

The World Leader in High Performance Signal Processing Solutions



BF52x USB应用程序开发介绍

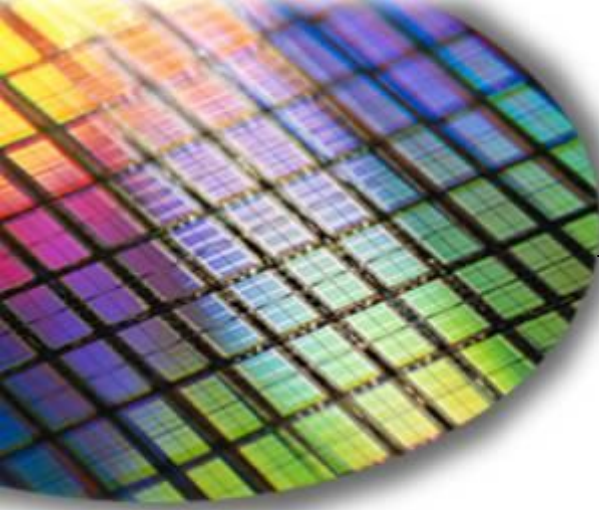


内容说明

本文档介绍**BF52x USB**应用程序开发过程，其中涉及到与程序开发相关的硬件平台和软件工具，文档将从如下几个方面进行描述：

- 开发环境及开发工具
- **Device Driver Model**
 - ◆ Device Manager& Logical Device
 - ◆ Physical Device Driver
- **USB** 硬件设备及驱动
- **ADI BULK**类设备实现

希望通过本文档能够对**BF52x USB**应用程序及相关驱动的开发有一个初步和快速的了解。



The World Leader in High Performance Signal Processing Solutions



开发环境及开发工具



开发环境及开发工具

一 开发工具下载

- ◆ 硬件开发平台: **ADSP-BF527 EZ-KIT Lite**
- ◆ 软件开发平台: **Visual DSP++ 5.0**
- ◆ **Update 版本: VisualDsp++ 5.0Update8.vdu** (目前最新 update10)

基础软件及更新版本下载地址:

http://www.analog.com/en/embedded-processing-dsp/blackfin/VDSP-BF-SH-TS/products/software-tools-upgrades/visualdsp_tools_upgrades/resources/fca.html

开发环境及开发工具

一 相关源代码文件

后边所有提到的
相关程序代码都
可以在以下目录
中找到

◆ 头文件

- **D:\Program Files\Analog Devices\VisualDSP 5.0\Blackfin\include**

◆ 源文件

- **D:\Program Files\Analog Devices\VisualDSP 5.0\Blackfin\lib\src\drivers**

◆ 库文件

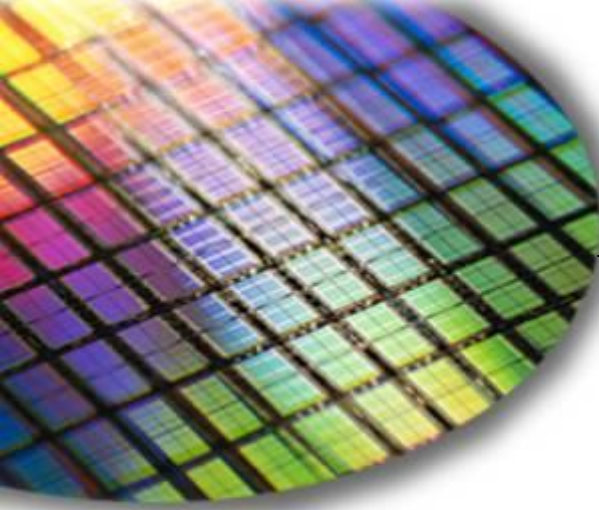
- **D:\Program Files\Analog Devices\VisualDSP 5.0\Blackfin\lib**

◆ USB例子源程序

- **D:\Program Files\Analog Devices\VisualDSP 5.0\Blackfin\Examples\ADSP-BF527 EZ-KIT Lite\Drivers\usb**

◆ 相关文档(参看帮助文档)

- **Device Drivers and System Services**
Manual for Blackfin® Processors



The World Leader in High Performance Signal Processing Solutions



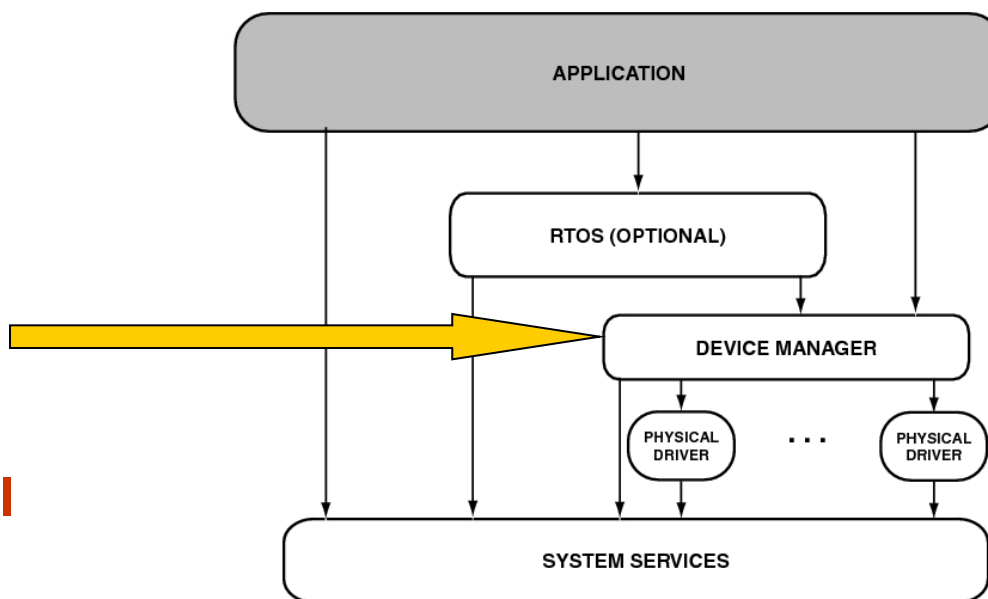
Device Driver Model

- ◆ **Device Manager & Logical Device**
- ◆ **Physical Device Driver**

Device Driver Model

— device manager & Logical Device

- `adi_dev_Init`
- `adi_dev_Open`
- `adi_dev_Close`
- `adi_dev_Read`
- `adi_dev_Write`
- `adi_dev_Control`



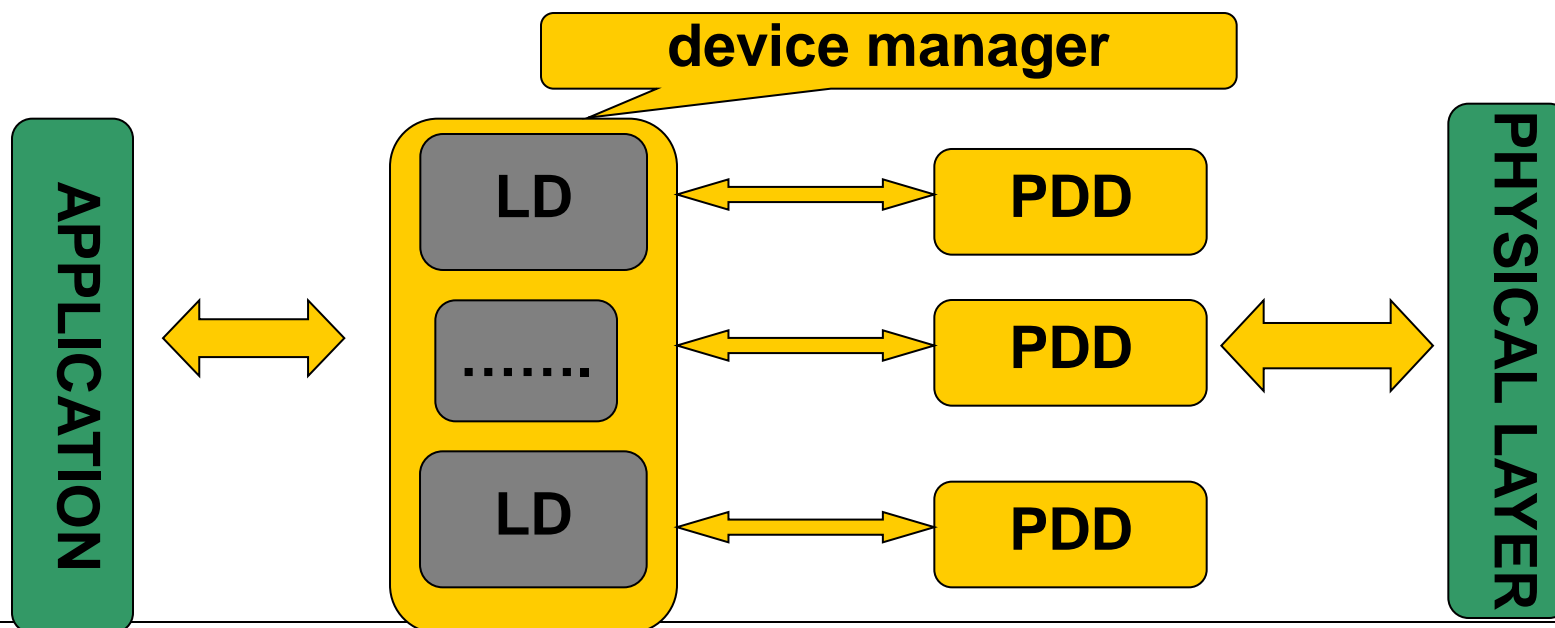
Device manager提供了一组对设备驱动进行读写及控制操作的统一**API**，它对硬件进行了抽象，加快了应用程序开发速度，便于进行调试并且能够使代码得到更好的重用，便于在不改变应用程序的情况下对硬件进行升级，利用此模型进行开发是比较好的选择。

Device Driver Model

— device manager和Device的关系

设备管理器所操作的设备是逻辑概念上的设备(LD),应用层对逻辑设备进行操作,这种操作再被映射到相应的物理设备驱动(PDD)上,从而实现完整的设备操作。

在ADI的设备模型中, **device manager**统一对每一个需要打开的设备进行创建,分配和管理。在整个程序中只存在一个设备管理器,但它对多个设备进行着管理。



