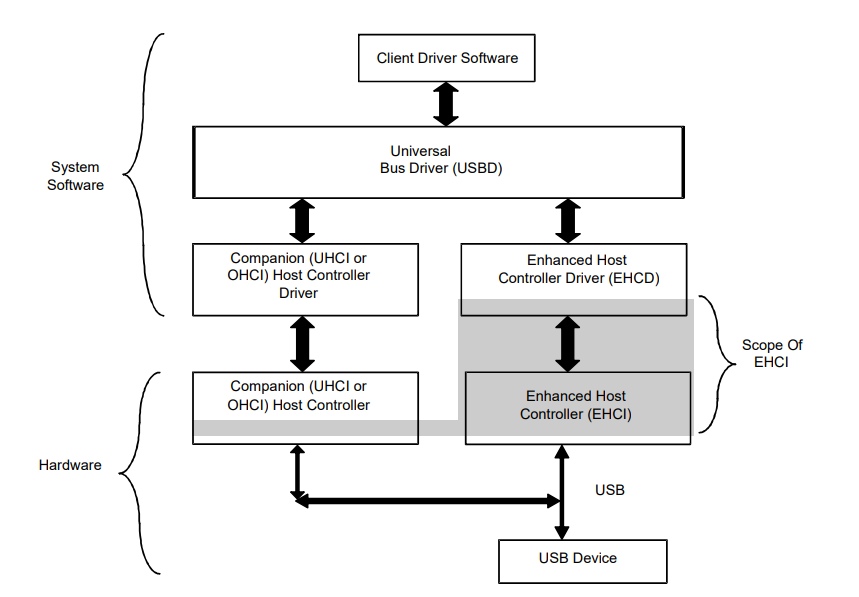
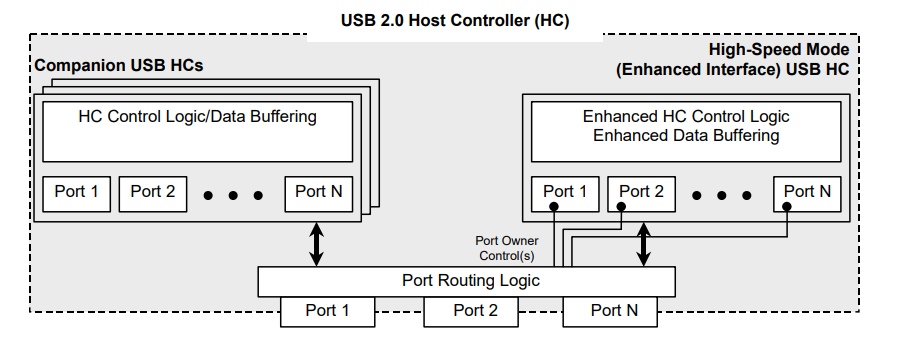
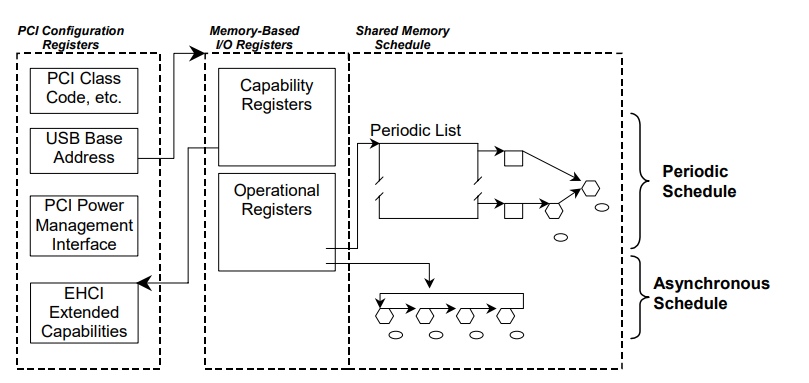
**USB**

**https://habr.com/ru/post/426421/**

**Введение**

Всех приветствую. Сегодня хочу поделиться опытом и всё-таки по-моему внятно объяснить про такой, на первый взгляд, простой стандарт для USB 2.0 хост-контроллера.  
  
Изначально можно представить себе что USB 2.0 порт — это всего лишь 4 пина, по двум из которых просто передаются данные(Как, к примеру, COM-порт), но самом деле всё не так, и даже совсем наоборот. USB-контроллер в принципе не даёт нам возможности передавать данные как через обычный COM-порт. EHCI — довольно замысловатый стандарт, который позволяет обеспечить надежную и быструю передачу данных от софта до самого девайса, и в обратную сторону.  
  
