 제한할 수 있다.

[채택 근거] HCI와 SW가 전송 데이터를 공유하는 Shared Memory 구조는 각 HCI 버전에 따라 차이가 발생한다. 따라서, 시스템 기능 중 Shared Memory를 관리하는 기능을 별도 모듈로 분리할 경우 시스템의 영향 범위를 해당 모듈로 제한할 수 있다.

## E8. 후보 구조 평가 결과

후보 구조에 대한 최종 평가 결과는 다음과 같다.

ID	후보구조	채택 결과
CA_1	자원 관리자를 통한 시스템 자원 관리 통합 구조	채택
CA_2	낭비되는 자원에 대한 garbage correction	채택
CA_3	자원할당 재시도	채택
CA_4	디바이스 매니저를 통한 디바이스 정보 통합 관리	채택
CA_5	Control 정보를 위한 Bandwidth Reservation	채택 안함
CA_6	Emulation 과정을 재시도	채택
CA_7	메모리 복사 기반의 전송 데이터 전달	채택 안함
CA_8	Ping/Echo 기반 시스템 health check	채택
CA_9	Heart bit 기반 시스템 health check	채택 안함
CA_10	Array 기반의 pipe 매핑 정보 관리	채택 안함
CA_11	List 기반의 pipe 매핑 정보 관리	채택 안함

CA_12	Cache 기반의 pipe 매핑 정보 관리	채택
CA_13	Periodic Transaction 처리를 위한 전용 Thread 운영 구조	채택 안함
CA_14	Interrupt 방식의 시스템<->Application간 이벤트 전달 구조	채택
CA_15	Polling 방식의 시스템<->Application간 이벤트 전달 구조	채택 안함
CA_16	디바이스 입력에 대한 Pre-fetch 구조	채택 안함
CA_17	데이터 Transfer와 Transaction을 단일 Thread에서 운영 하는 구조	채택
CA_18	데이터 Transfer와 Transaction을 개별 Thread에서 운영 하는 구조	채택 안함
CA_19	Transfer 요청에 대한 Thread 생성(Creation) 구조	채택 안함
CA_20	Transfer 요청 type별 Thread 할당(Dispatch) 구조	채택 안함
CA_21	Transfer 요청에 대한 Thread 임의 할당(Random Dispatch) 구조	채택
CA_22	Round Robin 방식의 Transfer 할당 방식	채택 안함
CA_23	Priority Queue 방식의 Transfer 할당 방식	채택
CA_24	On-demand 방식의 Transaction 생성	채택 안함
CA_25	Pre-generation 방식의 Transaction 생성	채택 안함
CA_26	Buffering 방식의 Transfer 생성 구조	채택
CA_27	단일 List 기반 Transfer 목록 관리 구조	채택 안함
CA_28	Transfer type별 Transfer 목록 관리 구조	채택
CA_29	FIFO 기반 Transfer scheduling	채택 안함
CA_30	Priority 기반 Transfer scheduling	채택
CA_31	단일 Message Queue 기반 컴포넌트간 메시지 전달 구조	채택 안함
CA_32	컴포넌트별 Message Queue 관리 구조	채택
CA_33	메모리 참조 기반의 전송 데이터 전달 구조	채택
CA_34	Tree 기반의 Configuration 정보 관리 구조	채택
CA_35	Tree 기반의 BUS topology 정보 관리 구조	채택
CA_36	데이터 Cache 관리 구조	채택 안함
CA_37	List 기반 디바이스 목록 관리	채택 안함
CA_38	Array 기반 디바이스 목록 관리	채택
CA_39	디바이스 연결 처리를 위한 독립 Thread 구조	채택
CA_40	디바이스 연결 요청 별 독립된 Thread 구조	채택 안함
CA_41	연결관리 객체의 Pre-Generation	채택
CA_42	연결관리 객체의 Lazy Clear	채택 안함
CA_43	가용 Address 목록 관리 구조(Heap)	채택 안함
CA_44	가용 Address 목록 관리 구조(List)	채택
CA_45	디바이스 Reconfiguration을 통한 자원 재 분배 구조	채택
CA_46	Delayed Resource Allocation 구조	채택 안함