Device Driver Model

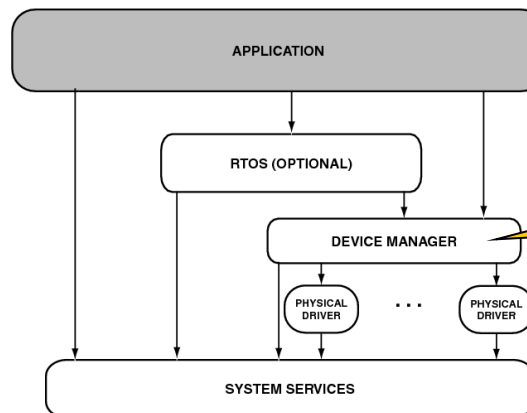
— device manager中两个重要数据结构

```
typedef struct ADI_DEV_DEVICE {  
    ADI_DEV_PDD_ENTRY_POINT *pEntryPoint;  
    struct ADI_DEV_MANAGER *pManager;  
    ADI_DEV_PDD_HANDLE PDDHandle;  
    ADI_DEV_MODE DataflowMethod;  
    ADI_DEV_DIRECTION Direction;  
    u8 InUseFlag;  
    u8 DataflowFlag;  
    u8 SynchronousFlag;  
    u8 StreamingFlag;  
    u32 PeripheralDMAFlag;  
    Config Config;  
    DMAHandle DMAHandle;  
    DCBHandle DCBHandle;  
    InboundDma InboundDma;  
    OutboundDma OutboundDma;  
    *pInboundDma;  
    *pOutboundDma;  
    *ClientHandle;  
    void ClientCallback;  
    ADI_DCB_CALLBACK_FN ADI_DCB_CALLBACK_FN;  
} ? end ADI_DEV_DEVICE ? ADI_DEV_DEVICE;
```

```
typedef struct ADI_DEV_MANAGER {  
    u32 DeviceCount;  
    void *pEnterCriticalSection;  
    ADI_DEV_DEVICE *Device;  
} ADI_DEV_MANAGER;
```

在设备模型中，所有的设备被抽象成左边这个结构体，它包含了对设备进行操作的所有信息

在设备模型中，设备管理器被抽象成这个结构体，通过这个结构可以对设备进行控制



这两个结构体就是组成这一层的实体

Device Driver Model

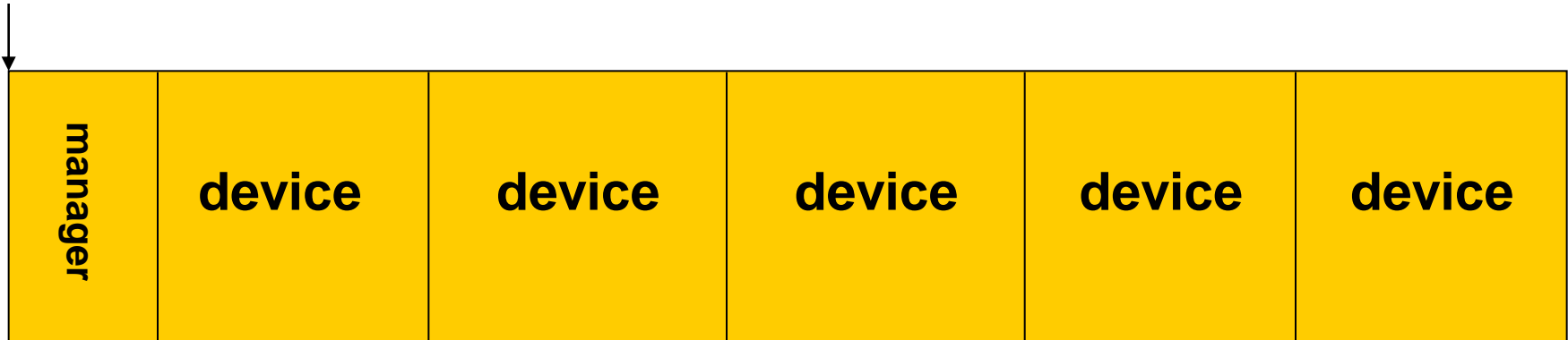
— device manager 的初始化

```
u32 adi_dev_Init(  
    void  
    size_t  
    u32  
    ADI_DEV_MANAGER_HANDLE  
    void  
);  
    *pMemory,  
    MemorySize,  
    *pMaxDevices,  
    *pManagerHandle,  
    *pEnterCriticalSectionParam
```

adi_dev_Init函数完成对设备管理器的初始化操作,主要是对由pMemory传入的缓冲区进行初始化, **此函数在程序中只需要调用一次**,通过pManagerHandle返回设备管理器句柄

此函数对pMemory所指向的内存进行如下的分配, 分成一个
ADI_DEV_MANAGER结构体和一个**ADI_DEV_DEVICE**结构体数组

pMemory



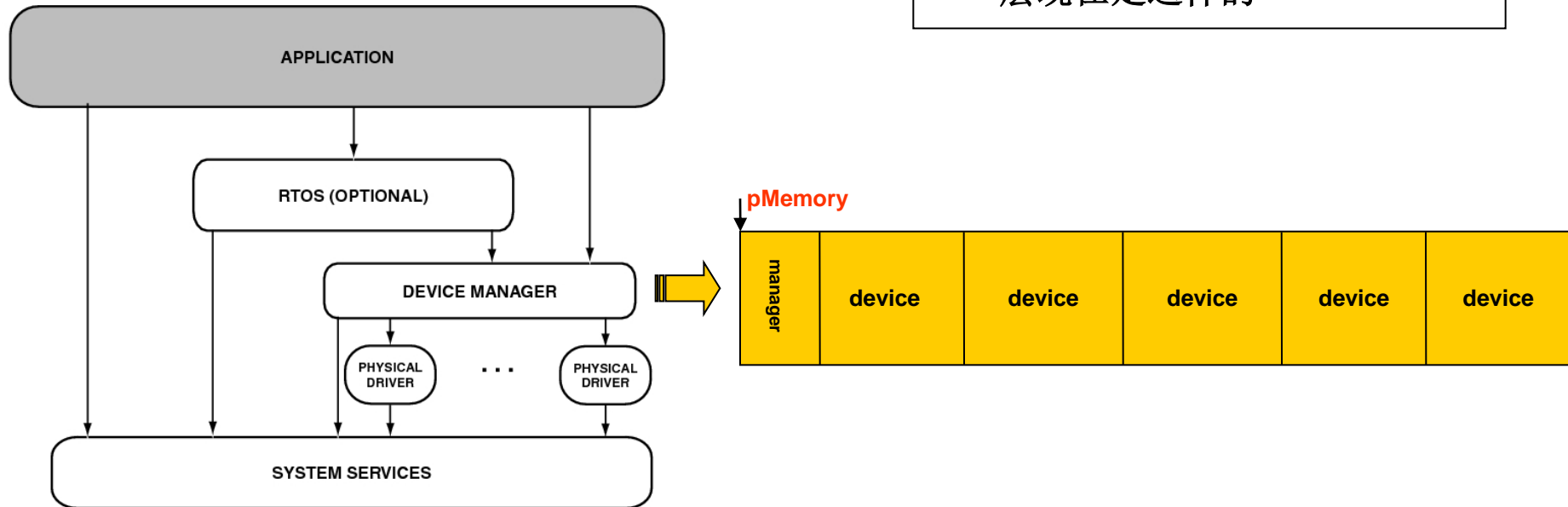
每一个**DEVICE**代表一个可以被管理的设备, **manager**对所有的设备进行管理

Device Driver Model

— device manager 的初始化

```
u32 adi_dev_Init(  
    void                *pMemory,  
    size_t              MemorySize,  
    u32                 *pMaxDevices,  
    ADI_DEV_MANAGER_HANDLE *pManagerHandle,  
    void                *pEnterCriticalParam  
);
```

执行完adi_dev_Init这个函数我们可以认为**DEVICE MANAGER** 这一层现在是这样的



Device Driver Model

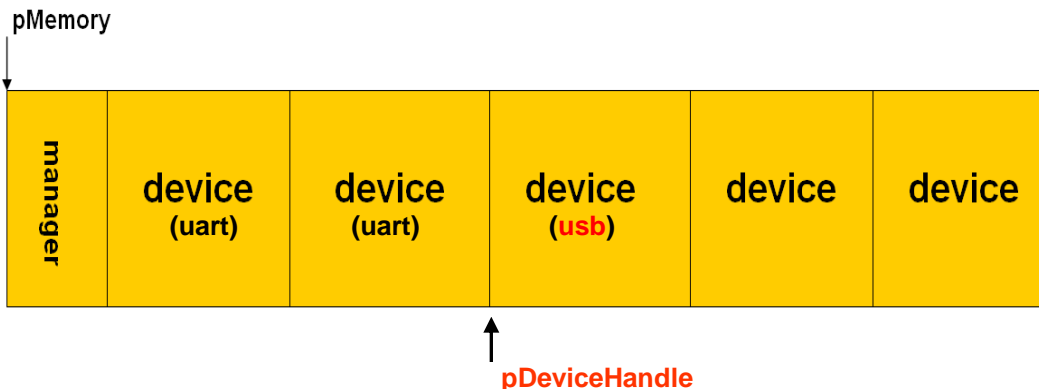
一 设备的打开

```
u32 adi_dev_Open(  
    ADI_DEV_MANAGER_HANDLE  
    ADI_DEV_PDD_ENTRY_POINT  
    u32  
    void  
    ADI_DEV_DEVICE_HANDLE  
    ADI_DEV_DIRECTION  
    ADI_DMA_MANAGER_HANDLE  
    ADI_DCB_HANDLE  
    ADI_DCB_CALLBACK_FN  
);  
  
ManagerHandle,  
*pEntryPoint,  
DeviceNumber,  
*ClientHandle,  
*pDeviceHandle,  
Direction,  
DMAHandle,  
DCBHandle,  
ClientCallback
```

通过上一个函数我们创建了设备管理器，在这里我们通过这个句柄作为参数打开一个设备

Physical device drivers (PDD) 提供了对物理设备进行实际寄存器读写的功能函数，我们此时打开的设备是逻辑设备，通过这个参数我们将逻辑设备和物理设备关联在一起，两者共同完成对设备的操作