Возможно, вам пригодиться эта статья, если, к примеру, вы не имеете достаточных навыков написания драйверов и чтение документации к хардвейру. Простой пример: хотите написать свою ОС для мини-ПК, дабы какая-нибудь винда или очередной дистрибутив линукса не загружали железо, и вы использовали всю его мощь исключительно в своих целях.

**Что такое EHCI**

Что же, давайте начнем. EHCI — Enhanced Host Controller Interface, предназначен для передачи данных и управляющих запросов USB-устройствам, и в другую сторону, а в 99% случаев — является связующим звеном, между каким-либо софтом и физическим устройством. EHCI работает как PCI-устройство, а соответственно использует MMIO(Memory-Mapped-IO) для управления контроллером(да-да, я знаю, что некоторые PCI-девайсы используют порты, но тут я всё обобщил). В документации от Intel описан лишь принцип работы, и никаких намеков на алгоритмы, написанные хотя бы на псевдокоде, нет вовсе. EHCI имеет 2 типа MMIO-регистров: Capability и Operational. Первые служат для получения характеристик контроллера, вторые же — для его управления. Собственно, прикреплю саму суть связи софта и EHCI контроллера:  
  
  
  
Каждый EHCI контроллер имеет несколько портов, каждому из которых могут быть подключены какие-либо USB-устройства. Так же, прошу заметить, что EHCI является улучшенной версией UHCI, который так же был разработан Intel на несколько годов раньше. Для обратной совместимости любой UHCI/OHCI контроллер, который имеет версию ниже, чем EHCI, будет компаньоном к EHCI. К примеру, у вас есть USB-клавиатура(А большинство клавиатур года так до сих пор были именно такими), которая работает на USB 1.1(заметим, что максимальная скорость работы USB 1.1 — 12 мегабит в секунду, а FullSpeed USB 2.0 имеет пропускную способность аж в 480 мбит/сек), а у Вас имеется компьютер с USB 2.0 портом, при подключении клавиатуры к компьютеру хост-контроллер EHCI как ни как будет работать с USB 1.1. Данная модель показана на следующей схеме:  
  
  
  
Так же на будущее хочу сразу предупредить, что Ваш драйвер может работать не правильно из-за такой вот нелепой ситуации: вы инициализировали UHCI, а после чего EHCI, при этом добавили два одинаковых устройства, поставили в регистр порта бит Port Owner Control, после чего UHCI перестал работать, из-за того, что EHCI автоматически перетягивает порт на себя, а порт на UHCI перестаёт откликаться, эту ситуацию надо отслеживать.  
  
Так же, давайте рассмотрим схему, показывающую саму архитектуру EHCI:  
  
  
  
Справа написано про очереди — о них чуть позже.

Регистры EHCI контроллера

Для начала хочется еще раз уточнить, что через данные регистры вы будете управлять вашим устройством, поэтому они очень важны — да и без них программирование EHCI невозможно.  
  
Для начала вам надо получить адрес MMIO, который выдан данному контроллеру, по смещению +0x10 будет лежать адрес наших долгожданных регистров. Есть одно но: сначала идут Capability регистры, а только после них — Operational, поэтому по смещению 0(от предыдущего адреса, который мы получили по смещению 0x10 относительно начала MMIO нашего EHCI) лежит один байт — длина Capability-регистров.

**Capability регистры**

По смещению 2 лежит регистр **HCIVERSION** — номер ревизии данного HC, который занимает 2 байта и содержит BCD версию ревизии (что такое BCD можно узнать из википедии).  
По смещению +4 лежит регистр **HCSPARAMS**, его размер — 2 слова, он содержит структурные параметры устройства и его биты показывают следующее:

* Бит 16 — Port Indicators — доступные световые индикаторы для подключенных USB-устройств.
* Биты 15:12 — номер контроллера-компаньона, который присвоен данному контроллеру
* Биты 11:8 — количество портов у компаньон-контроллера
* Бит 7 — Port Routing Rules — показывает, как данные порты привязаны к компаньон-портам
* Бит 4 — Port Power Control — показывает, надо ли включать питание каждому порту, 0 — питание подаётся автоматически
* Биты 3:0 — количество портов у данного контроллера.
* По смещению +8 лежит регистр HCCPARAMS — показывает параметры совместимости, его биты значат следующее:
* Бит 2 — доступность асинхронной очереди,
* Бит 1 — доступность периодической (последовательной) очереди
* Бит 0 — 64-битная совместимость

**Operation регистры**

По смещению 0 лежит регистр **USBCMD**— командный регистр контроллера, его биты означают следующее:

