Blackfin在线培训课程

课程单元：Blackfin®设备驱动程序
主讲人：David Lannigan

第8b节：UART程序次序
第8c节：构建/运行UART举例
第9章：结束语
第9a节：更多信息

第1章：简介

第1a节：概述

大家好，我是模拟器件公司（ADI）DSP和系统设计部门的工程师，我的名字叫David Lannigan。今天我们将讨论面向Blackfin处理器家族的设备驱动程序模型。

参加本课程的用户应当深刻理解Blackfin处理器架构，并且学习了BOLD培训课程中的系统服务程序单元。

在今天的课程中，我将首先介绍一些关于设备驱动程序的背景信息、一般规范和术语。然后，我们要讨论设备驱动程序API，并逐一解释API中包含的函数。我们还要讨论缓冲区，设备驱动程序将通过缓冲区，调用和处理应用提供的数据。之后，我们将讨论设备驱动程序采用的各种数据流方法，其实就是设备驱动程序在缓冲区中处理数据的过程。最后，我们将利用2005年12月发布的VisualDSP工具套件更新版，以UART设备驱动程序为例，演示关于设备驱动程序的诸多概念。

第2章：设备驱动程序模型

第2a节：概述

Blackfin处理器的设备驱动程序模型采用了标准化的API，也就是说，所有处理器和驱动程序均使用相同的API。对开发人员而言，只要一次性掌握这些API，就能一劳永逸。所有驱动程序均以完全相同的方式作用。这些API也是可扩展的，允许用户根据需要添加命令、事件和返回码等，其目的是支持绝大多数器件。目前，我们可以支持的器件包括SPORT串行端口、SPI、PPI、TWI和诸如以太网端口和USB端口等高级器件。当然，凡事都有例外。虽然我们的驱动程序模型旨在支持绝大多数器件，但是也会有某些器件不适用于这种模型。不过，这个模型非常适用于我们范围广泛、种类繁多的设备驱动程序。

这个设备驱动程序模型基于Blackfin处理器的系统服务程序，具备一个非常稳定的软件基础，免除了许多重复的编程任务，例如，每个设备驱动程序中都包含了DMA代码。设备驱动程序将自动调用系统服务程序。例如，PPI驱动程序就可以利用DMA服务程序，通过DMA来传输数据。通过利用系统服务程序，我们实现了一个模块化程度极高的软件环境，提供了更加杰出的兼容性，驱动程序之间的相互操作十分简单、明了。开发人员可以轻松地在同一个应用中整合多个驱动程序，让它们并肩工

作。这个设备驱动程序模型也是可移植的，BF533处理器的驱动程序也可以在其他Blackfin处理器，如BF537处理器甚或BF561双核处理器上发挥完全相同的作用。

这个框图显示的是一个应用的基本架构，最上面是应用。应用可以选择安装RTOS，如VDK或第三方操作系统，也可以采用所谓的“独立式”运行模式。接下来是设备驱动程序。如图所示，设备驱动程序位于系统服务程序上方，驱动程序将调用系统服务程序，如中断。当某个驱动程序想要控制一个中断时，它将调用中断管理器。如需使用我先前提到的DMA，则调用DMA服务程序，诸如此类。

第2b节：使用设备驱动程序

在VisualDSP中使用设备驱动程序的步骤非常简单。打开一个项目，在应用的源文件中，应当包括三行代码。这张幻灯片上列出了第一行“#include<services/services.h>”。这个包含文件中包括了所有的系统服务程序，如DMA、中断、端口控制等服务程序。接下来是设备管理器的包含文件“adi_dev.h”，其中包括了所有关于设备驱动程序的一般信息，如API、返回码、事件代码以及所有的设备驱动程序一般信息。第三类文件应当包含设备驱动程序本身，即，特定设备驱动程序的包含文件。如果构建的应用使用了UART，那么，其源文件中就应包括UART’s .h文件，其中囊括了关于UART设备驱动程序的所有特定信息。稍后我将举例演示。

对于诸如编解码器、视频编码器、视频解码器、片外以太网控制器等外接器件的片外设备驱动程序，程序员应当在源文件列表中，包含这些设备驱动程序的“.C”文件、其“名称.C”文件、或者取决于驱动程序，其“.C”文件集。由于库非常大，而我们又无从提前知晓各个应用将使用哪些外接器件，所以无法为所有外接器件提供库。为了节省存储空间，应用只需要根据自身需要，选用适当的驱动程序。

在连接器文件夹中，应当包含适当的系统服务程序库，即libssl库；以及适当的片上器件的设备驱动程序库，所有此类库均以“libdrv”为前缀。稍后我将演示如何选择和使用库。这里，我要强烈推荐用户选择连接器属性设置界面上提供的“代码消除选项”，它会将某个特定应用中毫无用处的代码全部删除，从而大大缩短代码长度。例如，如果我们仅使用了某个设备驱动程序中的A、B、C函数，那么，该选项将删除这个设备驱动程序中未使用的D、E和F函数。这个选项能够为嵌入式应用节省珍贵的代码存储空间。

第2c节：片上驱动程序库

片上驱动程序库分两种，一种是调试版，一种是发布版。强烈建议用户在项目开始时，采用调试版驱动程序库。在调试版中，所有编译器优化选项均已禁用；其中包含了符号信息；并且允许程序员逐行查看源代码，弄清这些代码的作用。此外，调

试版驱动程序库将执行大量的参数检查，因此，如果应用提供的参数中包含错误的值，调试版库将尽全力检查出这些错误，并返回相应的错误代码。

用户在开始时应当采用这种库；范例、演示等通常也采用调试版库来进行演示。

当用户认为驱动程序的运行情况达到要求时，可以更换为发布版片上驱动程序库。在发布版库中，所有编译器优化选项均已启用，因此，应用性能将大幅提升。不过发布版库中不包括符号信息，因此，程序员很难通过查看源代码，随时了解应用的执行情况。此外，发布版库只会执行少量的参数检查，因为我们认为，当用户选择采用发布版库时，很清楚自己在做什么，所以我们也可以卸下繁重的参数检查任务。用户在项目完成时，才需要使用发布版库。也就是说，利用调试版库进行产品开发，当应用达到性能目标后，再切换至发布版库。

如何选择库？设备驱动程序采用了与VisualDSP相同的命名约定，所有片上外设的设备驱动程序库的前缀均为“LIBDRV”，表示面向设备驱动程序的库。紧接着的三个字符是表明该库适用的处理器的变量。例如，如果紧接着的三个变量为532，则表示该库适用于BF531、BF532和BF533 Blackfin处理器，变量534则表示适用于BF534、BF536和BF537处理器，变量561表示适用于BF561双核处理器。在这个例子中，省略了再接下来的三个字符。如果驱动程序要求操作系统提供特殊支持，那么，这三个字符将表明适用于该特定驱动程序的操作系统。目前我们提供的所有设备驱动程序库都不要求RTOS提供任何特殊支持。所以，在本例中，“bbb”三个字符为空。我们提供的设备驱动程序既可支持独立式系统，又可用于VDK操作系统。

再后面的两个字符用于表明该库的任何特定条件，可以是“d”或“y”或者这两个字母的组合。如果这里只有一个“d”字母，则表示该库为调试版；没有“d”字母则表示该库为发布版；字母“y”则表示该库是适用于芯片异常情况的临时解决方案。如果库的名称中包含字母“y”，则表示该库包含了适用于各种修订版芯片的临时解决方案。“dy”字母组合则表示该库是调试版，并且包含了所有的临时解决方案。如果为空则表示该库不是调试版，也就是说，是发布版库，并且其中不包括任何临时解决方案。

如需了解关于设备驱动程序的更多信息，可以参阅“Blackfin\include\drivers”目录下的驱动程序包含文件。这张幻灯片上列出了用户安装VisualDSP时的文件默认保存位置。在这个目录下，用户可以找到所有设备驱动程序的包含文件。“Blackfin\lib\src\drivers”目录下的源文件中包含了我们目前拥有的所有设备驱动程序的源代码。设备驱动程序库和其他VisualDSP库保存在“Blackfin\lib”目录下。

我们还提供了面向BF533 EZ-Kit、BF537 EZ-Kit和BF561 EZ-Kit等评估板的范例。这些范例均在“Blackfin\EZ-KITS”目录下，特定目标处理器的范例则在相应的子目录下。

我们提供了两大类设备驱动程序技术文档，一类是用户手册，用户可以在Analog.com网站的“Technical Library（技术资料库）”页面下载。另一类技术文档是2005年9月发布的一个用户手册附录，其中提供了关于更新版本服务程序和设备驱动程序的最新功能的信息，用户可以在下面这个ADI公司网站，下载该技术文档，