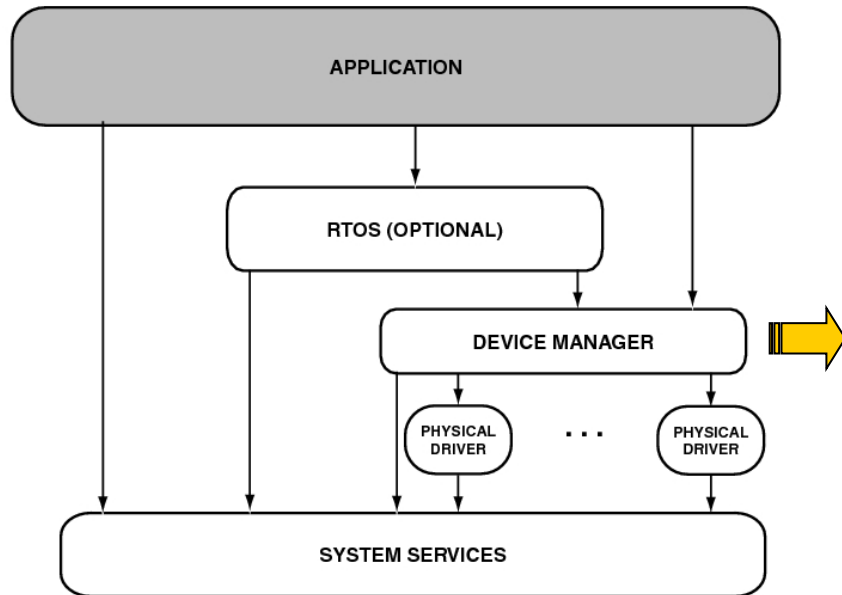
函数执行时从device结构体数组中找一个没有被占用的结构体，按照传入的参数对其进行初始化，并将其地址赋给pDeviceHandle返回,这样我们在内存中就存在一块区域对应我们的设备，例如如果pEntryPoint代表的是USB设备驱动，那么此区域就代表打开了一个USB设备，**pEntryPoint决定了打开的是什么设备，它代表了具体的物理设备。**



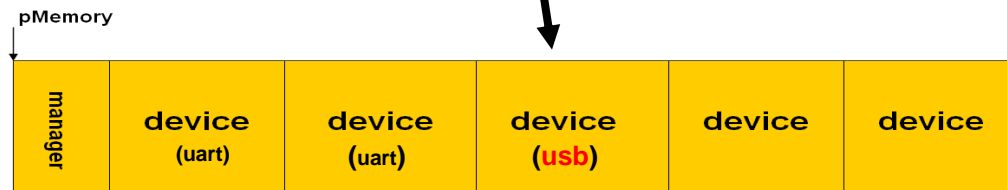
Device Driver Model

一 设备的打开

```
u32 adi_dev_Open(  
    ADI_DEV_MANAGER_HANDLE      ManagerHandle,  
    ADI_DEV_PDD_ENTRY_POINT     *pEntryPoint,  
    u32                          DeviceNumber,  
    void                         *ClientHandle,  
    ADI_DEV_DEVICE_HANDLE       *pDeviceHandle,  
    ADI_DEV_DIRECTION           Direction,  
    ADI_DMA_MANAGER_HANDLE       DMAHandle,  
    ADI_DCB_HANDLE              DCBHandle,  
    ADI_DCB_CALLBACK_FN         ClientCallback  
);
```



执行完这个函数我们可以认为**DEVICE MANAGER**这一层现在是这样的



Device Driver Model

— 设备的操作

```
u32 adi_dev_Control(  
    ADI_DEV_DEVICE_HANDLE  
    u32  
    void  
);
```

DeviceHandle,
Command,
*pArg

```
u32 adi_dev_Read(  
    ADI_DEV_DEVICE_HANDLE  
    ADI_DEV_BUFFER_TYPE  
    ADI_DEV_BUFFER  
);
```

DeviceHandle,
BufferType,
*pBuffer

```
u32 adi_dev_Write(  
    ADI_DEV_DEVICE_HANDLE  
    ADI_DEV_BUFFER_TYPE  
    ADI_DEV_BUFFER  
);
```

DeviceHandle,
BufferType,
*pBuffer

使用打开设备
(**adi_dev_open**函数)时,
返回得到的设备HANDLE

配置设备到工作状态,
利用读写函数进行读写
操作, 实现设备功能




pDeviceHandle

所有对设备的操作都会和这个结构发生关系, 各函数
并将对设备的操作映射到对PDD的操作

Device Driver Model

— 回调函数

```
u32 adi_dev_Open(  
    ADI_DEV_MANAGER_HANDLE  
    ADI_DEV_PDD_ENTRY_POINT  
    u32  
    void  
    ADI_DEV_DEVICE_HANDLE  
    ADI_DEV_DIRECTION  
    ADI_DMA_MANAGER_HANDLE  
    ADI_DCB_HANDLE  
    ADI_DCB_CALLBACK_FN  
);  
  
typedef void (*ADI_DCB_CALLBACK_FN) (void*, u32, void*);
```



```
ManagerHandle,  
*pEntryPoint,  
DeviceNumber,  
*ClientHandle,  
*pDeviceHandle,  
Direction,  
DMAHandle,  
DCBHandle,  
ClientCallback
```

当驱动中一个事件发生时，例如：缓冲区已经处理完成，数据已经发送完成，缓冲区已经被数据填满等，那么应用程序提供的回调函数会被调用，向应用程序传递发生的事件。

When an event occurs, the client's callback function is invoked and passed the enumeration of the event that occurred .

回调函数是驱动通知应用层的一种方式。通过回调函数，应用层可以处理比较感兴趣的事件，例如：对缓冲区的读或写操作完成，从而做出相应的处理，函数原型一般如下：

```
void ClientCallback( void *AppHandle, u32 Event, void *pArg)
```


Device Driver Model

— 回调函数

```
/* Callback Events from the peripheral driver */
enum USB_EPZERO_CALLBACK_EVENTS
{
    /* USB device events, raised by the peripheral driver */
    ADI_USB_EVENT_NONE=ADI_USB_ENUMERATION_START,
    ADI_USB_EVENT_SETUP_PKT, /* EPO D
    ADI_USB_EVENT_START_OF_FRAME, /* Start
    ADI_USB_EVENT_RESUME, /* Resum
    ADI_USB_EVENT_SUSPEND, /* Suspe
    ADI_USB_EVENT_ROOT_PORT_RESET, /* Rese
    ADI_USB_EVENT_VBUS_TRUE, /* cable
    ADI_USB_EVENT_VBUS_FALSE, /* cable
    ADI_USB_EVENT_SET_CONFIG, /* TODO
    ADI_USB_EVENT_START_DEV_ENUM, /*

    /* USB class driver or application events */
    ADI_USB_EVENT_DATA_RX, /* TODO
    ADI_USB_EVENT_DATA_TX, /* TODO
    ADI_USB_EVENT_RX_COMPLETE, /* Data
    ADI_USB_EVENT_TX_COMPLETE, /* Data
    ADI_USB_EVENT_PKT_RCVD_NO_BUFFER,
    ADI_USB_OTG_EVENT_ENUMERATION_COMPLETE,
    ADI_USB_OTG_EVENT_SET_CONFIG_COMPLETE,
    ADI_USB_OTG_EVENT_SET_INTERFACE_COMPLETE,
    ADI_USB_EVENT_DISCONNECT,
    ADI_USB_EVENT_RX_STALL,
    ADI_USB_EVENT_TX_STALL,
    ADI_USB_EVENT_CONNECT,
    ADI_USB_EVENT_RX_NAK_TIMEOUT,
    ADI_USB_EVENT_TX_NAK_TIMEOUT,
    ADI_USB_EVENT_RX_ERROR,
    ADI_USB_EVENT_TX_ERROR,
    ADI_USB_EVENT_SET_INTERFACE
};
```

我们可以给我们自己的设备定义应用需要响应的事件，当其发生时对它进行相应处理，**USB**设备定义的相关事件。

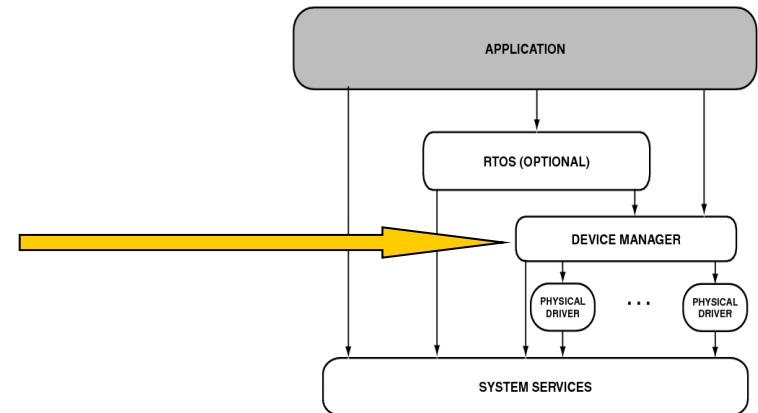
通过回调函数定义相关的事件通知,可以使得应用层和驱动的交互更加方便,例如: 应用层主动发起某一操作, 等待系统回调对此操作进行响应, 可根据实际的开发需求定义自己的事件。