CA_47	디바이스 정보 저장 자료 구조의 크기	채택
CA_48	USB Host Driver Interface 모듈 분리	채택
CA_49	디바이스 클래스별 Interface 적용	채택
CA_50	Layer별 OS 기능 추상화 Interface 구조	채택
CA_51	Host Controller 기능 사용 모듈 통합	채택
CA_52	Transfer type별 Interface 모듈 분리 구조	채택
CA_53	전송 특징(Async와 Periodic)에 따라 Interface 모듈 분리 구조	채택 안함

## F. 최종 구조 설계

### F1. 동작 측면의 최종 구조

최종 구조 설계에는 과제의 시스템에서 가장 우선 순위가 높은 품질 속성인 데이터 전송 성능을 향상시키기 위한 후보 구조를 우선적으로 고려하였다. 데이터 전송과 관련된 Task를 병렬 처리하기 위해 선택한 후보 구조 “CA\_17. 데이터 Transfer와 Transaction을 단일 Thread에서 운영 하는 구조”, “CA\_21. Transfer 요청에 대한 Thread 임의 할당(Random Dispatch) 구조”, “CA\_39. 디바이스 연결 처리를 위한 독립 Thread 구조” 와 각 Thread 사이의 메시지 전송 관리하기 위해 선택한 후보 구조 “CA\_32. 컴포넌트별 Message Queue 관리 구조”를 반영한 최종 후보 구조는 “**Message Buffer 기반의 Dispatcher 스타일**”로 다음 그림과 같다.