* Биты 23:16 — Interrupt Threshold Control — показывает сколько микро-фреймов будет использоваться на один обычный фрейм. Чем больше, тем быстрее, но если больше 8 — то микро-фреймы будут обрабатываться с той же скоростью, что и для 8.
* Бит 6 — прерывание после каждой транзакции в асинхронной очереди,
* Бит 5 — используется ли асинхронная очередь,
* Бит 4 — использование последовательной очереди,
* Биты 3:2 — размер FrameList'a (о этом — дальше). 0 означает 1024 элемента, 1 — 512, 2 — 256, 3 — зарезервировано
* Бит 1 — устанавливается для выполнение сброса хост-контроллера.
* Бит 0 — Run/Stop

.  
Далее, по смещению +4 идет регистр **USBSTS**— статут хост-контроллера,

* Бит 15 показывает используется ли асинхронная очередь
* Бит 14 показывает используется ли последовательная очередь,
* Бит 13 — показывает, что обнаружена пустая асинхронная очередь,
* Бит 12 установлен в 1, если при обработке транзакции произошла ошибка, тогда хост-контроллер остановит выполнение всех очередей.
* Бит 4 установлен в 1, если произошла серьезная ошибка, хост-контроллер останавливает выполнение всех очередей.
* Бит 3 FrameList (Регистр) Rollover — ставится в 1, когда хост-контроллер обработал весь frameList.
* Бит 1 — USB Error Interrupt — генерировать ли прерывание при ошибках?
* Бит 0 — USB Interrupt — выставляется после успешной обработки транзакции, если в TD был установлен IOC

Не устали? Можете налить себе крепкого чайку и принести печенок, мы еще в самом начале!  
  
По смещению +8 лежит регистр **USBINTR**— регистр включения прерываний  
Чтобы долго не писать, и тем более, Вам долго не читать, значения битов данного регистра можно посмотреть в спецификации, ссылка на неё будет оставлена внизу. Сюда я просто записываю 0, т.к. абсолютно не имею желания писать обработчики, мапить прерывания и т.п., так что это я считаю почти что абсолютно бессмысленным.  
  
По смещению +12(0x0C) лежит регистр **FRINDEX**, в котором просто лежит текущий номер фрейма, при чем, хочу заметить, что последние 4 бита показывают номер микро-фрейма, в старшие 28 — номер фрейма (так же значение не обязательно меньше размера frameList'а, если вам нужен индекс — лучше брать его с маской 0x3FF(или же 0x1FF, и т.п.).  
  
Регистр **CTRLDSSEGMENT**лежит по смещению +0x10, он показывает хост-контроллеру старшие 32 бита адреса листа фреймов.  
  
Регистр **PERIODICLISTBASE**имеет смещение +0x14, в него вы можете положить младшие 32 бита листа фреймов, заметим, что адрес должен быть выравнен по размеру страницы памяти (4096).  
  
Регистр **ASYNCLISTADDR**имеет смещение +0x18, в него вы можете положить адрес асинхронной очереди, заметим, что он должен быть выравнен по границе 32 байта, при этом должен находиться в первых четырех гигабайтах физической памяти.  
  
Регистр **CONFIGFLAG**показывает, настроено ли устройство. Вы должны выставить бит 0 после завершения настройки устройства, он имеет смещение +0x40.  
  
Перейдем к регистрам портов. Каждый порт имеет свой командно-статусный регистр, каждый регистр порта располагается со смещением **+0x44 + (PortNumber — 1)\*4**, его биты значат следующее:

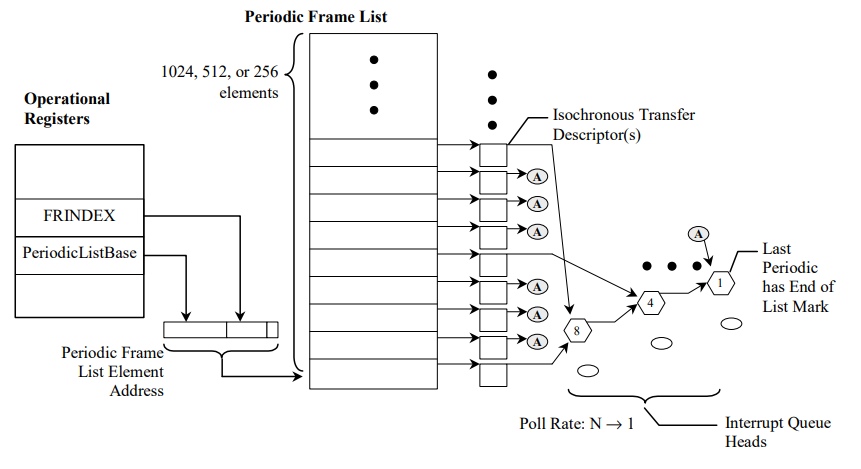
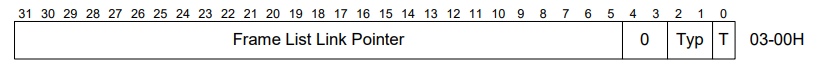
* Бит 12 — питание порта, 1 — питание подаётся, 0 — нет.
* Бит 8 — Port Rest — устанавливается для сброса устройства.
* Бит 3 — Port Enable/Disable Change — выставляется при изменении статуса «включенности» порта.
* Бит 2 — порт включен/не включен.
* Бит 1 — Изменение статуса подключения, ставится в 1, к примеру, если вы подключили, или отключили USB устройство.
* Бит 0 — статус подключения, 1 — подключено, 0 — нет.