Device Driver Model

一 程序典型开发流程

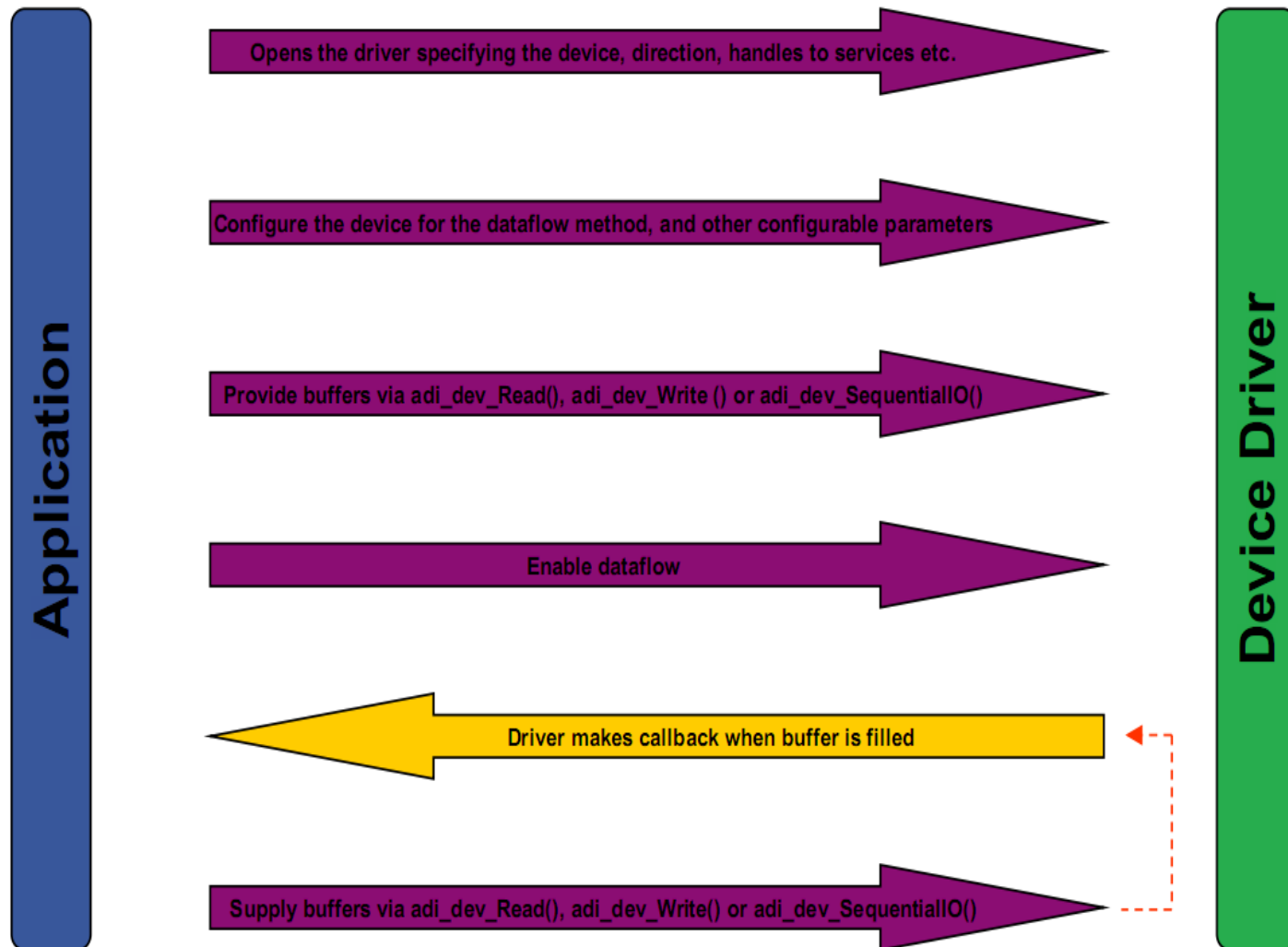
通过前边的分析，我们可以知道对**device manager**所提供的接口进行调用顺序是有规律的，从而我们可以得到一个基于**ADI**驱动模型进行应用开发的基本流程：

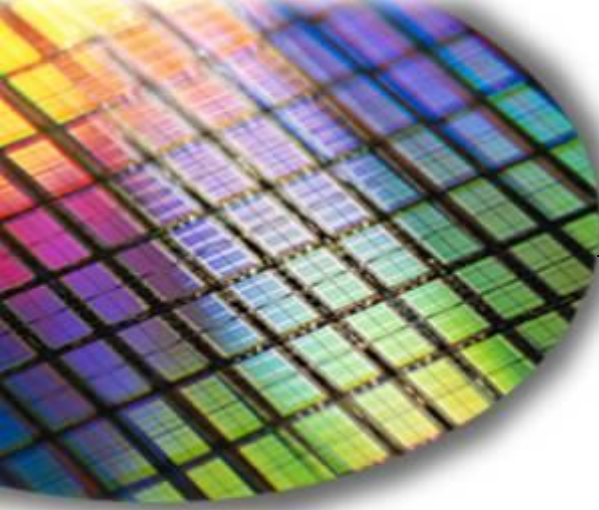
- I. 初始化**DEVICE MANAGER**,得到其句柄
- II. 利用得到的设备管理器句柄打开设备,提供相应的**PDD**及回调函数和其它相关的参数如**DMA MANAGER**,**DATA DERICTION**
- III. 对驱动进行相关的配置,包括**DATA FLOW METHOD**,**ENABLE DATAFLOW**及其它的配置
- IV. 通过读写函数给驱动提供缓存区和缓存区类型,进行读写操作
- V. 回调函数等待处理



Device Driver Model

一 程序典型开发流程





The World Leader in High Performance Signal Processing Solutions



Device Driver Model

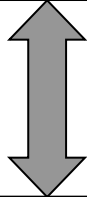
- ◆ Device Manager & Logical Device
- ◆ **Physical Device Driver**

Device Driver Model

— Physical Device Driver



Device manager 提供的API



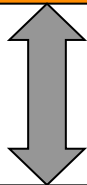
```
u32 adi_dev_Open(  
    ADI_DEV_MANAGER_HANDLE  
    ADI_DEV_PDD_ENTRY_POINT  
    u32  
    void  
    ADI_DEV_DEVICE_HANDLE  
    ADI_DEV_DIRECTION  
    ADI_DMA_MANAGER_HANDLE  
    ADI_DCB_HANDLE  
    ADI_DCB_CALLBACK_FN  
);
```

```
ManagerHandle,  
*pEntryPoint,  
DeviceNumber,  
*ClientHandle,  
*pDeviceHandle,  
Direction,  
DMAHandle,  
DCBHandle,  
ClientCallback
```

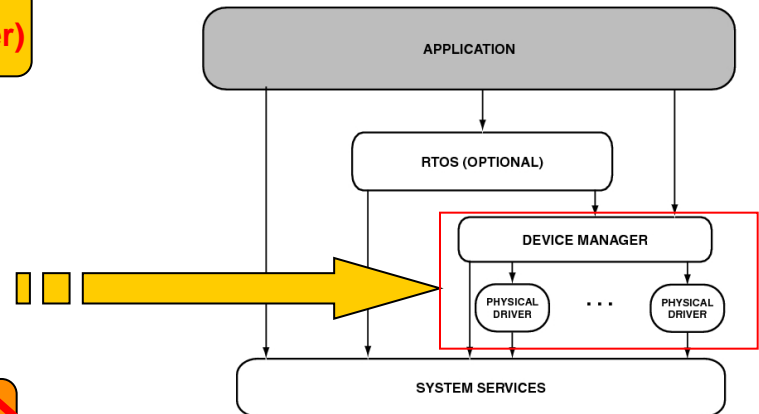
此函数将
逻辑设备
和物理设备
驱动关联在一块



寄存器或DMA操作



在Application只能看到逻辑设备其接口是统一的，不论下层的Physical Device是否发生变化，使得程序的移植更加容易，实现了应用逻辑的平台无关



把所有的硬件细节封装在Physical Device driver 这一层只通过pEntryPoint向上层提供服务

Device Driver Model

— Physical Device Driver

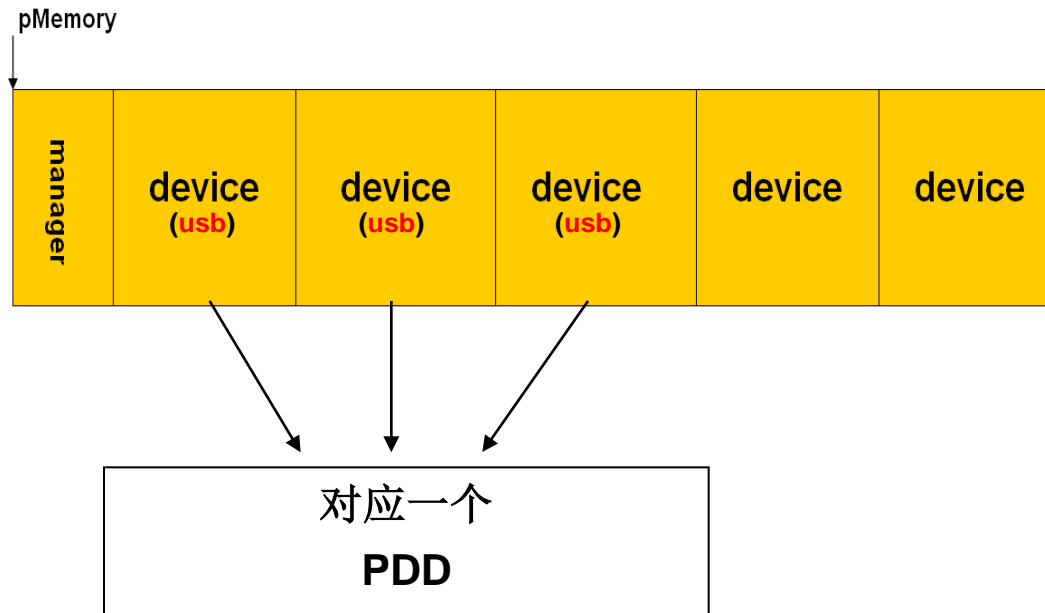
```
u32 adi_dev_Open(  
    ADI_DEV_MANAGER_HANDLE    ManagerHandle,  
    ADI_DEV_PDD_ENTRY_POINT    *pEntryPoint,  
    u32                        DeviceNumber,  
    void                        *ClientHandle,  
    ADI_DEV_DEVICE_HANDLE      *pDeviceHandle,  
    ADI_DEV_DIRECTION          Direction,  
    ADI_DMA_MANAGER_HANDLE     DMAHandle,  
    ADI_DCB_HANDLE             DCBHandle,  
    ADI_DCB_CALLBACK_FN        ClientCallback  
);  
  
typedef struct {  
    u32 (*adi_pdd_Open)(  
        ADI_DEV_MANAGER_HANDLE    ManagerHandle,  
        u32                        DeviceNumber,  
        ADI_DEV_DEVICE_HANDLE      DeviceHandle,  
        ADI_DEV_PDD_HANDLE          *pPDDHandle,  
        ADI_DEV_DIRECTION          Direction,  
        void                        *pCriticalRegionArg,  
        ADI_DMA_MANAGER_HANDLE     DMAHandle,  
        ADI_DCB_HANDLE             DCBHandle,  
        ADI_DCB_CALLBACK_FN        DMCallback  
    );  
    u32 (*adi_pdd_Close)(  
        ADI_DEV_PDD_HANDLE          PDDHandle  
    );  
    u32 (*adi_pdd_Read)(  
        ADI_DEV_PDD_HANDLE          PDDHandle,  
        ADI_DEV_BUFFER_TYPE         BufferType,  
        ADI_DEV_BUFFER              *pBuffer  
    );  
    u32 (*adi_pdd_Write)(  
        ADI_DEV_PDD_HANDLE          PDDHandle,  
        ADI_DEV_BUFFER_TYPE         BufferType,  
        ADI_DEV_BUFFER              *pBuffer  
    );  
    u32 (*adi_pdd_Control)(  
        ADI_DEV_PDD_HANDLE          PDDHandle,  
        u32                          CommandID,  
        void                          *Value  
    );  
    u32 (*adi_pdd_SequentialIO)(  
        ADI_DEV_PDD_HANDLE          PDDHandle,  
        ADI_DEV_BUFFER_TYPE         BufferType,  
        ADI_DEV_BUFFER              *pBuffer  
    );  
} ? end {anonADI_DEV_PDD_ENTRY_POINT} ? ADI_DEV_PDD_ENTRY_POINT;
```