第3章：设计要素

第3a节：存储器

我们的设备驱动程序均不使用动态存储器，所有驱动程序的存储空间都是静态分配的。不过，设备管理器自身需要利用存储器来管理系统中的各种器件。需要同时运行的设备驱动程序越多，设备管理器对存储空间的要求就越高。物理驱动程序，如面向PPI、SPORT和UART等器件的低级驱动程序，使用静态存储空间来保存所有内部数据，因此，无需向这些驱动程序动态分配存储空间。总之，应用无需向任何设备驱动程序动态分配存储空间，而是在应用时间，向设备管理器静态分配或提供存储空间。不论是片上内存还是外接SDRAM存储器，这完全由应用决定。用户可以根据应用的需要，将代码或数据保存到任何存储位置。

第3b节：句柄

我们利用句柄来进行通信或者标识出驱动程序模型中的各个组件。当一个设备驱动程序打开时，它将向应用传递一个句柄。此后，每当应用需要引用该设备驱动程序时，它都要传递这个设备句柄，以告知系统，在何时何地调用了哪一个设备驱动程序。设备句柄其实就是一个独一无二的身份标识，表明了保存管理特定设备驱动程序所需的所有信息的存储地址。反过来，当一个设备驱动程序打开时，应用也会向设备管理器提供一个针对该设备驱动程序的句柄，即客户端句柄。应用或客户端可以随意设置客户端句柄，它对设备驱动程序没有太大意义，只要客户端自己认得这个句柄就行。设备驱动程序通过客户端句柄与应用进行通信。以异步事件为例，当发生错误或者某个缓冲区满载之类的事件时，设备驱动程序就要通知应用发生了该异步事件，应用根据设备驱动程序发送给自己的客户端句柄，可以知道是哪个设备驱动程序发生了该事件。

第3c节：结果代码

几乎所有的API函数都会返回一个结果代码。虽然有两个显而易见的例外情况，但是总的来讲所有函数都会返回一个结果代码。零值表示成功，也就是说，如果设备驱动程序，确切说是设备驱动程序API调用，返回了一个零值，则表示该调用成功完成任务。如果返回的是一个非零值，则表示发生了某种类型的错误，或者向应用反馈了传达某种信息的结果。所有驱动程序都拥有一套独一无二的返回码。系统中

的每个驱动程序都各自具备一套唯一的返回码。这些返回码可以是u32值或者无符号32位值。应用将检查接收到的返回值，查明该值是代表错误还是传达某种信息的结果。例如，我们打开了一个串行端口驱动程序，该串行端口驱动程序必须挂接一个中断，比如说串行端口错误中断。该设备驱动程序将打开中断，正确地挂接中断，一切都有条不紊地顺利完成，最后，它会向应用返回一个零值。但是，如果有什么原因导致该设备驱动程序无法挂接该中断，那么，它将返回一个错误代码，表明未能挂接中断，应用必须采取某种措施。

如果应用收到了一个非零值，那么，只要查阅“services.h”文件，就能快速查明发生了什么错误。这个文件中包含了一系列独一无二的数字，分别是各个系统服务程序或设备驱动程序将反馈给应用的返回码。然后，再在相应的“.h”文件中查询该特定值的含义，就能了解该非零值代表了什么错误或信息。

第3d节：初始化

应用必须按照特定的步骤执行初始化。由于设备驱动程序是基于系统服务程序的，所以，必须先对系统服务程序进行初始化，然后，再初始化所谓的设备管理器。设备管理器负责向所有的设备驱动程序提供API，也就是说，设备管理器在某种意义上掌管着系统中所有的设备驱动程序。由于驱动程序位于服务程序上方，因此应当首先初始化服务程序，再初始化设备管理器。这张幻灯片列出了必须遵循的特定顺序。首先要初始化中断管理器，然后初始化EBIU，即外部总线接口单元，然后依次是电源管理、端口控制、延迟回调、DMA管理器、标志控制、定时器等服务程序，最后是设备管理器。无需使用的服务程序不用进行初始化。只要按照这个特定顺序正确地完成所有的初始化，就不会发生任何冲突，应用不会在未完成初始化的情况下试图调用任何服务程序或设备驱动程序。

第3e节：终止

同样地，应用也必须遵循特定的终止顺序。嵌入式系统应用程序通常无需终止，它将永不停息地运行。如果需要进行终止，则必须按照特定的顺序，终止堆栈。本幻灯片列出了终止顺序。这个顺序应该和初始化顺序恰恰相反，也就是说，应当按照与初始化顺序完全相反的顺序，首先终止设备驱动程序，然后再终止系统服务程序。在本例中，首先终止设备管理器，然后依次终止定时器、标志控制、DMA管理器、回调控制、端口控制、电源管理和EBIU等服务程序，最后终止中断管理器。

第3f节：关于RTOS的考虑

我们目前的设备驱动程序与RTOS之间不存在任何相关性。例如，我们没有提供针对模拟器件公司的VDK RTOS的特定设备驱动程序库，这些设备驱动程序既可以在独立模式下运行，又可以支持RTOS。服务程序或驱动程序与RTOS之间的所有交互作用，均以独立的代码段形式，包含在系统服务程序中，因此，我们针对独立式系统和

RTOS系统，提供了不同的系统服务程序，而设备驱动程序自身则不具备任何相关性。需要注意的是，不论是在RTOS环境下，还是在独立式环境下，设备驱动程序的API都是完全相同的，所以，当应用在独立式环境和RTOS环境之间进行切换时，无需对函数调用进行任何更改。

第4章：设备驱动程序API

第4a节：概述

API非常简单。在这张幻灯片上，左侧是应用，右侧是设备驱动程序。API中总共有6个函数，凡是用过设备驱动程序的程序员都应该非常熟悉这些函数。其中5个是常见的函数，包括：打开、关闭、读、写和控制函数，还有一个是我们增加的“SequentialIO”函数。接下来，我将逐一介绍这几个函数及其功能。

第4b节：API函数

显然，“adi_dev_Open()”函数负责打开设备驱动程序。当应用想要使用某个驱动程序时，它首先要调用“adi_dev_Open()”函数。如果应用决定不再需要使用该设备驱动程序，那么，它将调用“adi_dev_Close()”函数。“adi_dev_Close()”函数的功能是有序地关闭和释放应用使用的系统服务程序和硬件。如果应用为传入数据流或者双向数据流打开了某个设备驱动程序，那么，它将通过“adi_dev_Read()”函数，从器件读取数据。以连接至摄像头的PPI为例，读取数据过程就是利用“adi_dev_Read()”函数，通过PPI，读取摄像头捕捉到的数据。反过来，我们也提供了写函数。写函数的作用是通过设备驱动程序，发送数据。同样也以刚才的PPI为例，如果要将利用摄像头录制好的视频数据发送至显示器，那么，可以利用“adi_dev_Write()”函数，通过PPI，将该数据发送至显示器。

另外，还有一个“adi_dev_Sequential IO()”函数，允许应用指定特定的读写顺序。在没有调用“Sequential IO”函数的情况下，应用会尽快完成读操作或写操作，但两者之间并未实现同步。以DSL调制解调器为例，其下载速度通常比上传速度更快，也就是说，其读操作的速度比写操作快。

对于要求以特定顺序执行读操作和写操作的器件，应用就可以利用“adi_dev_Sequential IO()”函数，实现以特定的顺序，先执行一些读操作，再执行一些操作等等。

最后一个函数是“adi_dev_Control()”。过去，在版本较早的设备驱动程序中，这个函数有点像IOCTL或IOCTL函数。这个函数通常用于设置或检测设备驱动程序的某个参数。我将在稍后举例说明，如何使用“adi_dev_Control()”函数，设置UART的波特率、停止位等等参数。

应用只需要借助这6个函数，就能向设备驱动程序传递信息。而当设备驱动程序需要向应用反馈信息时，则要通过系统服务程序提供的事件机制和回调机制，与应用进行异步通信。当设备驱动程序需要通知应用发生了某个事件时，例如缓冲区已处理完毕或者器件发生错误等等，那么，设备驱动程序将调用应用的回调函数。这就是应用和设备驱动程序之间的所有API。这6个函数，以及设备驱动程序用于通知应用的应用回调函数。

总的来讲，设备驱动程序API非常简单。这里需要指出的是，我先前在介绍句柄时也说过，当应用调用“adi_dev_Open()”函数时，它将获得一个针对该设备驱动程序的句柄。之后，当应用调用其余5个函数，即adi_dev_Close()、adi_dev_Read()、adi_dev_Write()、adi_dev_SequentialIO()或adi_dev_Control()时，将以该句柄为这些API函数的第一个参数。这样，设备管理器就能知道，应用正在使用哪个设备驱动程序。而当设备驱动程序通过回调函数向应用发出事件通知时，设备驱动程序也会同时将其获得的客户端句柄，返回给应用。这个客户端句柄值也是包含在“adi_dev_Open()”函数中，传递给设备驱动程序的。通过客户端句柄，设备驱动程序可以向应用反馈信息。