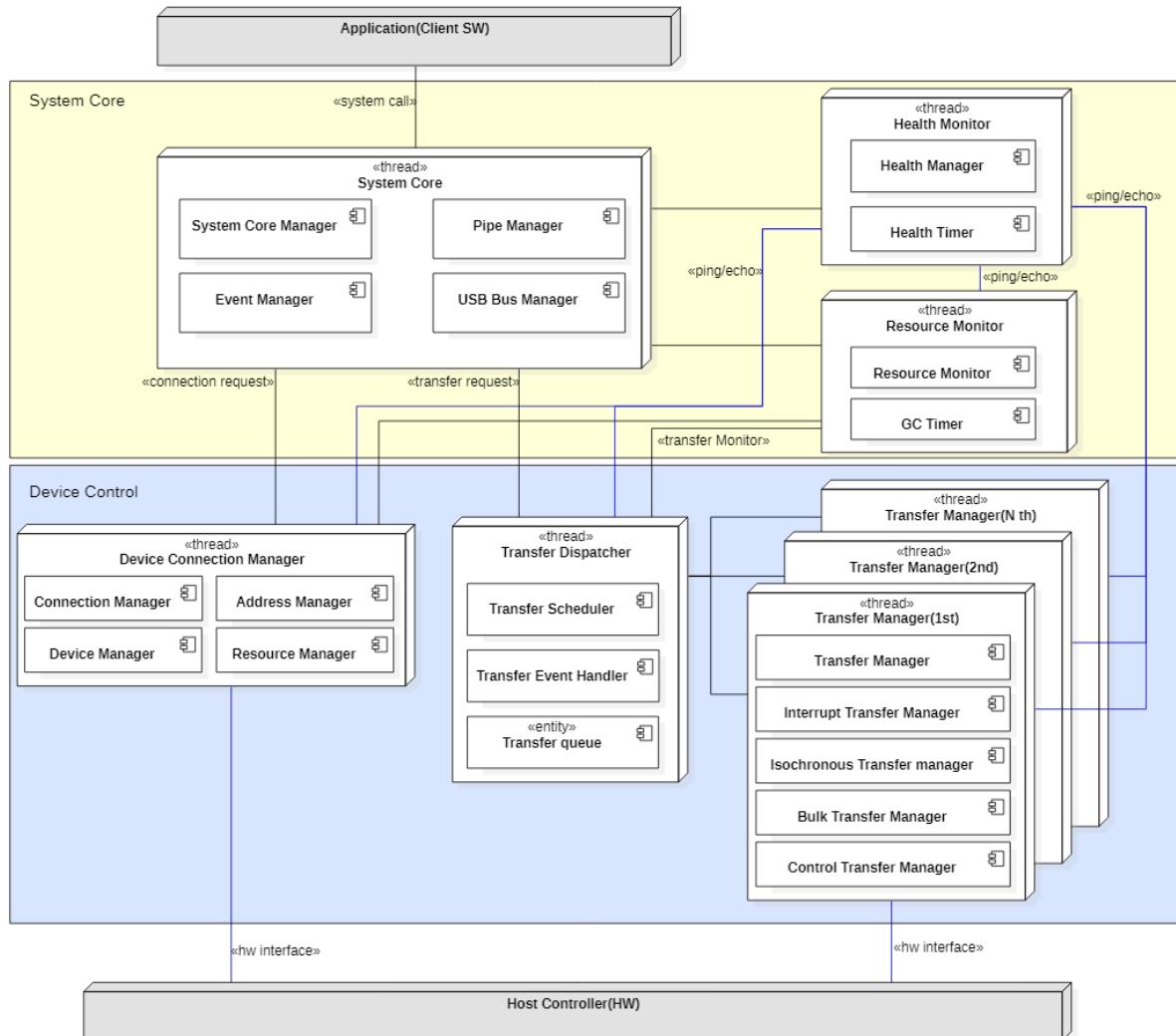


그림 102 시스템 Deployment View

동작 측면의 시스템 구조에 가장 큰 영향을 준 후보 구조는 “CA\_17. 데이터 Transfer와 Transaction을 단일 Thread에서 운영 하는 구조”와 “CA\_21. Transfer 요청에 대한 Thread 임의 할당(Random Dispatch) 구조”이다. 시스템은 각 Application에서 전달되는 데이터 전송을 최대한 병렬로 처리하기 위해 Transfer 요청을 각각의 Thread로 분산하여 처리하도록 하였다. 또한, Transfer를 처리하는 각 Thread의 Load를 관리하기 위한 별도의 Dispatcher Thread를 추가하여 전송 요청을 효율적으로 분배하도록 구조에 반영하였다. 병렬화로 인해 발생할 수 있는 공유 자원에 대한 동기화 문제 등을 보완하기 위해 후보 구조 “CA\_1. 자원 관리자를 통한 시스템 자원 관리 통합 구조”를 반영하여 시스템 자원을 관리하는 기능을 별도의 컴포넌트로 분리하였다. 시스템 자원 정보는 디바이스 연결 해제시 가장 빈번하게 사용되기 때문에 후보 구조 “CA\_39. 디바이스 연결 처리를 위한 독립 Thread 구조”와 USB Host Driver

통합하여 자원 관리 컴포넌트는 디바이스 연결을 처리하는 Thread에 배치하였다. 시스템 내부 동작 오류 및 외부 디바이스의 오류 등으로 이해 발생할 수 있는 자원의 낭비(leak)을 방지하기 위해 후보 구조 “CA2. 낭비되는 자원에 대한 garbage correction”를 반영하여 Resource Monitoring 기능을 적용하였다. 자원 모니터링을 위한 기능은 주기적인 동작이 필요하기 때문에 데이터 전송과 같은 시스템의 Main 기능에 영향을 최소화하기 위해 독립된 Thread로 배치하였다. 또한 각 Thread의 사용성을 높이기 위해 후보구조 “CA\_8. Ping/Echo 기반 시스템 health check”를 반영하여 Health Monitor 기능을 추가하였다. Health Monitor는 데이터 전송과 관련된 성능에 대한 영향을 최소화하기 위해 별도의 Thread로 동작하도록 구성하였다. 시스템을 구성하는 각 컴포넌트들 사이의 통신은 후보구조 “CA\_32. 컴포넌트별 Message Queue 관리 구조”를 반영하여 Message Buffer를 이용하여 컴포넌트 사이의 의존성을 줄이고 각 컴포넌트가 병렬적으로 동작할 수 있도록 구성하였다.