Теперь перейдем к самому соку.

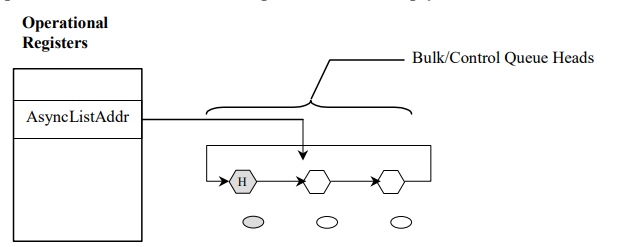
Структуры передачи данных и запросов

Организация структуры для обработки запросов включает в себя очередь и трансфер дескрипторы(TDs).  
  
На данный момент мы рассмотрим только 3 структуры.

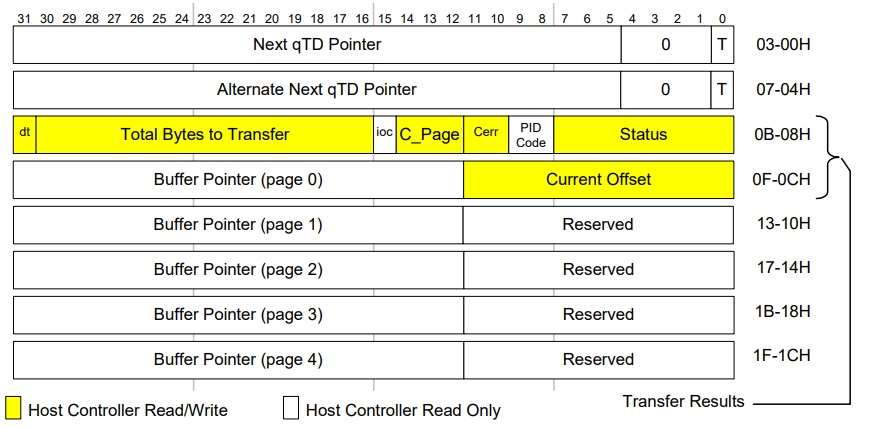
Последовательный список

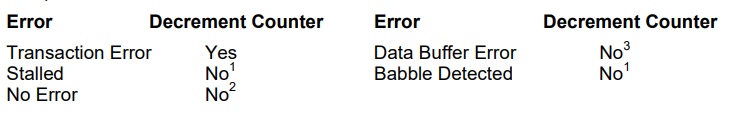
Последовательный(Периодичный, Pereodic) список устроен следующим образом:  
  
  
  
Как видно на схеме, обработка начинается с получения нужного фрейма из фрейм листа, каждый его элемент занимает 4 байта и имеет следующую структуру:  
  
  
  
Как видно на картинке, адрес очереди/трансфер дескриптора выровнен по границе 32 байта, бит 0 означает то, что хост-контроллер не будет обрабатывать данный элемент, биты 3:1 показывают тип того, что будет обрабатывать хост-контроллер: 0 — изосинхронный TD(iTD), 1 — очередь, 2 и 3 в данной статье я рассматривать не буду.

Асинхронная очередь

Хост контроллер обрабатывает данную очередь только тогда, когда фрейм последовательный пустой, либо хост-контроллер обработал весь последовательный список.  
  