不论哪种设备驱动程序，都包含这6个API函数。UART、PPI、SPI、SPORT、TWI、以太网端口或USB端口等，所有这些设备驱动程序的API都是完全相同的。即使诸如ADC、DAC、视频编码器、视频解码器等外设，也要使用这6个API函数。关于设备驱动程序的一般信息的.h文件“adi_dev.h”中阐明了这些API。每个设备驱动程序都可以扩展其API。用户可以添加自定义指令；例如，应用可以设置或检测自己的IOCTL值或某些独特参数。以DAC为例；用户可能希望为某个DAC设置容量；所有设备驱动程序都允许用户扩展控制函数或者可以通过控制函数传递给设备驱动程序的指令。此外，每个设备驱动程序都能为应用创建新的返回码，这样，应用就能通过API函数调用，传达更加明确的信息。除了一般信息文件“adi_dev.h”中提供的返回值，各个设备驱动程序还能自定义其他返回值。

设备驱动程序还能创建事件，用于通知应用。在标准的一般信息文件“adi_dev.h”中，我们提供了缓冲完毕和错误等事件。但是，如果设备驱动程序需要向应用通知某种特殊情况，那么，也可以扩展该设备驱动程序的API，创建更多事件。例如，一个片外控制器可能决定进入睡眠模式，并需要通知应用它即将进入睡眠模式，那么，它就可以为此创建一个事件，并向应用发出该事件通知。

第4c节：与系统服务程序之间的交互作用

在课程开始时我曾说过，设备驱动程序基于系统服务程序。得益于此，创建设备驱动程序变得轻而易举，同时，我们也能利用系统服务程序有效管理Blackfin处理器的各种资源，例如DMA。PPI设备驱动程序在需要通过DMA传输数据时，将调用DMA管理器，打开DMA通道、提供描述符等等；我们只需要在系统服务程序中实现DMA管理器调用函数，而不必在所有需要使用DMA的驱动程序中逐一实现该调用函数。只需

要在系统服务程序中实现一次，所有的设备驱动程序都能通过该系统服务程序，调用DMA。中断管理器、定时器和延迟调用等也一样。PPI设备驱动程序可以使用所有这些功能，它只需要调用相应的系统服务程序，就能使用该功能。对应用而言，它只需要初始化这些系统服务程序，然后，设备驱动程序将根据需要，自动调用相应的系统服务程序。还是以PPI设备驱动程序为例，当它需要使用DMA时，如果应用已经初始化了DMA服务程序，那么，PPI驱动程序将直接通过DMA服务程序调用DMA。这样，应用的任务就变得非常简单，它只需要通过设备驱动程序API发出命令，然后，设备驱动程序将自动调用相应的系统服务程序。

第5章：数据传输

第5a节：缓冲区概述

在利用设备驱动程序，向器件输出数据或者从器件输入数据的过程中，数据要保存到缓冲区中。在传输方向上，缓冲区分为两类：传入缓冲区，用于填充从器件读取的数据；和传出缓冲区，用于保存应用需要通过器件输出的数据。我们提供了多种不同类型的缓冲区；包括一维缓冲区，即传统的线性缓冲区。二维缓冲区，可以直接利用Blackfin处理器架构实现的2D DMA特性；对于需要读取较大帧中的宏块的视频应用，这个特性非常有用。此外，我们还提供了顺序缓冲区，可用于执行“Sequential IO”函数。如今，顺序缓冲区也是一维缓冲区，是一个允许用户规定特定的读、写顺序的线性缓冲区。最后，我们还提供了循环缓冲区，这些缓冲区可以直接利用Blackfin处理器提供的自动缓冲特性。如果我们向循环缓冲区填充一个巨大的数据块，那么，设备驱动程序将在这个循环缓冲区连续不断地反复处理。

第5b节：一维缓冲区

下面，我们首先讨论一维缓冲区。在这几种缓冲区中，一维缓冲区最简单，这里列出了一维缓冲区中存在的字段。第一个字段是“指向数据的指针”。数据可以保存在任何存储位置，所以，缓冲区和数据可以位于不同的位置。这样，我们就不必来回移动数据，也不用将缓冲区移动至数据所在位置。我们可以传输大量数据，例如在视频应用中，一个视频帧就包含约1 Mb数据。我们可以将缓冲区和其中包含的实际数据分别置于不同的存储位置。下面这个字段是“元素数量”，表明了指针指向的这个数据包含多少个元素，这个数据有多大。再下来的字段是“元素宽度”，即该数据中的每个元素的字节长度。

下面以1,024字节长的一段数据为例。通常，一维缓冲区会将其表示为元素数量为1,024，元素宽度为1，即1个字节长。如果是1,024个16位数据，那么，元素数量仍将为1,024，元素宽度则为2，因为该数据有2个字节长。每个缓冲区都具备一个“回调参数”，这个回调参数的值表明了当该缓冲区处理完毕后，设备驱动程序是否需要通知应用。如果回调参数为“空”（NULL），那么，当缓冲区处理完毕后，将不会执行回调。例如，一个应用通过一个器件发送一个缓冲区，如果应用在创建

缓冲区时将回调参数设置为空，那么，设备驱动程序在通过器件成功发送了该数据后，将不会通知应用它已经完成任务。如果回调参数值为非空，则表示，应用希望设备驱动程序在通过器件成功发送数据后，通知自己。设备驱动程序将回调应用，告知应用缓冲区中的数据已经成功发送。实际上，它将通过应用回调函数，将回调参数值传递回应用。在这个过程中，应用可以做出不同的选择；它可以命令设备驱动程序发送数据、填充数据或者从器件读取数据，并且在完成任务后不需要通知自己；或者，它可以命令设备驱动程序通过器件发送数据或从器件读取数据，并且在该缓冲区处理完毕后，通知自己。总之，应用可以选择是否要求设备驱动程序通知自己。

接下来的字段是“处理标记”，设备驱动程序将向缓冲区填充这个字段。这个简单的标记只有“真”（TRUE）和“假”（FALSE）两个值，表示设备驱动程序是否已经处理完毕该特定缓冲区。如果已经完成处理，设备驱动程序还会填充一个“处理数量”字段。处理数量字段表明了已处理完毕的缓冲区中的字节数。通常，处理数量应该等于元素数量乘以元素宽度，不过，也有不相同的情况，例如以太网驱动程序。在以太网系统中，线路传输的数据包设有最长长度限制。但是，并非所有的数据包都是最长长度，有些只是很小的数据包。应用可能向以太网驱动程序传递一个很大的缓冲区，并要求它将从以太网系统接收到的数据填充到这个缓冲区中，但是，它收到的可能只是很小的数据包，可能只是一个短数据包，那么，以太网驱动程序将通过处理数量，告知应用它实际向缓冲区填充了多少字节数据。

再接下来的字段是“pNext”，指向链接中的下一个缓冲区。这是一个非常重要的概念，我将在后面几张幻灯片中详细解释。我们可以将多个缓冲区链接起来，从一个缓冲区指向另一个缓冲区，再指向第三个缓冲区，让设备驱动程序依次处理这一系列缓冲区。

最后一个字段是“补充信息”，这个字段有点像是一个杂物箱。目前还没有任何设备驱动程序在使用这个字段。稍后，我将举例说明如何向这个字段填充数据，如何使用这个字段。不过也许未来的设备驱动程序会用到这个字段，例如，如果要将一些未曾预料的信息放到缓冲区中，应用和驱动程序就可以借助这个字段，包含一些我们现在没想到的信息。

第6章：数据流方法

第6a节：概述

目前有5种基本的数据流方法。数据流方法阐明了设备驱动程序处理数据的方式。在上一张幻灯片中，我曾提到简单的缓冲区链接，我们可以将一维缓冲区和二维缓冲区链接起来，创建一个包含1个、2个、3个或更多个缓冲区的链接。我们可以随时随地向设备驱动程序提供任意数量的缓冲区，对于排队中的缓冲区数量以及设备

驱动程序正在处理的缓冲区数量，没有任何限制。用户可以根据应用的需要，提供相应数量的缓冲区。

然后是环回链接。在环回链接中，设备驱动程序将循环往复地连续处理链接中的所有缓冲区，稍后我将详细解释。再下来是顺序链接，即缓冲区以特定顺序形成链接，在前面讨论以特定顺序执行读写操作时我也曾提到这种顺序链接。顺序链接也可以形成环回。此外，还可以实现循环数据流方法，该方法正好对应于Blackfin处理器的自动缓冲功能。应用向设备驱动程序提供一个连续的存储块，然后，设备驱动程序将连续不断地反复处理该存储块。