在驱动模型中每个物理设备对应一个物理驱动结构体，该结构实现了对硬件的直接操作，如寄存器读取及配置，从编程的角度讲它就是一组函数指针，我们在`adi_dev_Open`函数中完成逻辑设备和硬件驱动的关联。

在USB开发中我们要实现针对USB控制器的PDD

Device Driver Model

— Physical Device Driver



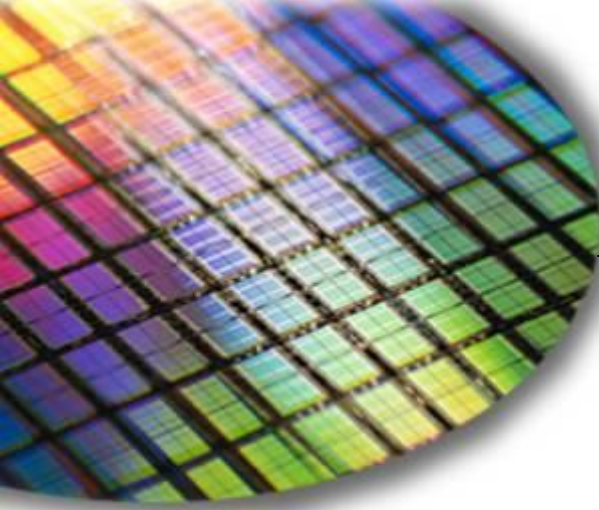
在实际的应用中我们可能打开多个usb逻辑设备作为不同的应用，但实际上系统中只存在一个物理设备，这也得益于这种设备驱动的分层设计



Device Driver Model

— Physical Device Driver

- ◆ **PDD**真正实现了物理设备的操作，但我们对这一层并不需要掌握太多
- ◆ **ADI** 已经实现了这一层的大多设备的驱动
 - **USB /PPI /EPPI /SPORT /SPI**
 - **LCD TOUCHSCREEN**
 - **ADC DAC CODEC DECODEC**
- ◆ 对这一层的了解有助于我们更好的理解**ADI**的设备驱动模型，我们只需关注应用层接口，通过使用已经完成的驱动来实现我们自己的应用，即使用**USB**只需找到**USB**驱动，使用**PPI**只要加载**PPI**的驱动



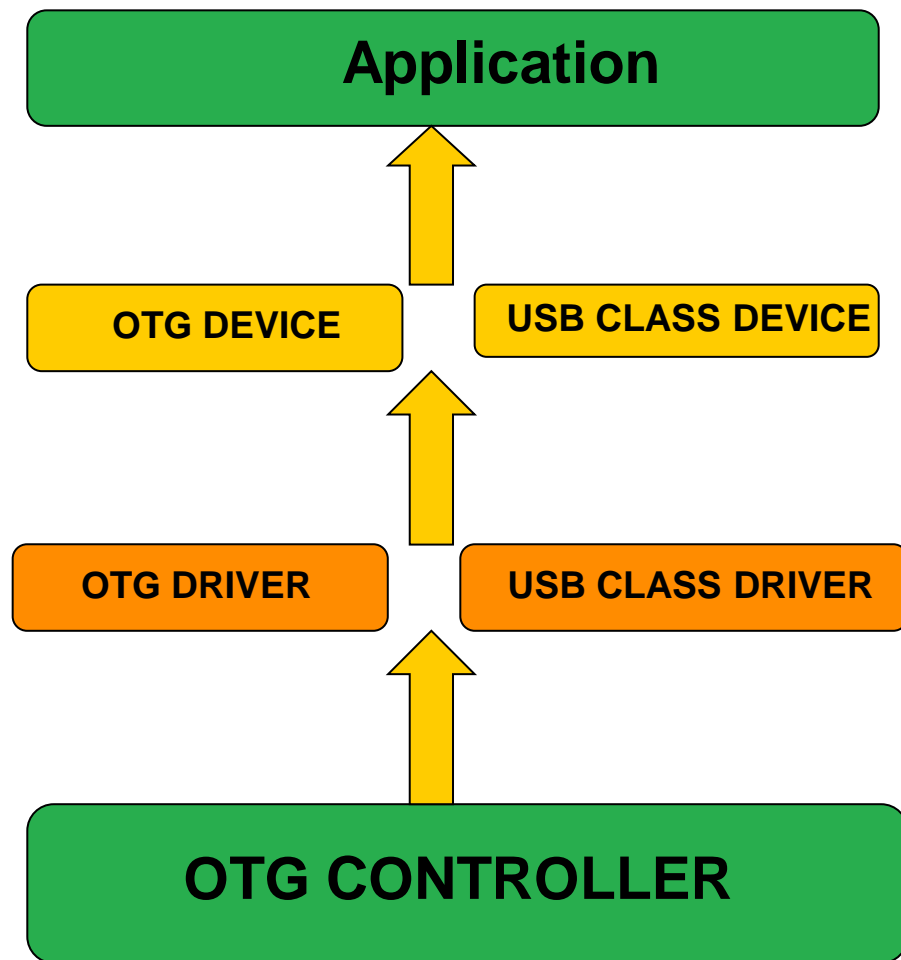
The World Leader in High Performance Signal Processing Solutions



USB 硬件设备及驱动

USB硬件设备及驱动

—BF527 USB开发



从下向上逐步介绍527
上USB设备的实现过程

- ◆ **PHYSICAL DEVICE ---- OTG**
- ◆ **PDD**
 - OTG DRIVER
 - USB CLASS DRIVER
- ◆ **LOGICAL DEVICE**
 - OTG DEVICE
 - USB CLASS DEVICE
- ◆ **APPLICATION**



USB硬件设备及驱动

—BF527 USB硬件资源介绍

BF527提供了一个标准的**USB2.0 OTG**控制器,该控制器支持**1.5Mbps,12Mbps**和**480Mbps**三种工作速率,它既可作为传统的**USB**外部设备,又可以作为**USB OTG**主机,其主要特性如下:

- low speed, full speed, high speed rates supported
- one bidirectional control endpoint
- seven transmit and seven receive unidirectional endpoints
- 7.232K Bytes of FIFOs for packet buffering
- eight DMA master channels
- three top-level maskable general purpose interrupts
- one asynchronous wakeup interrupt
- VBUS control interrupts for external analog VBUS control
- software-controlled clock control on each endpoint for power reduction
- session request protocol (SRP) and host negotiation protocol (HNP) capability
- host transaction scheduling in hardware
- soft connect/disconnect feature
- full- and high-speed physical layer UTMI+ level 2 interface for on-chip PHY
- backwards compatible with existing USB 1.1 hosts

USB硬件设备及驱动

—BF527 USB硬件资源介绍

BF527的USB2.0 OTG控制器，提供了**7.232KBytes FIFO**，其中**#5—#7 EP**具有**1024Bytes FIFO**，能够使**BULK** 传输在高速状态下以**双缓冲模式**来工作，极大的提高了系统的吞吐率。

Bidirectional Endpoint (RX and TX)	FIFO Size (each direction)	USB Transfer Types
0	64 bytes	Size fixed for Control transfers.
1-4	128 bytes	Bulk, Interrupt, Isochronous
5-7	1024 bytes	Bulk, Interrupt, Isochronous



USB硬件设备及驱动

—BF527 USB硬件资源介绍

BF527的USB2.0 OTG控制器，提供了**8个DMA通道**来实现处理器和**USB控制器**间大量数据的高效传输。

每个**DMA**控制器能够工作在两种模式下：**0** 和 **1**

工作在**模式1**下的**DMA**控制器，能够完成一次完整的**BULK**传输，而只在所有的包都传输完成之后才对**Processor**进行一次中断，即多次传输一次中断。**模式1**特别适合工作在**BULK**模式下的**EP**进行大数据量的传输。

NOTE

the last packet in the series may be less than the maximum packet size and the receiver may use this “short” packet to signal the end of the transfer. If the total size of the transfer is an exact multiple of the maximum packet size, the transmitting software should send a null packet for the receiver to detect.