Асинхронная очередь представляет собой указатель на очередь, где содержатся другие очереди, которые нуждаются в обработке. Схема:  
  


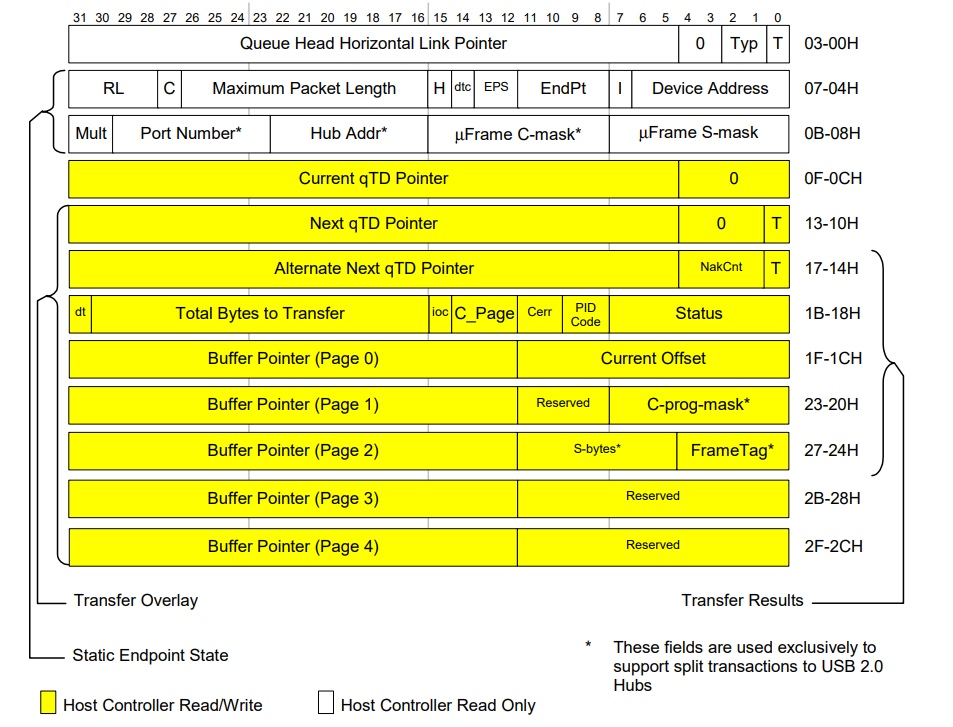
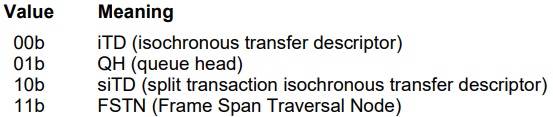
qTD(Queue Element Transfer Descriptor)

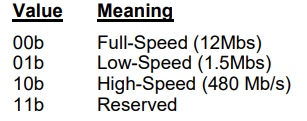
Данный TD имеет следующую структуру:  
  
  
  
**Next qTD Pointer** — указатель на продолжение очереди для обработки(для Horizontal Execution), бит 0 Next qTD Pointer'а показывает, то, что дальше нет еще одной очереди.  
**qTD Token** — токен TD, показывает параметры передачи данных:

* Бит 31 — Data Toggle (об этом дальше)
* Биты 30:16 — количество данных для передачи, после завершения транзакции их значение уменьшается на количество переданных данных.
* Бит 15 — IOC — Interrupt On Complete — вызвать прерывание после завершения обработки дескриптора.
* Биты 14:12 показывают номер текущего буфера, в который/из которого производиться обмен данными, об этом далее.
* Биты 11:10 — допустимое количество ошибок. Данная таблица показывает, когда счетчик количества ошибок уменьшается:  
    
    
    
  Сноска 1 — обнаружение Babble либо Stall автоматически останавливает выполнение головы очереди. Сноска 3 — Ошибки буфера данных — это проблемы с хостом. Они не учитывают повторные попытки устройства.
* 9:8 — PID Code — тип токена: 0 — токен на вход(от хоста к устройству), 1 — токен на выход(от устройства к хосту), 2 — «SETUP» токен
* Биты 7:0 показывают статус TD:  
  Бит 7 показывает, что данный TD имеет активное состояние(т.е. хост-контроллер обрабатывает данный TD)  
  Бит 6 — Halted — показывает, что произошла какая-либо ошибка и выполнение TD остановлено.  
  Бит 4 — Babble Detected — количество данных, которые мы отправили устройству, или на оборот, меньше, чем мы передаём, т.е., к примеру, нам устройство отправило 100 байт данных, а мы читаем только 50 байт, а потом еще 50. Бит Halted так же будет установлен, если данный бит установлен в 1.  
  Бит 3 — Transaction Error — произошла ошибка во время проведения транзакции.

**qTD Buffer Page Pointer List** — любой из 5 буферов. Содержит ссылку на то, куда в памяти производить транзакцию(отправить данные устройству/принять данные с устройства), все адреса в буферах, кроме первого, должны быть выровнены по размеру страницы (4096 байт).