虽然我们提供了5种不同的数据流方法，但是，就某一种设备驱动程序而言，它不一定全部支持这5种数据流方法。有些方法可能并不适用于某种特定的设备驱动程序，不过，设备驱动程序必须支持至少1种数据流方法。例如，PPI驱动程序可以支持一维缓冲区、二维缓冲区和循环缓冲区。UART驱动程序可支持一维缓冲区，我们稍后将以UART驱动程序为例进行演示；TWI是一种兼容I^2C规范的协议，可以支持顺序一维缓冲区。对于这种接口，应该说是这种协议，需要采用特定的读写顺序。

第6b节：简单链接

下面，我将演示简单链接数据流方法及其作用机制。在简单链接中，缓冲区排成队列等待设备驱动程序处理。缓冲区排队分两种，一种是传出缓冲区排队，应用将通过“adi_dev_Read()”函数，向设备驱动程序提供这个排队中的缓冲区。然后，设备驱动程序将向这些缓冲区填充其从器件读取的数据。此外，我们还提供了一个传出缓冲区。传出排队中的缓冲区包含了应用希望通过器件发送出去的数据。设备驱动程序将以先入先出（FIFO）的顺序，处理这两个排队中的缓冲区，即按照接收到这些缓冲区的顺序，逐一进行处理，并且是异步处理。以我刚才提到的DSL调制解调器为例，读缓冲区的处理速度可能比写缓冲区快，因此，设备驱动程序对读、写缓冲区的处理是异步的。

应用可以随时向设备驱动程序提供缓冲区，可以在应用时间或中断时间，应用可以随时向设备驱动程序提供需要进行处理的缓冲区。应用每次可以提供一个缓冲区也可以提供一组缓冲区，也就是说，应用既可以单独地提供一个又一个缓冲区，也可以将缓冲区链接起来，比如说链接10个缓冲区，然后，一次性将这个包含10个缓冲区的链接，提供给设备驱动程序。每个缓冲区都可以分别指向不同长度的数据。不要求将所有缓冲区都指向固定长度的数据，可以将一些缓冲区指向128字节长的数据，而将链接中的另一些缓冲区指向1,024字节长的数据。用户可以在某个缓冲区排队中混合搭配不同容量的缓冲区。

应用可以将任何或全部无缓冲区标记为生成回调，也可以不标记任何缓冲区。设备驱动程序利用回调机制，通知应用缓冲区已处理完毕。在这张幻灯片中，如果我们从左至右处理这个链接中的所有缓冲区，那么，当第一个缓冲区处理完毕后，设备

驱动程序将不会通知应用。而当处理完第二个贴有回调标签的缓冲区后，设备驱动程序将通知应用，“我已经处理完毕这个缓冲区”。在处理完后面两个缓冲区时，设备驱动程序也不会向应用发出回调。而当处理完倒数第二个缓冲区时，设备驱动程序会再一次通知应用它已经处理完毕该缓冲区。应用可以混合搭配，给任何缓冲区加贴回调标签、或者给所有缓冲区都加贴回调标签。有些应用只在链接中的最后一个缓冲区加贴回调标签，这样，当设备驱动程序处理完最后一个缓冲区时，将通知应用，然后应用就能知道前面所有的缓冲区都已处理完毕。

处理完毕后的缓冲区将不再使用，除非应用重新将其分配给设备驱动程序。这只是一个链接，设备驱动程序处理完最后一个缓冲区后，将停止工作。在下一张幻灯片中，我将介绍环回链接，以及如何实现环回链接。通过这个基本的简单链接，应用可以向驱动程序提供多个缓冲区，驱动程序将依次处理这些缓冲区，并在所有缓冲区都处理完毕后，返回应用域。

第6c节：环回链接

接下来，我们将讨论环回链接，通过这种有效的方法，应用可以向驱动程序提供一组缓冲区，然后，让设备驱动程序连续不断地循环处理这些缓冲区。我们还是以刚才介绍简单链接时列举的缓冲区序列为例。在简单的缓冲区链接中，设备驱动程序将在处理完最后一个缓冲区后停止工作；而在环回链接中，当设备驱动程序处理完最后一个缓冲区后，它将自动返回第一个缓冲区，并重新开始新一轮处理。这样，设备驱动程序处理的将是一个无限循环的缓冲区链接。不过，只有在数据流停止后，才能向设备驱动程序提供这种采用环回链接的缓冲区。这一点与直接的简单链接不同。我们刚才已经说过，利用简单链接，应用可以随时随地向设备驱动程序提供缓冲区。而采用环回链接，则只能在数据流停止后，向设备驱动程序提供缓冲区。这一点非常重要，因为如果应用正在处理一个缓冲区环回链接，同时又想添加一个缓冲区，那么，应该将这个缓冲区添加在环中的哪个位置？所以，我们要限制应用添加缓冲区，只能等该器件上运行的数据流停止后，再添加缓冲区。

一般而言，这种限制不会造成问题。应用通常只在进行初始化时才需要使用环回链接，应用将向设备驱动程序提供一组需要进行处理的缓冲区，然后，就不再需要为其重新提供缓冲区。在获得这些缓冲区后，设备驱动程序将循环往复地不断处理这些缓冲区。这个过程对系统而言，几乎不需要任何开销，应用不再需要调用“adi_dev_Read()”或“adi_dev_Write()”函数，设备驱动程序将自动地连续不断地反复处理这些缓冲区。如果设备驱动程序的任务是输出数据，那么，这些缓冲区将源源不断地向它提供数据；如果设备驱动程序的任务是从器件读取数据，那么，它可以将数据保存到这些缓冲区中，而永远不必担心会溢出。通过巧妙地利用回调，应用可以有效地管理这些缓冲区。

第6d节：顺序链接

接下来，我们讨论顺序链接。顺序链接与我们前面讨论过的简单链接很相似，只不过顺序链接没有为读和写操作分别提供一个排队，而是将所有的缓冲区都放到一个排队中。在顺序链接中，缓冲区中包含一个字段，表明其传输方向是传入还是传出。因此，如果设备驱动程序的任务是输出缓冲区中包含的数据，那么这个缓冲区将标记为传出缓冲区。如果设备驱动程序的任务是向缓冲区填充数据，那么这个缓冲区将标记为传入缓冲区。我们将这些传入和传出缓冲区放到一个排队中。驱动程序将按接收到这些缓冲区的顺序，逐一进行处理。和简单链接一样，采用顺序链接时，应用也可以随时随地向设备驱动程序提供缓冲区。我在这张幻灯片的底部，创建了一个顺序链接。采用顺序链接，应用可以随时随地提供缓冲区；可以一次一个地提供，也可以成组提供。这一次，我们只有一个排队，如图所示，每个缓冲区都被标记为传入缓冲区或传出缓冲区。此时，这个链接中只有3个缓冲区，第一个标记为用于传出业务，因此其中包含了设备驱动程序将要输出的数据。接下来的两个缓冲区标记为用于传入数据，因此，其中将包含设备驱动程序从器件读取的数据。

此外，和简单链接一样，每个缓冲区都可以指向不同长度的数据，也就是说，应用可以在这个链接中混合搭配不同容量的传入或传出缓冲区。比如，可以在这个链接中放入一些用于较短数据的传入缓冲区，再放入一些用于较长数据的传出缓冲区；对缓冲区的容量没有限制，不要求必须是固定长度。同样地，对缓冲区的数量也没有限制，可以随时随地向排队添加任何数量的缓冲区。缓冲区数量主要取决于系统的存储容量。差不多所有采用链接方法的缓冲区都是这样。

同样，应用可以将任何或全部无缓冲区标记为生成回调，也可以不标记任何缓冲区。处理完毕后的缓冲区将不再使用，除非应用重新将其分配给设备驱动程序。在简单的顺序链接中，设备驱动程序将首先处理排队中的第一个缓冲区，不论是传入缓冲区还是传出缓冲区，如果该缓冲区标记为生成回调，那么，设备驱动程序还要在处理完毕该缓冲区后通知应用。如果，该缓冲区未标记为生成回调，那么，设备驱动程序在处理完毕该缓冲区后将不会向应用发出通知，而是直接接着处理链接中的下一个缓冲区。这是一个功能强大的机制。目前，我们的TWI驱动程序，一个兼容I_2C规范的驱动程序，就采用了这种方法。熟悉I_2C规范的开发人员都知道，在这个简单的协议中，通常首先要执行写操作，即，阐明需要从中读取数据或向其提供数据的器件的地址；然后，再执行读操作。例如，设备驱动程序要从某个EPROM存储器读取数据，那么，它首先要发送该EPROM存储器的地址，然后，再从这个EPROM存储器中读取数据。利用顺序链接就可以实现这种特定顺序的读、写操作。在处理完链接中的最后一个缓冲区后，设备驱动程序将停止工作。如果应用又向设备驱动程序提供了更多缓冲区，那么，它将自动重新开始逐一处理这些缓冲区。