USB硬件设备及驱动

— BF527 USB OTG DRIVER

我们提供的USB PDD将OTG CONTROLLER抽象成下面的结构体：它就代表OTG控制器

```
typedef struct UsbOTGDeviceData
{
    ADI_DEV_DEVICE_HANDLE      DeviceHandle; /* Device handle */
    ADI_DMA_MANAGER_HANDLE     DMAHandle; /* DMA handle */
    ADI_DCB_HANDLE             DCBHandle; /* DCB handle */
    ADI_DCB_CALLBACK_FN        DMCallback; /* Callback routine */
    ADI_DEV_DIRECTION          Direction; /* Device direction */
    void                        *pCriticalRegionArg; /* Storage to save interrupt mask */
    PHYSICAL_ENDPOINT_OBJECT   PhysicalEndpointObjects[ NUM_PHYSICAL_ENDPOINTS ];
    s32_t                       NumPhysicalEndpoints; /* No# of physical endpoints */
    bool                        Started; /* Device Started */
    DEVICE_STATE                State; /* Device State */
    ADI_INT_PERIPHERAL_ID       PeripheralID; /* Peripheral ID */
    interrupt_kind              IvgUSBINT0; /* USB INT0 ivg */
    interrupt_kind              IvgUSBINT1; /* USB INT1 ivg RX */
    interrupt_kind              IvgUSBINT2; /* USB INT2 ivg TX */
    interrupt_kind              IvgUSBDMAMINT; /* USB DMAMINT ivg */
    ADI_FLAG_ID                 VbusDriveFlag; /* flag used to drive VBUS */
    s32_t                       DeviceID; /* Device ID */
    s32_t                       DeviceAddress; /* Device address- USB */
    DEVICE_OBJECT               *pDeviceObj; /* Logical Device OBJ */
    bool                        Cache; /* Cache on or off? */
    s32_t                       BufferPrefix; /* Buffer prefix */
    DEV_MODE                    Mode; /* Device MODE */
    ADI_USB_DEVICE_SPEED        Speed; /* Device SPEED */
    INTERRUPT_HANDLER_FUN       RxInterruptHandler; /* Function pointer to RX interrupt handler */
    INTERRUPT_HANDLER_FUN       TxInterruptHandler; /* Function pointer to TX interrupt handler */
    DEV_ENUM_STATE              EnumerationState; /* Device Enumeration state */
    s32_t                       RxDmaMode; /* RX DMA mode */
    s32_t                       TxDmaMode; /* TX DMA mode */
#ifdef ADI_HDRC_STATISTICS
    ADI_HDRC_STATS              Stats; /* Statistics */
#endif
} ADI_USBOTG_DEVICE;
```


USB硬件设备及驱动

— BF527 USB OTG DRIVER

在驱动程序中我们定义了一个变量**UsbOtgDevice**作为**OTG CONTROLLER** 的实例

```
static ADI_USBOTG_DEVICE UsbOtgDevice = {0};
```

所有的操作都是对这个变量进行，它保存了控制器的状态等信息

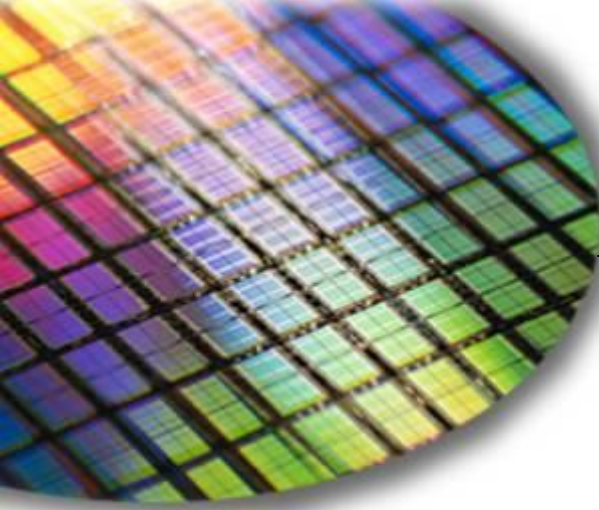
```
__ADI_USB_HDRC_SECTION_DATA
ADI_DEV_PDD_ENTRY_POINT ADI_USBDRC_Entrypoint =
{
    adi_pdd_Open,
    adi_pdd_Close,
    adi_pdd_Read,
    adi_pdd_Write,
    adi_pdd_Control
};
```

这是我们的**PDD endpoint**，它向上提供了对控制器操作的所有接口，在我们打开设备时将其作为参数传入打开函数

```
u32 adi_dev_Open(
    ADI_DEV_MANAGER_HANDLE    ManagerHandle,
    ADI_DEV_PDD_ENTRY_POINT    *pEntryPoint,
    u32                        DeviceNumber,
    void                       *ClientHandle,
    ADI_DEV_DEVICE_HANDLE     *pDeviceHandle,
    ADI_DEV_DIRECTION         Direction,
    ADI_DMA_MANAGER_HANDLE    DMAHandle,
    ADI_DCB_HANDLE            DCBHandle,
    ADI_DCB_CALLBACK_FN       ClientCallback
);
```

至此我们完成了**PDD**，我们已经可以在程序中打开一个**OTG**设备了，但并没实现完整的**usb**功能

驱动的详细实现可参照源码目录下的**adi_usb_hdrc.c**文件



The World Leader in High Performance Signal Processing Solutions



USB ADI BULK类设备实现

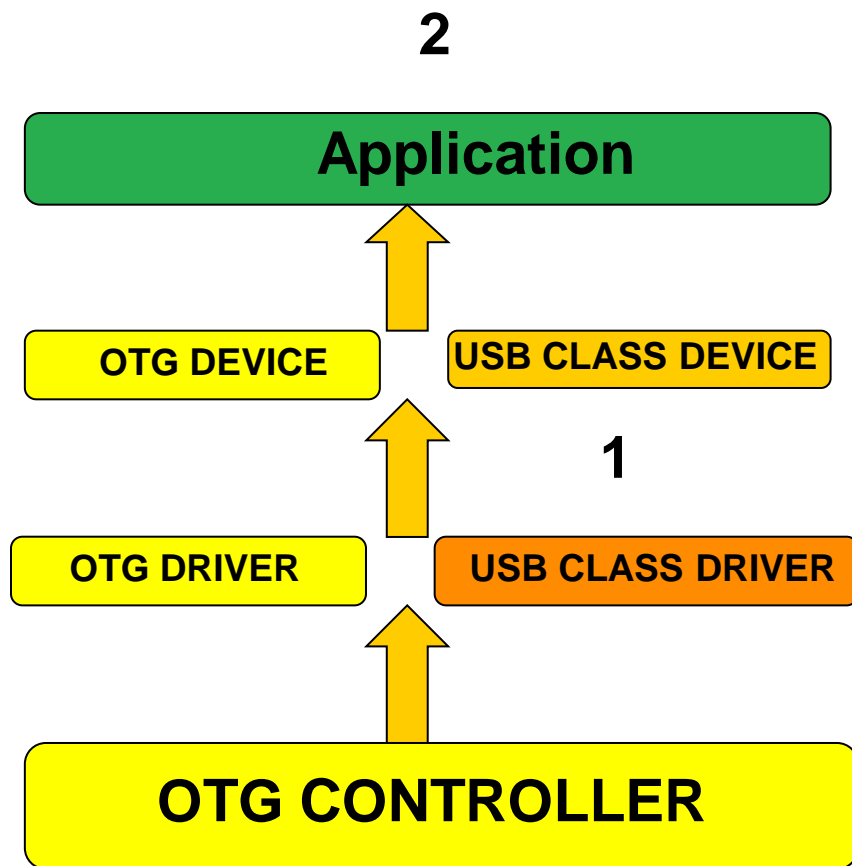


USB ADI BULK类设备实现

如何最快的实现我的**USB**应用

USB ADI BULK类设备实现

— 如何最快的实现USB应用



我们在了解了硬件实现了**OTG DRIVER**之后，还须实现一个**usb class driver**，它实现了最终的设备功能，应用程序直接对它进行编程，它可能是**Mass Storage**，**HID**，**Audio**，**MP3**.....

我们需要做的是：

- 1 --- 定义自己的类设备
- 2 --- 在其上实现应用开发



USB ADI BULK类设备实现

— 如何最快的实现**USB**应用

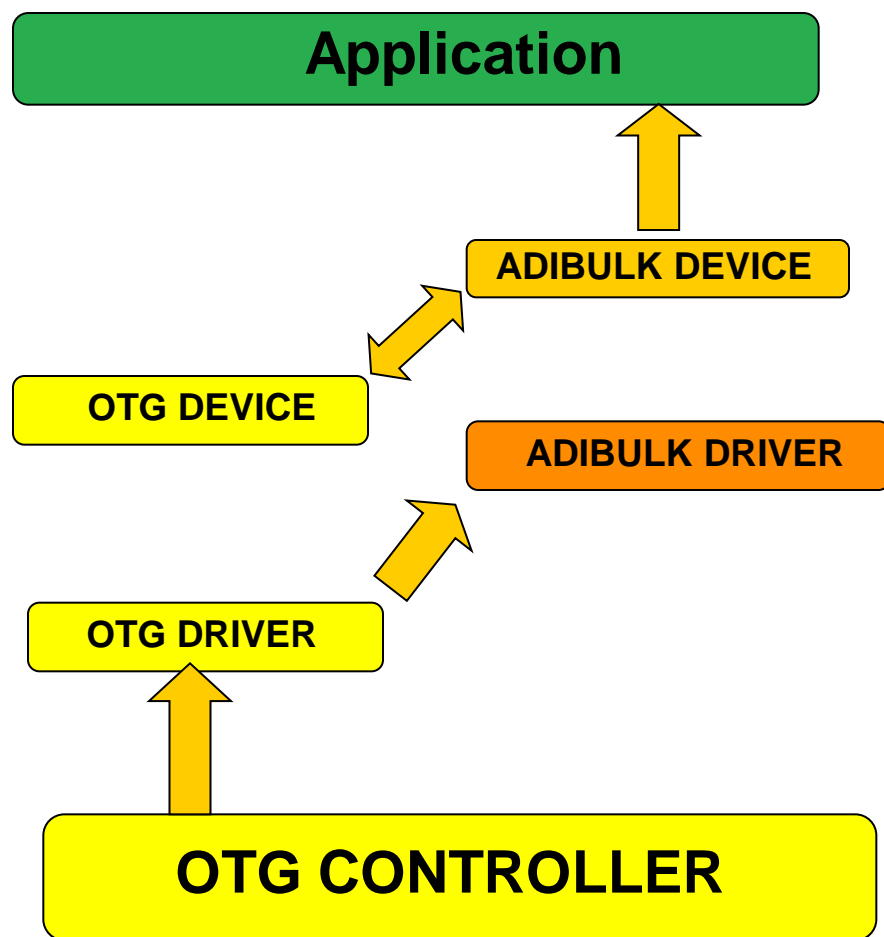
- ◆ **ADI** 已经提供了完整的**USB**核和硬件控制器(**OTG Controller**)驱动
- ◆ **ADI** 已经提供了进行**DMA**控制和操作的统一接口
- ◆ **ADI** 已经提供了进行**USB**应用开发的实例程序

用户需要做的工作:

- ◆ 分析**ADI BULK**类设备驱动的实现
- ◆ 定义满足自己需求的类设备
- ◆ 编写应用程序
- ◆ 调试程序功能

USB ADI BULK类设备实现

— ADI BULK类设备



USB类设备的实现:

ADIBulk Driver完成了USB协议中**DEVICE**、**CONFIG**、**INTERFACE**、**ENDPOINT**各描述符的定义，它实现各个**EndPoint**的功能，实现上层的应用逻辑，它是一个功能设备，它需要借助于**OTG Device**来实现**USB**硬件操作，作为一个功能设备我们同样要实现一个**EntryPoint**来完成驱动功能。

USB ADI BULK类设备实现

一 编写USB类设备

1 创建DEVICE、CONFIG、INTERFACE、ENDPOINT描述符

按照USB的协议，设备以DEVICE到ENDPOINT的树状结构来组织，按照应用需求来创建自己的配置及相关描述符。

```
/* *****  
 *  
 * Function:      VSBulkConfigure  
 *  
 * Description:   Performs class configuration  
 *  
 ***** */  
__ADI_USB_BULK_SECTION_CODE  
static s32_t VSBulkConfigure(void)
```

ADIBUIK类设备驱动在这个函数中实现了对EP的配置和功能分配。具体实现请参照 [adi_usb_bulkadi.c](#) 文件。

USB ADI BULK类设备实现

一 编写USB类设备

2. 创建entry point 结构体

根据设备驱动模型，每一个设备都有一个进入点结构来实现所有的物理操作，ADI BULK设备也应该实现这一结构体以完成对物理设备的操作。

```
__ADI_USB_BULK_SECTION_DATA
ADI_DEV_PDD_ENTRY_POINT ADI_USB_VSBulk_Entrypoint = {
    adi_pdd_Open,
    adi_pdd_Close,
    adi_pdd_Read,
    adi_pdd_Write,
    adi_pdd_Control
};
```

这个结构实现了BULK应用的进入点（Entry Point）结构体。

USB ADI BULK类设备实现

一 编写USB类设备

3. 实现entry point 结构体中各个函数

定义了entry point 结构我们就要实现其中的每个函数。

```
__ADI_USB_BULK_SECTION_CODE
static u32 adi_pdd_Open(
    ADI_DEV_MANAGER_HANDLE   ManagerHandle, /* Open a device */
    u32                       DeviceNumber,  /* device manager */
    ADI_DEV_DEVICE_HANDLE    DeviceHandle,  /* device number */
    ADI_DEV_PDD_HANDLE       *pPDDHandle,   /* device handle */
    ADI_DEV_DIRECTION        Direction,     /* pointer to PDD han */
    void                      *pCriticalRegionArg, /* data direction */
    ADI_DMA_MANAGER_HANDLE    DMAHandle,     /* critical region */
    ADI_DCB_HANDLE           DCBHandle,      /* handle to the DM */
    ADI_DCB_CALLBACK_FN      DMCallback,    /* callback handle */
    /* device manager */
)
{
    VENDORSPECIFIC_DEV_DATA *pVSDev = & VSDevData ;
    u32 Result = ADI_DEV_RESULT_SUCCESS;

    /* Check if class driver has been already already opened */
    if(!pVSDev->Open)
    {
        pVSDev->Open = true;
        pVSDev->DeviceHandle = DeviceHandle;
        pVSDev->DCBHandle    = DCBHandle;
        pVSDev->DMCallback    = DMCallback;
        pVSDev->CriticalData = pCriticalRegionArg;
        pVSDev->Direction    = Direction;
        pVSDev->IsClassConfigured = false;
    }
    else
        Result = ADI_DEV_RESULT_DEVICE_IN_USE;

    return(Result);
} ? end adi_pdd_Open ?
```



USB ADI BULK类设备实现

一 编写USB类设备

3. 实现entry point 结构体中各个函数

定义了entry point 结构我们就要实现其中的每个函数。

```
__ADI_USB_BULK_SECTION_CODE
static u32 adi_pdd_Read(          /* Reads data or queues ;
    ADI_DEV_PDD_HANDLE PDDHandle, /* PDD handle */
    ADI_DEV_BUFFER_TYPE BufferType, /* buffer type */
    ADI_DEV_BUFFER *pBuffer      /* pointer to buffer */
)
{
    unsigned int Result;
    unsigned int epNum;
    VENDORSPECIFIC_DEV_DATA *pVSDev = & VSDevData ;

    Result = adi_dev_Read(    pVSDev->PeripheralDevHandle,
                             BufferType,
                             pBuffer);

    return(Result);
}
```



USB ADI BULK类设备实现

一 编写USB类设备

3. 实现entry point 结构体中各个函数

定义了entry point 结构我们就要实现其中的每个函数。

```
__ADI_USB_BULK_SECTION_CODE
static u32 adi_pdd_Write(          /* Writes data or queue */
    ADI_DEV_PDD_HANDLE PDDHandle, /* PDD handle */
    ADI_DEV_BUFFER_TYPE BufferType, /* buffer type */
    ADI_DEV_BUFFER *pBuffer        /* pointer to buffer */
)
{
    unsigned int Result;
    VENDORSPECIFIC_DEV_DATA *pVSDev = & VSDevData ;

    Result = adi_dev_Write(    pVSDev->PeripheralDevHandle,
                               BufferType,
                               pBuffer);

    int i=0;

    return(Result);
}
```

USB ADI BULK类设备实现

一 编写USB类设备

3. 实现entry point 结构体中各个函数

定义了entry point 结构我们就要实现其中的每个函数。

```
static u32 adi_pdd_Control(          /* Sets or senses a device specific parameter */  
    ADI_DEV_PDD_HANDLE PDDHandle,  /* PDD handle */  
    u32 Command,                    /* command ID */  
    void *pArg                      /* pointer to argument */  
)
```

在控制函数我们需注意一个命令：

ADI_USB_CMD_CLASS_ENUMERATE_ENDPOINTS

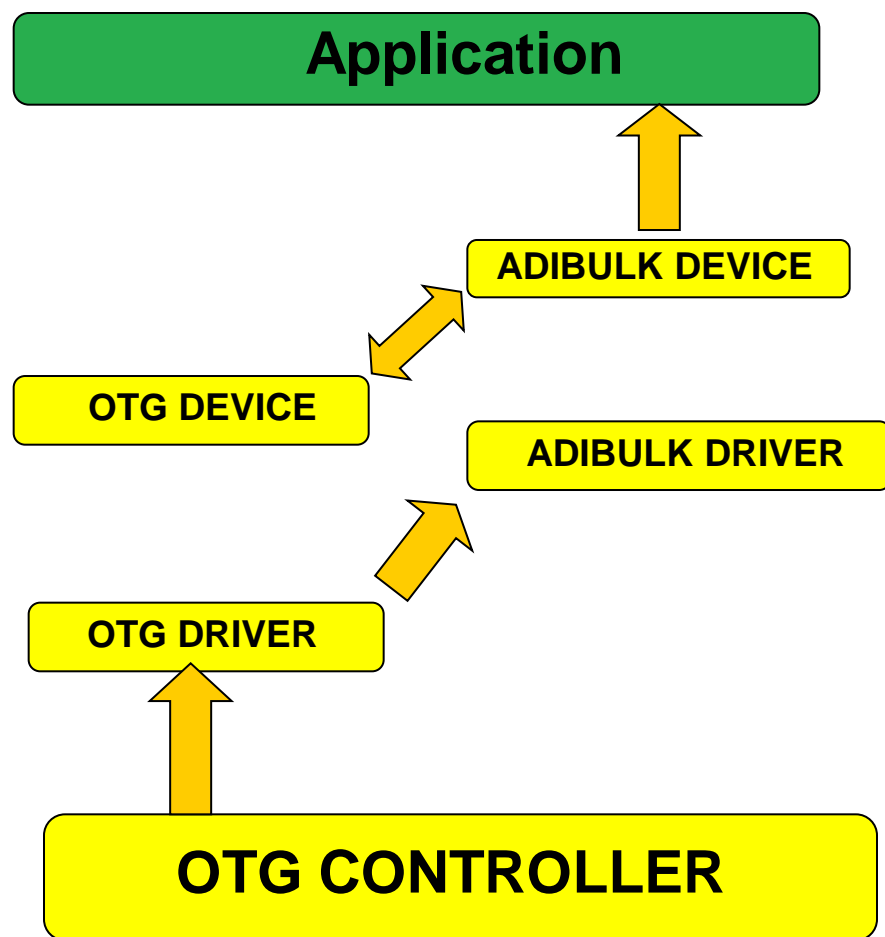
提供给应用程序对当前设备使用的ENDPOINT进行查询，因为在读写函数中均指定了进行操作的ENDPOINT号，其实现函数如下：

```
static s32 EnumerateEndpoints(ADI_ENUM_ENDPOINT_INFO *pEnumEpInfo);
```

其它命令参见具体实现。

USB ADI BULK类设备实现

一 编写USB应用程序



至此我们已经实现了一个USB BULK类设备，接下来需要进行应用层编程，结合我们应用层编程的典型模型可以得到：

usb应用程序的一般实现

1. 初始化device manager
2. 打开 OTG Device
3. 打开 ADIBULK Device
4. 关联OTG和ADIBULK Device
5. 配置设备相关参数
6. 进行读写操作

USB ADI BULK类设备实现

一 编写USB应用程序

1. 初始化设备 Device manager

```
/* initialize device manager */
Result = adi_dev_Init( DevMgrData , /* ptr to memory for use by DevMgr */
                      sizeof( DevMgrData ), /* size of memory for use by DevMgr */
                      &ResponseCount, /* returns number of devices DevMgr can support */
                      &DeviceManagerHandle, /* ptr to DevMgr handle */
                      & CriticalRegionData ); /* ptr to critical region info */

if (Result != ADI_DEV_RESULT_SUCCESS)
    failure();
```

2. 初始化设备 Usb Core

```
/* Initialize USB Core */
adi_usb_CoreInit((void*)&Result);
```