Голова очереди

Голова очереди(Queue Head) имеет следующую структуру:  
  
  
  
**Queue Head Horizontal Link Pointer** — указатель на следующую очередь, биты 2:1 имеют следующие значения в зависимости от типа очереди:  
  
  
  
**Endpoint Capabilities/Characteristics** — характеристики очереди:

* Биты 26:16 содержат максимальный размер пакета для передачи
* Бит 14: Data Toggle Control — показывает, где хост-контроллер должен брать изначальное значение Data Toggle, 0 — игнорирует бит DT в qTD, сохраняет бит DT для головы очереди.
* 
* Бит 13:12— характеристики скорости передачи:
* Биты 11:8 — номер конечной точки, к которой выполняется запрос
* Биты 6:0 — адрес устройства  
  **Endpoint Capabilities: Queue Head DWord 2** — продолжение предыдущего двойного слова:
* Биты 29:23 — номер Хаба
* Биты 22:16 — адрес Хаба

**Current qTD Link Pointer** — указатель на текущий qTD.  
  
Переходим к самому интересному.

**Драйвер EHCI**

Начнем с того, какие запросы может выполнять EHCI. Есть 2 типа запросов: Control — а-ля команд, и Bulk — к конечным точкам, для обмена данными, к примеру, абсолютное большинство флешек (USB MassStorage) использует тип передачи данных Bulk/Bulk/Bulk. Мышь и клавиатура для передачи данных тоже используют Bulk — запросы.  
  
Инициализируем EHCI и настраиваем асинхронную и последовательные очереди:

// Base I/O Address

PciBar bar;

PciGetBar(&bar, id, 0);

EhciController \*hc = VMAlloc(sizeof(EhciController));

hc->capRegs = (EhciCapRegs \*)(uintptr\_t)bar.u.address;

hc->opRegs = (EhciOpRegs \*)(uintptr\_t)(bar.u.address + hc->capRegs->capLength);

// Read the Command register

// Читаем командный регистр

uint cmd = ROR(usbCmdO);

// Write it back, setting bit 2 (the Reset bit)

// Записываем его обратно, выставляя бит 2(Reset)

// and making sure the two schedule Enable bits are clear.

// и проверяем, что 2 очереди выключены

WOR(usbCmdO, 2 | cmd & ~(CMD\_ASE | CMD\_PSE));

// A small delay here would be good. You don't want to read

// Небольшая задержка здесь будет неплоха, Вы не должны читать

// the register before it has a chance to actually set the bit

// регистр перед тем, как у него не появится шанса выставить бит

ROR(usbCmdO);

// Now wait for the controller to clear the reset bit.

// Ждем пока контроллер сбросит бит Reset

while (ROR(usbCmdO) & 2);

// Again, a small delay here would be good to allow the

// reset to actually become complete.

// Опять задержка

ROR(usbCmdO);

// wait for the halted bit to become set

// Ждем пока бит Halted не будет выставлен

while (!(ROR(usbStsO) & STS\_HCHALTED));

// Выделяем и выравниваем фрейм лист, пул для очередей и пул для дескрипторов

// Замечу, что все мои дескрипторы и элементы очереди выравнены на границу 128 байт

hc->frameList = (u32 \*)VMAlloc(1024 \* sizeof(u32) + 8192 \* 4);

hc->frameList = (((uint)hc->frameList) / 16384) \* 16384 + 16384;

hc->qhPool = (EhciQH \*)VMAlloc(sizeof(EhciQH) \* MAX\_QH + 8192 \* 4);

hc->tdPool = (EhciTD \*)VMAlloc(sizeof(EhciTD) \* MAX\_TD + 8192 \* 4);

hc->qhPool = (((uint)hc->qhPool) / 16384) \* 16384 + 16384;

hc->tdPool = (((uint)hc->tdPool) / 16384) \* 16384 + 16384;

// Asynchronous queue setup

// Инициализируем асинхронную очередь

EhciQH \*qh = EhciAllocQH(hc);

// Это указатель на нашу очередь, она у нас будет одна

// указываем, что это очередь

qh->qhlp = (u32)(uintptr\_t)qh | PTR\_QH;

// устанавливаем бит, который показывает, что это Голова очереди

qh->ch = QH\_CH\_H;

qh->caps = 0;

qh->curLink = 0;

qh->nextLink = PTR\_TERMINATE;

qh->altLink = 0;

qh->token = 0;

// Заполняем буферы нулями

for (uint i = 0; i < 5; ++i)

{

qh->buffer[i] = 0;

qh->extBuffer[i] = 0;

}

hc->asyncQH = qh;

// Periodic list queue setup

// Инициализируем последовательную очередь

qh = EhciAllocQH(hc);