第6e节：环回顺序链接

顺序链接也可以实现环回。这和简单的环回链接有些相似，只不过形成环回的是一个包含传入缓冲区和传出缓冲区的顺序链接。与普通环回一样，设备驱动程序处理完链接中的最后一个缓冲区后，将自动返回第一个缓冲区，进行新一轮处理。同样

地，只能在数据流停止的情况下向驱动程序提供缓冲区，这通常是在执行初始化时。一般而言，应用实现环回顺序链接的步骤是：打开设备驱动程序、配置设备驱动程序、通过“adi_dev_SequentialIO()”函数向其提供缓冲区，包括数个传入缓冲区和传出缓冲区，还可以根据需要加贴回调标签。然后，发起数据流。此后，设备驱动程序将循环往复、连续不断地依次处理这些缓冲区。环回顺序链接非常方便实用，利用这个方法，应用只需要一次性向设备驱动程序提供适当的缓冲区，设备驱动程序就会周而复始地不断处理这些缓冲区。与普通的简单链接一样，数据将源源不断地填充到缓冲区中，驱动程序可以通过这些缓冲区持续不断地读取或发送数据，而这一切操作仅需要很低的应用开销。

第6f节：最大限度地提高吞吐量

在使用任何一种将缓冲区链接起来的数据流方法时，应用可以传递一条名为“数据流指令”的指令。对于诸如音频流等对中断极为敏感的数据流而言，这个数据流指令非常有用，可以确保平稳、流畅的音频或视频信号。需要注意的一点是，应用在使用数据流指令时，将向设备驱动程序发出一些断言消息。应用要确保向设备驱动程序提供的缓冲区永远不会出现“青黄不接”的现象。如果设备驱动程序要处理传入或双向业务，那么，应用会确保始终向其提供一个传入缓冲区；如果设备驱动程序要处理传出或双向业务，那么，应用会确保始终向其提供一个传出缓冲区。第二条断言消息是，如果缓冲区要生成回调，那么，系统定时将确保系统中的中断不会丢失。通过这种便捷的方法，设备驱动程序可以最大限度地提高吞吐量。如果某个设备驱动程序使用了DMA，那么，利用数据流指令，DMA可以实现连续的全速运行。这个特性对于音视频应用尤为重要，它可以消除音频信号中的卡啦声和朴朴声，也可以避免视频信号瞬时中断。任何将缓冲区链接起来的数据流方法都可以结合采用数据流指令，帮助设备驱动程序实现全速运行，应用将保证向其提供充足的缓冲区“后援”。

第6g节：循环数据流

循环数据流方法是我们提供的最后一种数据流方法，这种方法正好对应于DMA的自动缓冲功能。在使用循环数据流方法时，应用将向设备驱动程序提供一个被划分成许多子块或子缓冲区的缓冲区。就好像是将一个连续的数据块划分为许多小块，如幻灯片左侧的这些子缓冲区所示。在使用循环数据流方法时，必须满足一定的限制条件。Blackfin处理器仅实现了16位宽的循环缓冲区，也就是说，循环缓冲区的长度不能超过64 Kb。不过，这是一种向设备驱动程序提供数据的非常简单、高效的方法。首先是应用命令设备驱动程序采用循环数据流方法，并向其提供一个缓冲区，然后，设备驱动程序将从最上面的子缓冲区开始，依次处理这个缓冲区，直到最后一个子缓冲区。应用可以选择让设备驱动程序在处理完每一个子缓冲区时，都向自己发出通知；或者只在处理完最后一个子缓冲区时，向自己发出通知；或者根本不向应用发出任何通知。这是一个非常简单的方法，应用只需要向设备驱动程序提供一个连续的数据块，然后将其划分为多个子缓冲区；应用可以根据需要，要求

设备驱动程序向自己发出通知，可以是在处理完各个子缓冲区时，也可以是在处理完整个缓冲区时。

第6h节：选择数据流方法

如何决定应该采用哪种数据流方法？这张幻灯片列出了我们提供的所有数据流方法，第一个就是我们刚刚讨论的循环数据流方法。如果应用需要处理的数据适用于64 Kb连续数据块，并且是流式数据，如典型的音频流，那么，就可以选择采用循环数据流方法。接下来是未实现环回的简单链接。如果应用需要处理的是基于分组的数据流，通常是突发数据流，如以太网端口、UART和USB等，则应该采用直接的简单链接。环回链接，适用于稳定的数据流，通常是视频流或音频流。在使用环回链接时，应用还需要使用数据流指令，以避免出现卡啦声和朴朴声等噪声或信号瞬时中断，同时还能确保向从器件读取数据的驱动程序提供用于保存数据的缓冲区，或者向通过器件发送数据的驱动程序源源不断地提供缓冲区。不难想象，在处理音频流或视频流时，最好的选择就是采用环回链接数据流方法。实现/未实现环回的顺序链接。这种数据流方法通常用于半双工串行器件，例如，我前面提到过的符合I^2C规范的TWI端口就采用了顺序IO。通过顺序链接，应用可以实现按照预先规定的特定顺序执行读、写操作。

下面，我们来讨论应用在使用设备驱动程序时遵循的标准程序次序。回到我们讨论API时使用的示意图，左侧是应用，右侧是设备驱动程序。在所有初始化都执行完毕后，应用首先要打开设备驱动程序。在这个“adi_dev_Open()”函数中，应用阐明了需要使用哪个器件，也就是说，如果系统中配置了三、四个器件，那么，需要使用其中的哪一个。设备驱动程序应该在哪个方向上打开这个器件，传入、传出还是双向。在这个过程中，应用和驱动程序还要交换句柄，所以，应用向设备驱动程序提供了一个客户端句柄。设备驱动程序在回调应用时，在向应用发出事件通知时，就需要将这个客户端句柄传递回应用。同样地，设备驱动程序也是这个打开函数的一部分，它将向应用返回一个设备句柄。之后，在对该器件进行API调用时，应用将向设备驱动程序发出该设备句柄。

第7章：程序次序

第7a节：链接方法

驱动程序在打开后，通常已配置完毕。一个必须阐明的强制性配置项是我们刚才讨论过的数据流方法，包括简单链接、环回链接、顺序链接等等。任何其他参数，默认参数或配置参数通常都会在器件打开后立即完成配置。稍后，我将举例演示如何为UART配置特定的波特率、数据位、停止位等等参数。配置完毕后，最好是向设备驱动程序提供用于处理的缓冲区，特别是如果设备驱动程序的任务是处理传入业务或双向业务的话。这样，就能确保在发起数据流后，驱动程序可以将数据保存到适当位置。应用可以通过 “adi_dev_Read()”函数、“adi_dev_Write()”函数或

“adi_dev_SequentialIO()”函数，向驱动程序提供缓冲区。此时，还不会发生任何数据传输；截至目前，我们只是向设备驱动程序提供了用于处理的缓冲区。在向设备驱动程序提供缓冲区时，应用也可以有所选择；在有些情况下，应用不一定要在发起数据流之前向其提供缓冲区，但是，在大多数情况下，需要提前向其提供缓冲区，如本例。在提供了缓冲区之后，应用就可以发起数据流了。发起数据流之后，设备驱动程序将开始移动数据，如果它的任务是处理传入或双向业务，那么，它将开始向缓冲区填充其从器件读取的数据。如果，它的任务是处理传出或双向业务，那么，它将开始通过器件发送数据。总之，它将开始处理上一个步骤提供的任何数据。

在缓冲区处理完毕后，如果该缓冲区被标记为生成回调，那么，设备驱动程序将通知应用；它将利用应用的回调函数，通知应用该缓冲区已处理完毕。通常，应用会继续告知设备驱动程序，还有一些缓冲区需要处理，然后，它将调用“adi_dev_Read()”函数、“adi_dev_Write()”函数或“adi_dev_SequentialIO()”函数。同样地，应用可以随时调用这些函数，但采用环回链接的情况除外。显然，在使用环回链接时，应用不能随时向设备驱动程序提供其他缓冲区。这些程序次序是应用在使用设备驱动程序时应当遵循的基本顺序。打开设备驱动程序，配置参数，向其提供缓冲区，发起数据流，当设备驱动程序处理完毕这些缓冲区后，将向应用发出通知，然后，应用将选择向其提供其他缓冲区。下面这个过程将周而复始、不断执行。