시스템의 각 구성요소에 반영된 후보 구조의 세부 내용은 다음과 같다.

#### F1.1. System Core Thread 구성과 관련된 후보 구조

- CA\_12. Cache 기반의 pipe 매핑 정보 관리**
- CA\_14. interrupt 방식의 시스템<->Application간 이벤트 전달 구조**
- CA\_23. 컴포넌트별 Message Queue 관리 구조**
- CA\_31. Tree 기반의 BUS topology 정보 관리 구조**
- CA\_41. 디바이스 Reconfiguration을 통한 자원 재 분배 구조**
- CA\_44. USB Host Driver Interface 기능 분리**
- CA\_45. 디바이스 클래스별 Interface 적용**

System Core Thread는 시스템의 메인 Thread 기능을 하며 Application에 대한 USB Host Driver Interface를 제공한다. Pipe 매핑 정보 관리와 USB BUS topology에 대한 정보는 시스템 내부에서 전체적으로 공유되는 정보이기 때문에 System core thread에서 관리하도록 배치하였다. 특히, Pipe 매핑 정보의 경우 Application 측면에서는 USB 디바이스와 통신하는 interface 역할을 하기 때문에 Application에 대한 interface 기능을 제공하는 System core thread에서 처리하는 것이 효율적이라 판단하였다. 추가적으로 Host Controller를 포함하여 시스템 외부에서 발생하는 다양한 interrupt에 대한 Handler를 처리할 수 있는 기능과, Application으로 Interrupt를 발생 시킬 수 있는 Interrupt manager 기능도 System core thread에 배치하였다.

#### F1.2. Device Connection Manager Thread 구성과 관련된 후보 구조

- CA\_1. 자원 관리자를 통한 시스템 자원 관리 통합 구조**

**CA\_4. 디바이스 매니저를 통한 디바이스 정보 통합 관리****CA\_30. Tree 기반의 Configuration 정보 관리 구조****CA\_35. 디바이스 연결 처리를 위한 독립 Thread 구조****CA\_41. 디바이스 Reconfiguration을 통한 자원 재 분배 구조**

Device connection Manager Thread는 디바이스 연결과 관련된 기능을 전담하는 Thread이다. 디바이스 연결을 위해 필요한 USB 리소스 정보, 디바이스 정보에 대한 접근 시간은 “QA\_2 디바이스의 연결 /해제시 인식 지연 시간”에 직접 영향을 주는 요소로 판단하여 Device Connection Manager Thread에서 직접 관리하도록 배치하였다. “**CA\_30. Tree 기반의 Configuration 정보 관리 구조**”는 별도의 Configuration Manager를 적용하는 대신 Device Manager에서 통합 관리하는 것이 효율적이라 판단하여 통합하였다.

**F1.3. Transfer Dispatcher Thread 구성과 관련된 후보 구조****CA\_21. Transfer 요청에 대한 Thread 임의 할당(Random Dispatch) 구조****CA\_23. Priority Queue 방식의 Transfer 할당 방식****CA\_30. Priority 기반 Transaction scheduling**

시스템은 후보구조 “**CA\_21. Transfer 요청에 대한 Thread 임의 할당(Random Dispatch) 구조**”를 반영하여 Transfer를 처리하는 N개의 Thread를 생성하도록 구성되었다. 각 Transfer Thread에 요청되는 전송의 수행 시간은 각기 다르기 때문에 Transfer 요청에 대한 scheduling이 필요하기 때문에, 각 Thread의 Load를 효율적으로 관리하기 위한 별도의 Dispatcher Thread를 추가하였다.