// Мы ничего не делаем

qh->qhlp = PTR\_TERMINATE;

qh->ch = 0;

qh->caps = 0;

qh->curLink = 0;

qh->nextLink = PTR\_TERMINATE;

qh->altLink = 0;

qh->token = 0;

// Заполняем буферы

for (uint i = 0; i < 5; ++i)

{

qh->buffer[i] = 0;

qh->extBuffer[i] = 0;

}

qh->transfer = 0;

qh->qhLink.prev = &qh->qhLink;

qh->qhLink.next = &qh->qhLink;

hc->periodicQH = qh;

// Заполняем фреймлист ссылками на нашу последовательную очередь

for (uint i = 0; i < 1024; ++i)

hc->frameList[i] = PTR\_QH | (u32)(uintptr\_t)qh;

kprintf("FrameList filled. Turning off Legacy BIOS support...");

// Check extended capabilities

// Отключаем BIOS Legacy support

uint eecp = (RCR(hccParamsO) & HCCPARAMS\_EECP\_MASK) >> HCCPARAMS\_EECP\_SHIFT;

if (eecp >= 0x40)

{

// Disable BIOS legacy support

uint legsup = PciRead32(id, eecp + USBLEGSUP);

kprintf(".");

if (legsup & USBLEGSUP\_HC\_BIOS)

{

PciWrite32(id, eecp + USBLEGSUP, legsup | USBLEGSUP\_HC\_OS); kprintf(".");

for (;;)

{

legsup = PciRead32(id, eecp + USBLEGSUP);

kprintf(".");

if (~legsup & USBLEGSUP\_HC\_BIOS && legsup & USBLEGSUP\_HC\_OS)

{

break;

}

}

}

}

kprintf("Done\n");

// Disable interrupts

// Отключаем прерывания

//hc->opRegs->usbIntr = 0;

MWIR(ehcibase, usbIntrO, 0);

// Setup frame list

// Устанавливаем ссылку на фреймлист

//hc->opRegs->frameIndex = 0;

WOR(frameIndexO, 0);

//hc->opRegs->periodicListBase = (u32)(uintptr\_t)hc->frameList;

WOR(periodicListBaseO, (u32)(uintptr\_t)hc->frameList);

// копируем адрес асинхронной очереди в регистр

//hc->opRegs->asyncListAddr = (u32)(uintptr\_t)hc->asyncQH;

WOR(asyncListAddrO, (u32)(uintptr\_t)hc->asyncQH);

// Устанавливаем сегмент в 0

//hc->opRegs->ctrlDsSegment = 0;

WOR(ctrlDsSegmentO, 0);

// Clear status

// Чистим статус

//hc->opRegs->usbSts = ~0;

WOR(usbStsO, ~0);

// Enable controller

// Запускаем контроллер, 8 микро-фреймов, включаем

// последовательную и асинхронную очередь

//hc->opRegs->usbCmd = (8 << CMD\_ITC\_SHIFT) | CMD\_PSE | CMD\_ASE | CMD\_RS;

WOR(usbCmdO, (8 << CMD\_ITC\_SHIFT) | CMD\_PSE | CMD\_ASE | CMD\_RS);

while (ROR(usbStsO)&STS\_HCHALTED);

// Configure all devices to be managed by the EHCI

// Говорим, что завершили

//hc->opRegs->configFlag = 1;

WOR(configFlagO, 1);\

// Probe devices

// Пробуем порты

EhciProbe(hc);

Собственно, код для сброса порта в изначальное состояние:

volatile u32 \*reg = &hc->opRegs->ports[port];

// Включаем питание на порту, ждём 100мс

\*reg|=(1<<12)|(1<<20);

Wait(100);

// Сбрасываем порт, ждем 50 мс

EhciPortSet(reg, PORT\_RESET | (1<<12) | (1<<20) | (1<<6));

Wait(50);

EhciPortClr(reg, PORT\_RESET);

// Wait 100ms for port to enable (TODO - what is appropriate length of time?)