第8章：UART举例

第8a节：概述

现在，我们通过具体的例子来演示刚才讨论的过程。在本例中，我将以UART为例。这是一个非常简单的演示程序。我们将使用BF537 EZ-Kit评估板，我已经将这个EZ-Kit评估板上的串行端口连接至这台PC的串行端口。现在，我要在Windows系统下打开HyperTerminal，我们的任务是将其运行参数配置为57,600波特率、8数据位、1停止位、无奇偶。其实，在这个例子中，我们要做的就是发出这些字符并让其返回。当我们在HyperTerminal中键入一个字符后，它会将这个字符传送至EZ-Kit评估板，然后，设备驱动程序将通过一个缓冲区接收到这个字符，并将其传递给应用。接下来，应用会通过UART，将该字符传递回PC；此时，显示屏上显示这个返回的字符。这是一个非常简单的例子，但是完整地演示了应用使用设备驱动程序的所有步骤。在本例中，我们将采用简单链接数据流方法，因此，大家可以了解简单链接数据流方法的工作方式，我们还要使用回调，让UART驱动程序在处理完这些缓冲区后，通知应用。

第8b节：UART程序次序

我们再回顾一遍程序次序，接下来，我们将一步一步执行这些步骤。首先，应用要打开UART驱动程序，并要求UART驱动程序为处理双向数据流而打开。换句话说，UART驱动程序要处理传入业务和传出业务。然后，我们要配置UART驱动程序的运行参数：采用简单链接数据流方法、波特率为57,600、1停止位、8数据位等等。接下来，我们要向UART驱动程序提供一些用于填充的缓冲区。我们不会立即发送任何数据，而是要通过“adi_dev_Read()”函数，向UART驱动程序提供一些缓冲区，以保存其从HyperTerminal接收到的数据。之后，我们将发起数据流。数据流一旦生成，我们就要马上切换回HyperTerminal。这时，我们要输入一个字符；UART驱动程序将收到这个字符，并将之发送给应用。在我们键入字符时，驱动程序将回调应用，告诉应用“我刚刚收到了一些数据。”然后，应用将通过“adi_dev_Write()”函数，要求驱动程序再将该字符返回给HyperTerminal。当这个缓冲区发送完毕后，驱动程序会再一次通知应用，“我已经将该缓冲区发送出去了。”接着，应用将接收这个缓冲区并再次将其放入读通道。实际上，这个过程将循环往复、不断执行。当应用向UART驱动程序提供用于填充的缓冲区后，驱动程序将告知应用它在何时处理了这个缓冲区，然后，应用将回复UART驱动程序“好，再将这个缓冲区发送回PC。”此时，UART驱动程序将回复道“好的，我已经发送了这个缓冲区。”再接着，应用重新将该缓冲区放入读通道。接下来的演示程序就实现了这个循环。

第8c节：构建/运行UART举例

首先打开VisualDSP。好了，我们使用的是VisualDSP 4.0版本，我已经在这台PC上安装了2005年12月发布的更新版。我把这个窗口向这边拖动一点，以便大家看清楚我们有哪些文件。这个例子中只有一个文件，就是这个小文件——“uartexample.c”，其余的设置都由VisualDSP的默认设置自动完成。现在，我们来到了这个函数文件的顶端，我将顺次讲解这个文件的所有内容。这里，有三个包含文件，我在课程开始时曾经介绍过这些文件。这里有“services.h”文件，其中包含了所有的系统服务程序。然后是“adi_dev.h” 文件，也就是设备管理器，其中包含了所有关于设备驱动程序的一般信息，如API、通用指令等等。再下来就是UART设备驱动程序自身，以及针对UART设备驱动程序的特定信息，如关于波特率和停止位的指令等等，所有UART设备驱动程序独具的信息。

现在，向下滚动一点。我刚才说过，在本例中，我们要采用简单链接数据流方法，我们需要使用两个缓冲区，因此，我在这个宏命令中定义了缓冲区的数量。这第一行，我亮显了这一行代码，这个数据就是实际保存将要接收到的字符的存储位置。记得吗？我们之前曾讨论过，缓冲区及其指向的数据可以位于不同的存储位置。如果应用需要处理大量的数据，如视频应用，那么，就需要将数据保存在SDRAM中，而将缓冲区放在其他的存储空间。例如在双核处理器中，可以将缓冲区保存在L2内存中，以便快速调用，而将数据保存在SDRAM中。这样，实际上就将缓冲区与数据分别放置在不同的位置进行保存和处理。

下面这一行是缓冲区，其中的“ADI_DEV_1D_BUFFER”字段，就是我们先前在介绍数据结构中的不同条目时提到的缓冲区，包括指向数据的指针、元素数量、元素宽度等等参数。这就是数据结构。再下来是句柄，这是UART设备驱动程序提供给应用的设备句柄，也就是说，当应用打开UART设备驱动程序时，会将其提供的设备句柄保存到这个存储位置中。之后，应用每一次与UART设备驱动程序通信时都要利用这个值来标示其身份。

再向下滚动。本例中有三个函数，一个是我马上要演示的主函数，另外两个分别是回调函数，即设备驱动程序用于向应用发出事件通知的函数；和负责初始化系统服务程序的函数。在本例中，我们不会实际演示系统服务程序的初始化过程，如需了解关于系统服务程序的更多信息，请复习BOLD培训课程中的系统服务程序单元。虽然我们会迅速地执行这个步骤，不过如需了解详细信息，请复习系统服务程序课程单元，以了解其工作原理。总之，这个过程就是将系统服务程序初始化，以便设备驱动程序使用。

接下来，我们要讨论主程序。在主程序中，首先就是这个UART配置表。应用在打开一个器件时，通常必须立刻为其配置一些参数。对于UART设备驱动程序，我们需要配置其运行波特率、需要采用哪种数据流方法、数据位的数量、停止位的数量等等。为便利起见，我们将这些参数都包含在一个表中，我们既可以单独将这些指令一个一个发送给设备驱动程序，也可以利用“adi_dev_Control()”函数，通过一个表，将所有运行参数发送给设备驱动程序。为了让本例简单明了，我将所有需要发送给驱动程序的指令都放到了这个简单的小表中。下面，我将逐一解释这些指令。第一个，是设置数据流方法，我们将采用先前讨论过的简单链接数据流方法。然后，将数据位设置为8，也就是说UART驱动程序需要处理的数据将与8数据位相关。我们不需要使用奇偶，因此，该参数值为“假”（FALSE）。我们设置了1个停止位，并将波特率设置为57,600。最后一个条目的作用是告知驱动程序，这个配置表结束了。

现在运行这段代码，好了，接下来我们要初始化系统服务程序。之前我们已经讨论了在初始化时必须遵循的特定顺序。在本例中，我们将迅速执行这个过程。我们要设置所有这些参数值。这个过程其实就是为EBIU服务程序初始化SDRAM存储器。然后是设置电源参数，我们将指定处理器的实际运行电压和频率等参数。通过这种方式有效地管理电源，电源管理服务程序就能知道在各种不同的电压电平下，应该以什么频率运行等等。

好，运行这个函数。现在，我们要初始化中断控制服务程序，回想一下前面介绍的初始化顺序，第一个项目就是中断管理器。所以，我们要执行这个步骤。接下来，我们要初始化SDRAM，我们要调用“adi_ebiu_Init()”函数，并以前面提到的配置表为参数。如需了解关于EBIU初始化、电源管理服务程序初始化等的详细信息，请复习BOLD培训课程中的系统服务程序单元。现在，执行这个函数，接着，初始化电源管理服务程序。在本例中，我们只需要使用这三种服务程序。我们不需要使用

DMA以及任何其他服务程序，因此，可以不必理会初始化顺序列表中的其他项目。这就是我们要初始化的三个服务程序。

现在，返回我们的主程序。请记住，在完成初始化系统服务程序后，还要初始化设备管理器，因此，我们现在就要初始化设备管理器。我们要告诉设备管理器，在本例中，我们需要使用一个设备驱动程序。为了节约存储空间，在嵌入式系统中，存储空间是极为珍贵的资源，我们没有采用动态分配方案或预定义固定数量的设备驱动程序，而是允许应用阐明将会同时调用多少个设备驱动程序。然后，应用将向设备管理器提供足以同时运行这些数量的设备驱动程序的存储空间。设备管理器将利用初始化函数完成这个设置。现在，我要向设备管理器提供其用于管理这一个UART驱动程序所需的存储空间。在此之前，需要指出的是，对于每一个API调用——我们先回到上面这个API调用——调用结果将保存在这个被称为“Result（结果）”的变量中。之前，我们讨论过返回码，系统服务程序和设备驱动程序中的每一个API函数都会返回一个结果。如果这个返回值为零，则表示一切顺利。如果为非零值，则表示发生了某种错误或者反馈了某个信息。通常，应用要测试返回值是否为零，如果是零，则表示万事大吉，如果是非零值，那么，应用就必须采取某种措施，查明情况。在本例中，但愿所有的API调用返回值均为零。现在鼠标光标正位于结果变量上，确实是零，我们顺利地完成了初始化设备管理器。