**F1.4. Transfer Manager Thread 구성과 관련된 후보 구조****CA\_17. 데이터 Transfer와 Transaction을 단일 Thread에서 운영 하는 구조****CA\_21. Transfer 요청에 대한 Thread 임의 할당(Random Dispatch) 구조****CA\_23. Priority Queue 방식의 Transfer 할당 방식****CA\_26. Buffering 방식의 Transaction 생성 구조**

시스템은 후보구조 “**CA\_21. Transfer 요청에 대한 Thread 임의 할당(Random Dispatch) 구조**”를 반영하여 Transfer를 처리하는 N개의 Thread를 생성하도록 구성되었다. 각 Transfer Thread에 요청되는 전송의 수행 시간은 각기 다르기 때문에 Transfer 요청에 대한 scheduling이 필요하기 때문에, 각 Thread의 Load를 효율적으로 관리하기 위한 별도의 Dispatcher Thread를 추가하였다.

**F1.5. Health Monitor Thread 구성과 관련된 후보 구조****CA\_8. Ping/Echo 기반 시스템 health check**

시스템내부의 각 컴포넌트의 정상 동작 여부를 모니터링 하기 위한 Health Monitor는 ping/echo 기

반으로 동작하도록 구성하였다. 과도한 ping/echo 유발을 막기 위한 ping/echo 주기 설정과, ping에 대한 echo 메시지의 timeout을 설정하기 위한 별도의 timer 컴포넌트를 추가하였다. 또한, Health check 기능은 주기적 동작이 필요하기 때문에 시스템 내부의 다른 컴포넌트의 동작에 영향을 주지 않고 병렬로 동작 할 수 있도록 별도의 thread로 구성하였다.

### F1.6. Resource Monitor thread 구성과 관련된 후보 구조

#### **CA\_2 낭비되는 자원에 대한 garbage correction**

Resource Garbage Corrector는 USB Bus system에서 디바이스 오동작 등으로 인해 낭비되는 자원을 회수하는 기능을 한다. 이를 위해 주기적으로 Transfer Manager에서 전송중인 정보를 수집하여 디바이스별로 할당된 자원 상황과 비교하기 위해 Transactor, Device Connector Manager와 통신할 수 있도록 구성하였다. 또한, 주기적인 모니터링을 위한 Timer 기능도 함께 적용하였다.

### F1.7. 시스템 메시지 교환과 관련된 후보 구조

#### **CA\_23. 컴포넌트별 Message Queue 관리 구조**

#### **CA\_29. 메모리 참조 기반의 전송 데이터 전달 구조**

시스템 내부의 메시지 교환은 각 컴포넌트 내부의 메시지 큐를 이용하여 메시지를 전달하는 구조를 적용하였다. 이를 위해 시스템 내부의 각 Thread는 독립된 메시지 큐를 가지며, 메시지 큐에 전달된 요청을 바탕으로 각각 병렬적으로 동작하도록 구성하였다. 또한 크기가 가변적인 전송데이터도 데이터가 저장된 위치에 대한 메모리 참조 방식으로 전달하기 때문에 메시지 큐의 크기를 일정한 크기로 구성할 수 있다.

## F2. 개발 측면의 최종 구조

개발 측면의 시스템 구조에는 주로 유지보수성과 관련된 후보 구조들을 우선적으로 고려하였다. 새로운 디바이스 클래스 추가로 인한 변경 영향을 최소화하기 위해 선택한 후보 구조 “**CA\_48. USB Host Driver Interface 모듈 분리**”, “**CA\_49. 디바이스 클래스별 Interface 적용**” 와 실행 환경(OS, HW)의 변경에 대한 영향을 최소화 하기 위해 선택한 후보 구조 “**CA\_50. Layer별 OS 기능 추상화 Interface 구조**”, “**CA\_52. Transfer type별 Interface 모듈 분리 구조**”를 반영한 최종 후보 구조는 “**Layer 스타일**”의 “**2-Layer 구조**”로 다음 그림과 같다.