// Ждем 100 мс чтобы порт включился, в документации написано,

// что 100 мс должно хватить

uint status = 0;

for (uint i = 0; i < 10; ++i)

{

// Delay

Wait(10);

// Get current status

// Получаем текущий статус

status = \*reg;

// Check if device is attached to port

// Проверяем подключение устройства к контроллеру

if (~status & PORT\_CONNECTION)

break;

// Acknowledge change in status

// Если статус поменялся - чистим биты порта

if (status & (PORT\_ENABLE\_CHANGE | PORT\_CONNECTION\_CHANGE))

{

EhciPortClr(reg, PORT\_ENABLE\_CHANGE | PORT\_CONNECTION\_CHANGE);

continue;

}

// Check if device is enabled

// Проверяем устройство на то, что оно запустилось

if (status & PORT\_ENABLE)

break;

}

return status;

Control-запрос к устройству:

static void EhciDevControl(UsbDevice \*dev, UsbTransfer \*t)

{

EhciController \*hc = (EhciController \*)dev->hc;

UsbDevReq \*req = t->req;

// Determine transfer properties

// Обозначаем свойства транзакции

uint speed = dev->speed;

uint addr = dev->addr;

uint maxSize = dev->maxPacketSize;

uint type = req->type;

uint len = req->len;

// Create queue of transfer descriptors

// Создаём очередь TDs

EhciTD \*td = EhciAllocTD(hc);

if (!td)

return;

EhciTD \*head = td;

EhciTD \*prev = 0;

// Setup packet

// Инициализирующий пакет

uint toggle = 0;

uint packetType = USB\_PACKET\_SETUP;

uint packetSize = sizeof(UsbDevReq);

EhciInitTD(td, prev, toggle, packetType, packetSize, req);

prev = td;

// Data in/out packets

packetType = type & RT\_DEV\_TO\_HOST ? USB\_PACKET\_IN : USB\_PACKET\_OUT;

u8 \*it = (u8 \*)t->data;

u8 \*end = it + len;

//EhciPrintTD(td);

while (it < end)

{

td = EhciAllocTD(hc);

if (!td)

return;

toggle ^= 1;

packetSize = end - it;

if (packetSize > maxSize)

packetSize = maxSize;

EhciInitTD(td, prev, toggle, packetType, packetSize, it);

it += packetSize;

prev = td;

}

// Status packet

// Получаем статус

td = EhciAllocTD(hc);

if (!td)

return;

toggle = 1;

packetType = type & RT\_DEV\_TO\_HOST ? USB\_PACKET\_OUT : USB\_PACKET\_IN;

EhciInitTD(td, prev, toggle, packetType, 0, 0);

// Initialize queue head

// Инициализируем голову очереди:

EhciQH \*qh = EhciAllocQH(hc);

EhciInitQH(qh, t, head, dev->parent, false, speed, addr, 0, maxSize);

// Wait until queue has been processed

// Ждем пока очередь не будет обработана

EhciInsertAsyncQH(hc->asyncQH, qh);

EhciWaitForQH(hc, qh);

}

Код обработки очереди:

if (qh->token & TD\_TOK\_HALTED)

{

t->success = false;

t->complete = true;

}

else if (qh->nextLink & PTR\_TERMINATE)

if (~qh->token & TD\_TOK\_ACTIVE)

{

if (qh->token & TD\_TOK\_DATABUFFER)

kprintf(" Data Buffer Error\n");

if (qh->token & TD\_TOK\_BABBLE)

kprintf(" Babble Detected\n");

if (qh->token & TD\_TOK\_XACT)

kprintf(" Transaction Error\n");

if (qh->token & TD\_TOK\_MMF)

kprintf(" Missed Micro-Frame\n");

t->success = true;

t->complete = true;

}

if (t->complete)

....

И теперь запрос к конечной точке(Bulk-запрос)

static void EhciDevIntr(UsbDevice \*dev, UsbTransfer \*t)

{

EhciController \*hc = (EhciController \*)dev->hc;

// Determine transfer properties

// Обговариваем характеристики транзакции

uint speed = dev->speed;

uint addr = dev->addr;

uint maxSize = t->endp->desc->maxPacketSize;

uint endp = t->endp->desc->addr & 0xf;

EhciTD \*td = EhciAllocTD(hc);

if (!td)

{

t->success = false;

t->complete = true;

return;

}

EhciTD \*head = td;

EhciTD \*prev = 0;

// Data in/out packets

uint toggle = t->endp->toggle;

uint packetType = t->endp->desc->addr & 0x80 ? USB\_PACKET\_IN : USB\_PACKET\_OUT;

uint packetSize = t->len;

EhciInitTD(td, prev, toggle, packetType, packetSize, t->data);

// Initialize queue head

// Инициализируем голову очереди

EhciQH \*qh = EhciAllocQH(hc);

EhciInitQH(qh, t, head, dev->parent, true, speed, addr, endp, maxSize);

//printQh(qh);

// Schedule queue

// Добавляем в очередь

EhciInsertPeriodicQH(hc->periodicQH, qh);

}

Думаю, что тема достаточно интересная, в интернете на русском документаций, описаний и статей на эту тему почти нет, а если есть — очень размыто. Если интересна тема работы с железом и разработки ОС, то есть много чего рассказать.