设备管理器初始化完成后，下一步就是打开UART驱动程序，这也是我们刚才讨论程序次序时列出的第一个步骤。现在，打开UART驱动程序；上面这些注释阐明了我们通过“adi_dev_Open()”函数，为UART设置的参数。我们要将这个句柄传递给设备管理器，作为该设备管理器的标识。然后，是需要打开的设备驱动程序的入口点。这个参数表明该入口点是UART入口点。每一个设备驱动程序都具备不同的入口点，也就是说，UART驱动程序有UART入口点，PPI驱动程序有PPI入口点，而AD1836音频编解码器也有自己的入口点。通过入口点，系统可以辨认出应用到底在调用哪个设备驱动程序。

再下来，就是器件编号，在本例中，我们将打开0号器件，也就是系统中配置的第一个UART。如果我们的系统配置了多个UART，那么，我们可以将这个参数设置为0、1、2或者任何其他需要打开的器件的器件编号。在设备驱动程序和系统服务程序中，所有索引号均以0为起始编号，因此，0号UART实际就是第一个UART。下一个参数是客户端句柄。在讨论“adi_dev_Open()”函数时，我们曾介绍过交换句柄，这就是客户端句柄，每一次设备驱动程序回调应用时，都要将这个客户端句柄传回应用。这个参数值对设备驱动程序而言没有任何意义，恐怕只有应用理解它的含义，尤其是当应用同时调用多个设备驱动程序时，它将为这些设备驱动程序提供不同的客户端句柄，以便独一无二地识别出是哪个驱动程序在回调自己。在本例中，客户端句柄为0x12345678，因此，回调函数会将这个值传回应用。

接下来，是UART设备句柄的地址，设备驱动程序会将我们将要打开的UART设备驱动程序的设备句柄保存到这个存储位置。此后，应用每一次调用该UART设备驱动程

序，都要通过这个UART设备句柄来独一无二地标识出该器件。这个器件将用于传输双向数据流，也就是说，它既要读取数据，又要发送数据。下面这个参数为空（NULL），表示本例不需要使用DMA，我们今天使用的UART驱动程序不需要执行DMA，因此，无需向DMA服务程序提供句柄或诸如此类的参数。再下来是提供给回调服务程序的句柄，应该说是延迟回调服务程序。回想一下我们在系统服务程序课程中介绍的，回调分两种，“实时”回调，即在硬件中断时执行的回调；和“延迟”回调，即将回调延迟至较低优先级。为了使本例简单明了，我们将这个参数设置为空，表示将采用实时回调。如需了解关于回调的工作原理以及实时回调与延迟回调之间的具体区别的更多信息，请复习系统服务程序单元。“adi_dev_Open()”函数中的最后一个参数是回调函数的地址。每一次，当设备驱动程序需要通知应用发生了异步事件时，都要调用这个函数。好了，现在执行这个函数，看看其运行结果，是零，也就是说，我们成功地打开了这个器件。

接下来，我们要配置UART。还记得我们先前讨论的位于主程序顶部的配置参数表码？这个配置表中的参数包括：简单链接数据流方法、8数据位、无奇偶、1停止位、57,600波特率。现在，我们要利用“adi_dev_Control()”函数，配置设备驱动程序。执行这个函数。结果显示为零，表示任务顺利完成。这是我们先前讨论的UART设备句柄，在“adi_dev_Open()”函数执行完毕后，应用每一次对UART执行API函数调用都要通过这个句柄来标识其身份。

现在，我们要创建缓冲区。请回想一下，我们先前介绍一维缓冲区时使用的幻灯片，这里列出了那张幻灯片中介绍的所有字段。包括数据字段、元素数量、元素宽度、回调参数、指向下一个缓冲区和补充信息等参数。现在，我们要创建两个这样的缓冲区，以便向这些缓冲区填充数据。我们要将数据字段指向我们已经分配给该缓冲区的数据；也就是驱动程序用于保存其接收到的数据的存储位置，或者用于发送数据的存储位置。在本例中，需要处理的每段数据只包含一个元素，每个元素仅为一字节宽。也就是说，我们将缓冲区映射至一个字符，每个缓冲区仅可容纳一个字符，一个8位字符。下一个是回调参数，我们希望驱动程序在处理完每一个缓冲区后，立即向应用发出回调，因此，这个字段的值为“空”。在本例中，驱动程序通过回调向应用传回的值是缓冲区的地址。也就是说，当驱动程序调用回调函数时，将以刚刚处理完毕的那个缓冲区的地址为参数。这样，我们就能轻松自如地将其映射回来。接着，我们将链接中的下一个缓冲区指向这个pNext值，从而创建了一个缓冲区链接，在本例中，这个链接只包含两个缓冲区。这个值指向链接中的下一个缓冲区。本例其实并不需要使用“补充信息”参数，这个参数在这里只是占个位置，以便向设备驱动程序提供一些我们目前没想到的信息。在本例中，我将利用这个参数来标识缓冲区是传入缓冲区还是传出缓冲区。为了演示这个参数的作用，我将为其设置一个值。在这个字段中，0值表示其为传入缓冲区，而1值则表示其为传出缓冲区。现在，我们为UART驱动程序创建了用于填充数据的缓冲区，并将补充信息参数设置为0。一般情况下不会使用这个参数，此处仅以为例。

接下来，我们要创建缓冲区链接，并将链接中的最后一个缓冲区指向空，从而创建一个空终止的缓冲区链接，即，链接中的最后一个缓冲区指向空值。现在，执行这段代码。好了，缓冲区链接已创建完毕，这两个缓冲区分别指向存储器中需要填充的一个字符。然后，应用将通知设备驱动程序，需要处理这些缓冲区。应用将通过“adi_dev_Read()”函数，将这些缓冲区提供给设备驱动程序。好，执行这段代码，顺利完成。

在向设备驱动程序提供了用于保存其接收到的数据的缓冲区后，接下来，我们要发起数据流。应用将通过“adi_dev_Control()”函数，发起数据流。在此之前，我们先介绍下面的回调函数。设备驱动程序在处理完毕一个缓冲区后，将调用这个回调函数。下面，我们来看看这个回调函数究竟有什么作用。向上滚动一点。这就是回调函数，其中有三个参数。第一个是客户端句柄，记得吗，这个值就是“adi_dev_Open()”函数中的“0x12345678”。请记住，设备驱动程序每一次向应用发出事件通知时，都要向其传回这个客户端句柄。这是回调函数中的第一个参数。不论何时设备驱动程序调用这个回调函数，其客户端句柄值均为“0x12345678”。

第二个参数是发生的事件。在本例中，这个事件就是“缓冲区处理完毕事件”。不论何时发生任何事件，设备驱动程序都要调用这个回调参数，而在本例中，我们真正关心的只有这个“ADI_DEV_EVENT_BUFFER_PROCESSED（缓冲区处理完毕）”事件。应用在收到这个事件通知后，将结束该事件。最后一个参数实际取决于发生的事件，这个参数将用于表明一些关于该事件的信息。在技术文档中，详细介绍了这个参数。就“缓冲区处理完毕事件”而言，这个pArg值就是前面设置的这个回调参数值。也许大家还记得，我们刚才已将这个回调参数值设置为缓冲区的地址。因此，当设备驱动程序处理完毕一个缓冲区后，将调用回调函数。它将向应用返回其客户端句柄“0x12345678”，告知应用发生的事件为“ADI_DEV_EVENT_BUFFER_PROCESSED”，表明自己已经处理完毕该缓冲区；最后一个参数则是该缓冲区的地址，即，指向该缓冲区。

下面我们来看看这个回调函数的代码。首先，由于我知道这个pArg值将指向缓冲区，并且这个函数包含了一段“void*”代码，因此，应用不必执行数据转型，而是直接声明一个属于该数据类型的临时变量。这样，当设备驱动程序调用该回调函数时，就会向应用返回该缓冲区的地址。然后，应用要确认该事件是不是“缓冲区处理完毕事件”。接下来是“补充信息”字段。前面我已经说过，我们将传入缓冲区的这个字段值设置为0，将传出缓冲区的这个字段值设置为1。在收到一个传入缓冲区或者说收到一个缓冲区时，应用需要确认或者说测试该缓冲区是否为传入缓冲区。如果确实是传入缓冲区，那么，应用就会更改这个标签，将其变为传出缓冲区。当然，通常不需要使用这个“补充信息”字段，这里仅为演示读缓冲区和写缓冲区。接下来，应用要确认该缓冲区是不是空终止，然后，将缓冲区发送至UART驱动程序。换句话说，当应用接收到一个字符，即一个缓冲区后，也就是当UART驱动程序告知应用它接收到了HyperTerminal发出的一些数据后，应用要做的就是将这