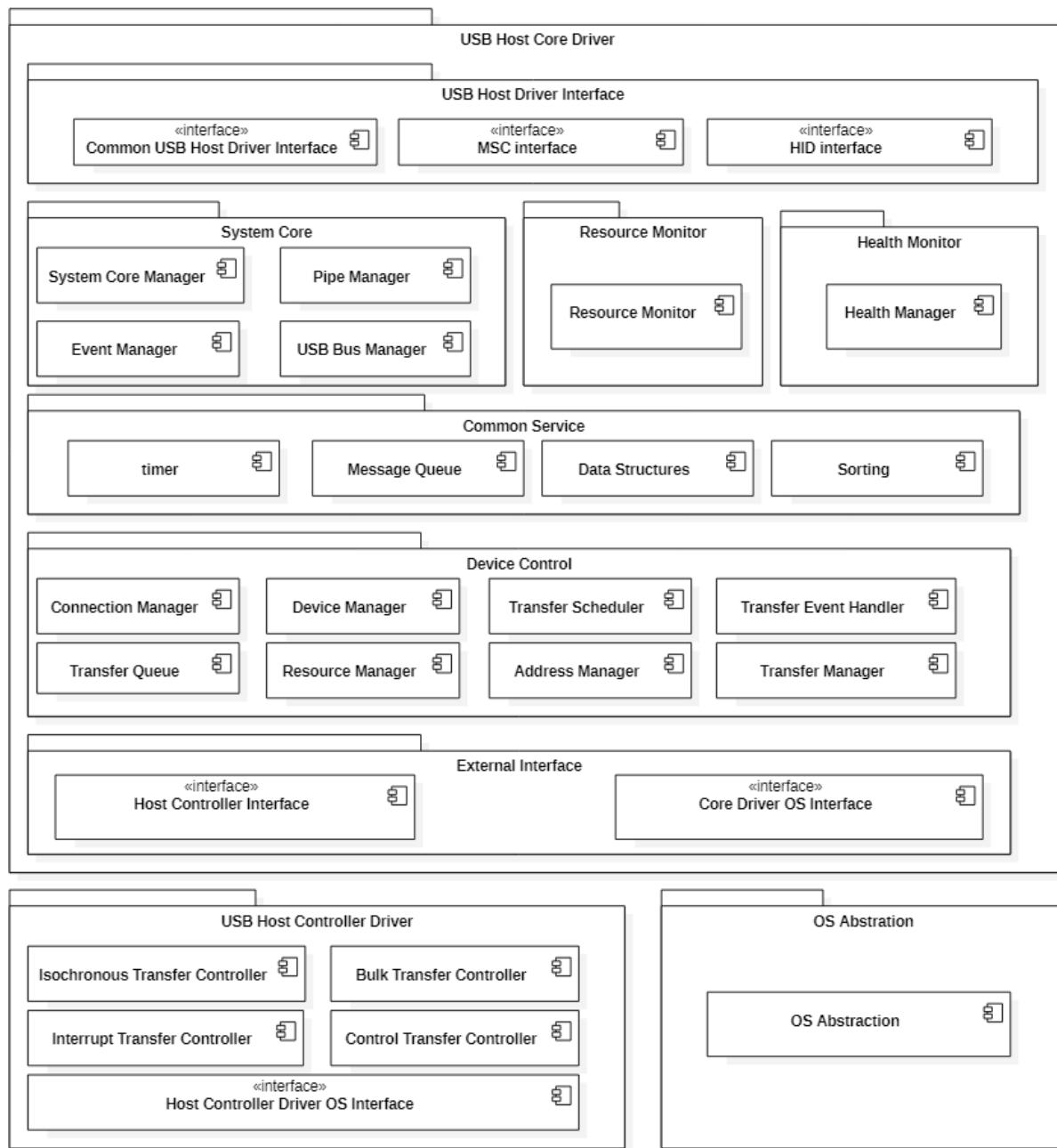


그림 103 시스템 Module View

구현 측면의 시스템 구조에 가장 큰 영향을 준 후보 구조는 “**CA\_48. USB Host Driver Interface 모듈 분리**”와 “**CA\_52. Transfer type별 Interface 모듈 분리 구조**”, “**CA\_50. Layer별 OS 기능 추상화 Interface 구조**”이다. 시스템을 외부 시스템의 영향으로부터 최대한 분리하기 위하여 Application과 통신하는 부분과 HW와 통신하는 부분을 각각의 Interface 모듈로 분리하고, 외부 시스템에서는 해당 Interface를 사용하여 기능을 구현하도록 하였다. 또한, OS와 관련된 기능을 별도의 Interface로 분리

하여 OS 변경에 따른 영향성을 최소화 할 수 있도록 하였다. OS 기능의 경우 Host Core Driver Layer 와 Host controller Driver 모두 사용하기 때문에 각 Layer가 독립적인 Interface를 패키지 내에 정의하고, OS Abstraction 패키지에서 다중 상속하는 형태로 설계 하였다.