USB ADI BULK类设备实现

一 编写USB应用程序

3. 打开 USB 控制器和 BULK 类设备

```
/* Open controller driver */
Result = adi_dev_Open( DeviceManagerHandle, /* DevMgr handle */
                      & ADI_USBDRC_Entrypoint, /* pdd entry point */
                      0, /* device instance */
                      (void*)1, /* client handle callback identifier */
                      &PeripheralDevHandle, /* device handle */
                      ADI_DEV_DIRECTION_BIDIRECTIONAL, /* data direction for this device */
                      DMAHandle, /* handle to DmaMgr for this device */
                      NULL, /* handle to deferred callback service */
                      ClientCallback); /* client's callback function */

if (Result != ADI_DEV_RESULT_SUCCESS)
    failure();

/* Open the vendor specific Bulk USB class driver */
Result = adi_dev_Open( DeviceManagerHandle, /* DevMgr handle */
                      & ADI_USB_VSBulk_Entrypoint, /* pdd entry point */
                      0, /* device instance */
                      (void*)0x1, /* client handle callback identifier */
                      &DevHandle, /* DevMgr handle for this device */
                      ADI_DEV_DIRECTION_BIDIRECTIONAL, /* data direction for this device */
                      DMAHandle, /* handle to DmaMgr for this device */
                      NULL, /* handle to deferred callback service */
                      ClientCallback); /* client's callback function */

if (Result != ADI_DEV_RESULT_SUCCESS)
    failure();
```

USB ADI BULK类设备实现

一 编写USB应用程序

4. 为我们的设备指定VID和PID

```
/* get a pointer to the device descriptor which is maintained by the usb core */
pDevDesc = adi_usb_GetDeviceDescriptor();
if (!pDevDesc)
    failure();

/* customize the device descriptor for this device */
pDevDesc->wIdVendor = USB_VID_ADI_TOOLS;
#if defined(__ADSPBF527__)
pDevDesc->wIdProduct = USB_PID_BF527KIT_BULK;
#elif defined(__ADSPBF548__)
pDevDesc->wIdProduct = USB_PID_BF548KIT_BULK;
#elif defined(__ADSPBF526__)
pDevDesc->wIdProduct = USB_PID_BF526KIT_BULK;
```

5. 关联ADIBULK类设备和OTG控制器设备，并且对设备进行配置

```
/* configure the controller handle */
Result = adi_dev_Control( DevHandle, ADI_USB_CMD_CLASS_SET_CONTROLLER_HANDLE, (void*)PeripheralDevHandle);
if (Result != ADI_DEV_RESULT_SUCCESS)
    failure();

/* configure the class */
Result = adi_dev_Control( DevHandle, ADI_USB_CMD_CLASS_CONFIGURE, (void*)0);
if (Result != ADI_DEV_RESULT_SUCCESS)
    failure();
```

此时会调用VSBulkConfigure()
函数

USB ADI BULK类设备实现

一 编写USB应用程序

6. 配置缓冲区方式并使能USB

```
/* configure the controller mode */
Result = adi_dev_Control( DevHandle, ADI_DEV_CMD_SET_DATAFLOW_METHOD, (void*)ADI_DEV_MODE_CHAINED);
if (Result != ADI_DEV_RESULT_SUCCESS)
    failure();

/* enable data flow */
Result = adi_dev_Control( DevHandle, ADI_DEV_CMD_SET_DATAFLOW, (void*)TRUE);
if (Result != ADI_DEV_RESULT_SUCCESS)
    failure();

/* enable USB connection with host */
Result = adi_dev_Control( PeripheralDevHandle, ADI_USB_CMD_ENABLE_USB, (void*)0);
if (Result != ADI_DEV_RESULT_SUCCESS)
    failure();
```

至此我们的USB已经配置完成，可以进入到数据读写过程中。

USB ADI BULK类设备实现

一 编写USB应用程序

7. 在回调函数中等待事件发生，对设备进行读写操作

```
u32 usb_Read(ADI_DEV_DEVICE_HANDLE DeviceHandle, /* device handle */
             ADI_DEV_BUFFER_TYPE BufferType, /* buffer type */
             ADI_DEV_BUFFER *pBuffer, /* pointer to buffer */
             bool bWaitForCompletion) /* completion flag */
{
    u32 Result = ADI_DEV_RESULT_SUCCESS;
    ADI_DEV_1D_BUFFER *p1DBuffer;

    p1DBuffer = (ADI_DEV_1D_BUFFER*)pBuffer;

    /* Place the Read Endpoint ID */
    p1DBuffer->Reserved[BUFFER_RSVD_EP_ADDRESS] = g_ReadEpID;

    /* if the user wants to wait until the operation is complete */
    if (bWaitForCompletion)
    {
        /* clear the rx flag and call read */
        g_bRxFlag = FALSE;
        Result = adi_dev_Read(DeviceHandle, BufferType, pBuffer);

        /* wait for the rx flag to be set */
        while (! g_bRxFlag)
        {
            /* make sure we are still configured, if not we should fail */
            if (! g_bUsbConfigured)
                return ADI_DEV_RESULT_FAILED;
        }

        return Result;
    }

    /* else they do not want to wait for completion */
    else
    {
        return (adi_dev_Read(DeviceHandle, BufferType, pBuffer));
    }
} /* end usb_Read */
```

指定
ENDPOINT

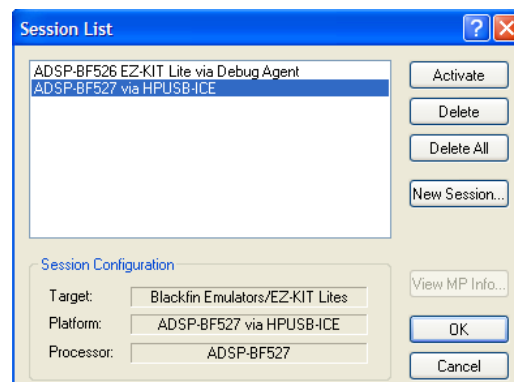
该变量是在回调函数中被置位的

USB ADI BULK类设备实现

一 测试 USB BULK应用程序

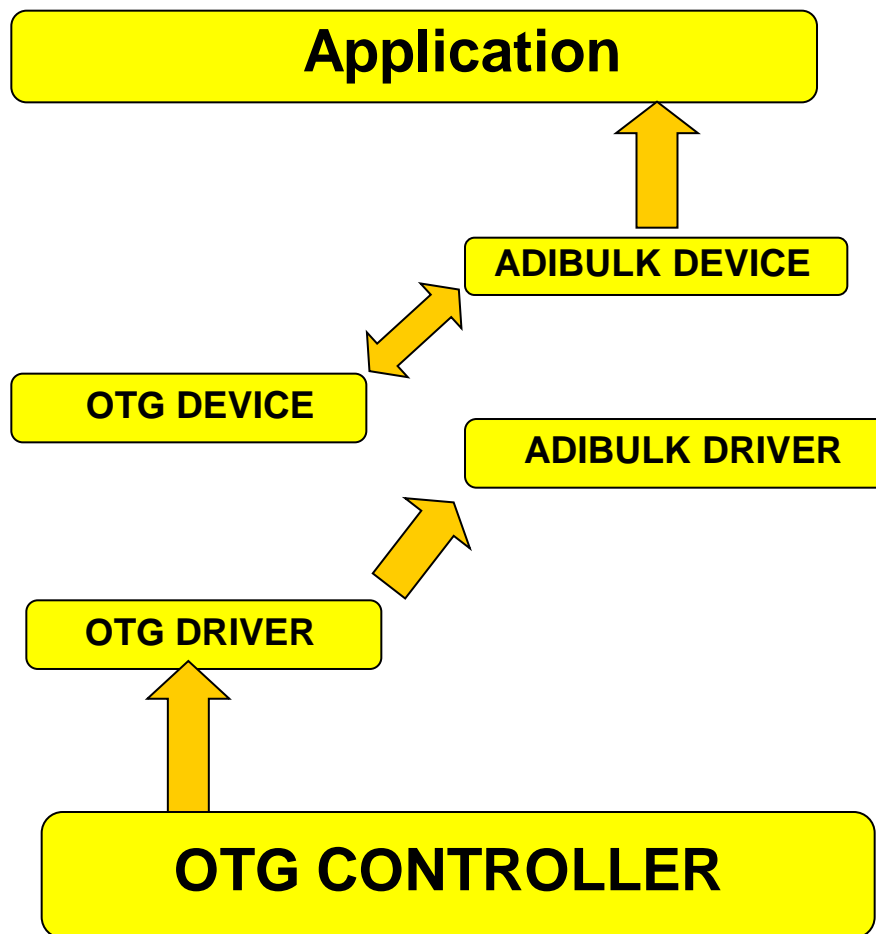
我们可以对BULK应用程序进行测试了

1. 连上仿真器后加电，配置Session如图



2. 打开<VisualDSP 5.0>\Blackfin\Examples\ADSP-BF527 EZ-KIT Lite\Drivers\usb\bulk_loopback_app目录下的bulk_loopback_app_bf527.dpj
3. 执行F7编程程序，执行F5执行程序。
4. 当windows发现在新硬件后，添加<VisualDSP 5.0>\Blackfin\Examples\usb\host\windows\drivers目录下的驱动程序
5. 在命令行下运行<VisualDSP 5.0>\Blackfin\Examples\usb\host\windows\drivers\hostapp目录下的hostapp.exe -l

USB ADI BULK类设备实现



这就是我们USB从下到上的一个应用开发过程，我们可以借助这个实例进而开发出其它的USB应用程序，借助于ADI提供的技术资源进行开发，无论从硬件还是软件都能节省开发时间，缩短产品上市时间

芯片对比参考

ADI BF527与TI AM1808 USB对比

	BlackFin 527	AM1808
Speed	In host mode, the USB module supports transfers at high-speed (480Mbps), full-speed (12Mbps), and low-speed (1.5Mbps) rates.	usb 2.0 host at speeds HS,FS,and low speed(LS)
	Peripheral mode supports the high- and full-speed transfer rates.	USB 2.0 peripheral at speeds high speed and full speed
Transfer Mode	support all the four transfer modes	support all the four transfer modes
Endpoint&FIFO	7 transmit endpoints and 7 receive endpoints in addition endpoint 0	4 transmit endpoints and 4 receive endpoints in addition to endpoint 0
	7.232K Bytes of FIFOs for packet buffering	4k endpoints RAM
Other	software-controlled clock control on each endpoint for power reduction	
	backwards compatible with existing USB 1.1 hosts	
	eight DMA master channels	
	double buffers and single buffer mode	