些数据发送回去，即，将其放入写通道，并告知驱动程序希望将该数据发送回HyperTerminal。当驱动程序完成这个任务后，它将再次回调应用，报告它已经处理完毕该写缓冲区。我们再来看看“补充信息”字段，现在，这个字段表明，当前缓冲区为传出缓冲区。接下来，我们要再次改变这个参数，表示希望将这个缓冲区放入读通道，也就是置于读缓冲区排队中。我将这个值设置为0，并确认该缓冲区链接为空终止。然后，调用“adi_dev_Read()”函数。好了，稍后我们将返回这个函数的开头，逐步演示整个过程。

接下来，我要在第一个存储位置设置一个断点，这样，当设备驱动程序的回调应用时，程序将暂停在这个地方。现在，回到上面的主程序，我们要发起数据流，也就是说，告诉UART驱动程序，“好了，开始读取数据。”接下来，应用只需要呆在这个循环里，静静等待接收字符。单击“运行”图标，屏幕下方的窗口显示应用正在运行。实际上，应用只是呆在这个“while”循环中，等待接收字符。我们向下滚动一点，这里显示了一些信息。我们是在VisualDSP中运行，这里是Blackfin处理器的内存，应用正在运行，所以一切都是灰色显示。这里是数据缓冲区，当我们在HyperTerminal中键入字符时，在这个字段将显示键入的字符。现在，我要打开HyperTerminal，好了，我已经按照设备驱动程序的设置对其完成了相应的设置：57,600波特率、8数据位、无奇偶、1停止位。现在，我要键入字母“A”。在我键入字母“A”时，HyperTerminal会将该字符发送至EZ-Kit评估板，设备驱动程序将收到该字符并进行相应的处理，即将其保存到缓冲区中，并调用回调函数，通知应用，从而达到我们设置断点的位置。因此，当我键入字母“A”后，VisualDSP会在断点位置停下来。键入字母“A”，返回VisualDSP，在屏幕的下方可以看出，应用已经暂停，我们正好停在回调函数上。

此时，驱动程序正要向应用发出事件通知，因此，我们现在只单步执行这个回调函数，向应用发出事件通知。这个事件就是数据缓冲区已经处理完毕。接下来，我们要检查并确认，这个缓冲区是一个传入缓冲区，或者说测试该缓冲区是不是传入缓冲区。它应该是传入缓冲区。现在，在右上方，显示了我刚才键入的字母“A”。这就是我们刚才在上面定义缓冲区时为其分配的数据，也是驱动程序需要处理的数据，即用于保存数据的存储位置。确实是一个传入缓冲区。现在，我要玩个小花样，看看会发生什么情况。我们将补充信息字段设置为“1”，也就是将缓冲区设置为传出缓冲区。接下来，我们要确认该缓冲区是否为空终止，也就是说，这个链接中只有一个缓冲区，我们要发送的不是整个缓冲区链接。然后，应用将调用“adi_dev_Write()”函数。通过“adi_dev_Write()”函数，应用将告知UART驱动程序，还有另一个缓冲区需要处理，希望它将这个缓冲区发送出去。如果我单击“运行”，那么，应用会将这个缓冲区提供给设备驱动程序，然后，设备驱动程序会将该缓冲区传回HyperTerminal，并再次回调应用，告知应用已将该缓冲区发送。现在，我们再次打开HyperTerminal，但是屏幕中还没有任何返回的数据。如果我单击“运行”，驱动程序将传出该数据，应用将立即回到断点，因为驱动程序在发送完缓冲区后要立即回调应用。好了，单击“运行”，不出所料，我们回到了断点。现在，再打开HyperTerminal，返回的字母“A”赫然在此。此时，驱动程序

正要回调应用。如果我们单步执行这个回调函数，通知应用发生了缓冲区处理完毕事件。这一次处理的缓冲区是一个传出缓冲区，补充信息字段已经被设置为“1”。现在，我们要再次将这个字段设置为“0”，因为，接下来应用要将该缓冲区分配给UART驱动程序用于填充其他数据。将这个链接设置为空终止，也就是说，这个链接中只有一个缓冲区。接着，应用要调用“adi_dev_Read()”函数。如果继续运行，驱动程序将通知应用它收到了这个缓冲区，并将接收到的数据填充到这个缓冲区中，然后，通知应用已经将缓冲区处理完毕。如果我单击“运行”，那么，应用将开始运行，驱动程序将等待接收另一段数据，并在收到后回调应用。此时，如果我打开HyperTerminal，并键入字母“B”，应用将再一次到达断点。好了，执行完毕。我们回到了回调函数，现在，右上方显示的数据应该为字母“B”。接下来，我们要再一次执行这个循环，将缓冲区设置为传出缓冲区，并将缓冲区链接设置为空终止，然后将其发送回HyperTerminal。再次调用“adi_dev_Write()”函数，HyperTerminal的屏幕中将显示返回的字母“B”。确然在此。返回VisualDSP，现在，驱动程序将告知应用，其已经将该缓冲区发出。接下来，我们要执行这个函数，将缓冲区标记为传入缓冲区，并确认链接中只有这一个缓冲区，然后，通过“adi_dev_Read()”函数，将其分配给驱动程序。

现在，我取消了断点，这样，应用在运行过程中不会产生任何停顿。单击“运行”。现在返回HyperTerminal，大家可以看到，我键入的字母立即返回了屏幕。这是一个非常简单的例子，演示了使用设备驱动程序的基本步骤，如何打开设备驱动程序、如何配置、如何向其提供传入缓冲区或者传出缓冲区、设备驱动程序在处理完缓冲区后如何通知应用等。这确实是一个简单的例子。另外还需要注意的是，除了初始化系统服务程序，设备驱动程序将自动完成所有设置。UART中断自动完成挂接。虽然我们在本例中未使用DMA，但是如果我们使用了DMA的话，设备驱动程序将妥善地管理好DMA。在本例中，UART设备驱动程序实际上调用了电源管理服务程序，来确定系统时钟频率，从而计算出支持UART以57,600波特率运行的分压比。所有这些处理都是由驱动程序自动完成的，并且全都借助了系统服务程序的力量。通过让服务程序和驱动程序实现协同合作，我们为应用提供了一个功能强大、易于使用的环境，帮助开发人员以前所未有的更快速度，迅速完成应用编程，将产品推向市场。开发人员不再需要为每一个器件单独编写设备驱动程序，客户也不再需要为自己的特有应用开发专门的器件。借助这些设备驱动程序，客户可以大幅缩短产品上市时间。

第9章：结束语

第9a节：更多信息

好了，我们回到演示文稿。前面的课程已经表明，我们的系统服务程序和设备驱动程序共同实现了一个非常稳定的软件基础，能够帮助客户快速完成产品开发。重复工作减少了，应用或用户不再需要完全从新创建所有的程序，它们可以充分利用我们提供的系统服务程序和设备驱动程序。我们提供了高度模块化的软件，所有驱动

程序都能同心协力、并肩作战。不但各个软件组件团结一致，硬件资源也得到了有效管理。此外，这个环境还实现了杰出的可移植性，开发人员可以轻松地将在BF537 EZ-Kit评估板上运行的程序导入BF533 EZ-Kit评估板或BF561双核EZ-Kit评估板。完全无需对应用进行任何修改，因为所有设备驱动程序均采用完全相同的API，系统服务程序以完全相同的方式运行，以完全相同的方式实现所有功能。客户可以高效率地将其在现有处理器上运行的应用导入未来问世的更先进、更强健、更快速的处理器中，极其迅速地将应用移植到我们最新发布的处理器中。

如需了解更多信息，请参阅模拟器件公司网站技术资料库页面提供的《设备驱动程序和系统服务程序手册》（Device Driver and System Services Manual），地址如下。2005年9月，我们发布了用户手册附录，其中提供了关于最新版本服务程序和设备驱动程序的最新功能的信息。我们还将陆续推出其他技术文档。我们将在VisualDSP安装目录的Blackfin/doc目录下，提供各种最新信息，包括最新发布的技术文档和驱动程序等。

如有任何特殊疑问，请单击下面的“ask a question（我有疑问）”按钮，或者发送电子邮件至processorsupport@analog.com。谢谢参加设备驱动程序BOLD培训课程，希望能对大家有所帮助！

再次感谢！