### F3. 최종 구조의 단점/Risk 분석

#### F3.1. 전송 타입에 따른 고정 우선 순위 방식

선정된 최종 구조에서는 데이터 전송 시간 지연을 최소화하기 위해 Transfer의 우선순위를 전송 타입에 따른 고정된 선정 방식을 채택하였다. 이 방식의 경우 Transfer의 우선 순위를 정하기 위한 Sorting 시간을 줄여 전송 시간을 줄이는 이점이 있지만, 우선 순위가 높은 Transfer 요청이 계속해서 전달되는 경우 Bulk 데이터 전송과 같은 우선 순위가 낮은 Transfer는 Starvation이 발생할 가능성이 존재한다. 우선 순위가 높은 Transfer는 경우 비교적 짧은 데이터 전송으로 처리시간이 짧아 우선 순위가 낮은 작업이 지속적으로 처리되지 않을 가능성이 높지는 않지만, 시스템이 동작하는 실제 환경에서 Starvation이 발생하는 경우 우선 순위 선정 알고리즘의 변경이 필요할 수 있다. 따라서 이러한 상황이 발생할 경우 우선 순위 선정 알고리즘의 교체가 용이하도록 Transfer Scheduler Module 구현시 고려하도록 한다. 전략 패턴 등을 이용하여 알고리즘 교체가 용이하도록 구현하는 방식 등을 고려할 수 있다.

#### F3.2. 공유 자원 접근 성능

시스템에서 각 Thread간에 가장 빈번하게 접근되는 공유 자원은 Configuration 정보를 포함한 디바이스 정보와 USB Bus의 자원에 대한 정보이다. 최종 후보 구조에서는 시스템이 동작하는 환경을 고려하여 해당 정보를 가장 빈번하게 접근할 것으로 예상되는 Device connection manager thread에서 해당 정보를 관리하게 함으로써 디바이스 연결/해제시 성능을 향상시킬 수 있도록 하였다. 하지만 실제 동작 환경에서 해당 정보를 자주 접근하는 Thread가 System Core thread 등과 같은 다른 Thread 가 된다면, 해당 정보를 접근하기 위해 Thread간 빈번한 메시지 교환으로 인한 overhead가 발생할 수 있다. 실제 동작 환경에서 해당 정보를 자주 사용하는 Thread를 모니터링 하여 해당 정보에 대한 관리 기능을 재배할 필요도 있다. 따라서, Resource Manager와 Device Manager기능 구현시 Thread가 재배치가 용이하도록 가능한 독립된 모듈로 구현하도록 한다.

### F3.3. 메모리 참조 방식의 데이터 전달

최종 구조에서는 시스템이 외부와 데이터를 전달하는 방식으로 메모리 참조 방식을 채택하였다. 메모리 참조 방식은 외부 시스템에서 전달된 데이터에 대한 메모리 복사를 줄이기 때문에 성능 측면에서는 큰 이점이 있다. 하지만, 공유된 메모리가 시스템이 참조하는 동안 변경되는 경우 시스템에 예기치 않은 오류가 발생할 가능성이 있다. 시스템에서 메모리 참조로 접근하는 외부 메모리는 대부분 전송 데이터로 시스템의 동작에 영향을 줄 가능성은 있지만, 시스템의 안정성을 위해 해당 메모리에 대한 접근을 데이터를 사용하는 Application과 본 시스템으로 제한할 필요가 있다. 시스템 외부적으로 OS에서 제공하는 Memory Access Control(MMU등 활용) 기능과 메모리 Sanitizer 기능 등을 활용하여 비정상적인 메모리 접근에 대한 Risk를 보완할 수 있다.