Blackfin Online Learning & Development

Presentation Title: Blackfin® Device Drivers

Presenter Name: David Lannigan

Chapter 1: Introduction

Subchapter 1a: Overview

Hi my name is David Lannigan. I work in the DSP and Systems group at Analog Devices. Today I'm going to be talking about the device driver model for the Blackfin family of processors.

Prior to viewing this module the user should have a good understanding of the Blackfin architecture, and have also viewed the system services module of the BOLD training.

I'm going to start by going through some background information, some common conventions, terminology that we use with the device drivers. I'm going to talk about the device driver API, show the functions that exist in the API. We'll talk about buffers, which are how the application provides data that the device drivers go and process. We'll talk about the various dataflow methods that the device drivers use, which describe really how the device drivers actually process the data within those buffers. And we'll go through a very simple example with our UART device driver, again with the VisualDSP toolset in the December 2005 update, just a very simple talk through type example that will illustrate many of the concepts of the device drivers.

Chapter 2: Device Driver Model

Subchapter 2a: Overview

The device driver model uses a standardized API, so all Blackfin processors use the same API regardless of the processor, regardless of the driver. Developers only have to learn the API once. All drivers operate the exact same way. It's extensible, there are definitions that each driver can go and add their own commands, their own events, their own return codes, the goal is to cover the vast majority of device drivers. Today we support such diverse devices as SPORTS, Serial Ports, SPI, PPI, TWI and advanced devices such as Ethernet and USB. There will be exceptions though, the goal is to cover the vast majority but it's anticipated there will be devices that come up some time where the model doesn't fit. But like I've said we've created a very broad and diverse set of device drivers, and the model does fit them quite nicely.

The device driver model is built upon the system services, that allows us to have a very stable software base and we don't have to reinvent such things as DMA and include that in each individual device driver. The drivers themselves leverage the system services. For example a PPI driver will make use of the DMA service to go and use DMA to move data. This allows us to have a very modular software environment. It allows us to have better compatibility so that the drivers inter-operate with each other in a very simple and clean manner. Easier integration; there's very few issues in terms of having multiple drivers working concurrently in the same application. And it's portable, the driver for the BF533 works identically as it does on some of our other Blackfin processors such as the BF537 even the dual core BF561.

The basic architecture for an application is shown in this diagram here, where the application is up at the top. Sometimes as an RTOS such as a VDK or a third party operating system, sometimes not, often the applications run as we call just "stand alone." Below that are the device drivers. And as you can see on the diagram the device drivers sit on top of the system services, and the drivers themselves make calls into the system services, such as interrupts. Whenever a driver wants to control an interrupt, it makes a call to the interrupt manager. DMA, as I said earlier, makes calls into the DMA service and so forth.

Chapter 2b: Using device driver

To use a device driver in a VisualDSP project is quite simple. In the application's source file you need to include three lines. The first line that I've shown here on the slide, shows "#include <services/services.h>". That include file pulls in all the system services so DMA, Interrupts, Port Control, all of those services are brought in through that include file. Next up is the device manager's include file, it's called "adi_dev.h". That pulls in all the generic device driver information such as the API itself, return codes, event codes, and all common device driver information. The third file that would need inclusion is the device driver itself; the include file for the specific device driver. If we were building an application as I'll show in the example later on using the UART, we would include the UART's .h file and that pulls in all the specific information regarding the UART device driver.

For off-chip device drivers, for things such as external devices such as a CODEC or a video encoder, or a video decoder, an off-chip Ethernet controller and what not, to pull in that device driver the user should include the device driver's .C file or it's "name ".C file or collection of .C files depending upon the driver and a list of sources. We don't provide a library with all our external devices because the library is actually quite big, and we don't know ahead of time what devices each application is going to use. The application, in order to save memory space, just brings in the active drivers that are needed for that particular application.

In the linker files folder the application needs to link with the proper system services library that's the libssl library. And it also needs to pull in the proper on-chip device driver library. All those libraries are prefixed by the letters "libdrv". I'll go through how to decide which library to pull in, in just one minute. Another option that is strongly recommended is to the Code Elimination Option in the Linker Property Page. This significantly reduces code size and what it does is it throws away all the code that's not used in the particular application. If we're using a device driver and we're only using functions A, B, and C of the device driver it'll throw out the functions D, E, and F if they're not used in the application. This significantly saves code space which is important in embedded applications.

Chapter 2c: On-Chip Driver Library

The on-chip drivers come in two flavors, we have a Debug version of the library and we have a Release version of the library. We strongly recommend that when users are starting off they use the Debug version of the library. All the compiler optimizations are turned off, symbolic information is included, which allows you to step into the source code and see exactly what happens on a line by line basis. We do a lot of parameter checking in the Debug version of the library so that if parameters are passed and they contain incorrect values we do our best to make sure that we check for those and return the appropriate error codes should they occur.

As I said these should be the libraries that users start with; examples, demos typically use the Debug version for illustrative purposes.

Once the user is satisfied that the driver is working properly, they can switch to the Release version of the on-chip library. In the Release version all the compiler optimizations are turned on, so performance is improved. However we don't put in symbolic information so it makes it very difficult to step into the sources and see what is executing in any point and time. We do much fewer, if any, parameter checks because we assume at that point the user is employing the Release version of the Library, they know what they're doing so we actually relax the vast majority of our parameter checking. These should be the Libraries that users end with. Again, start out with the Debug Version of the Library, but when you're ready to release your product switch to the Release version of the Library.

How do I decide which Library to pull in? Well we use the same naming convention as is the convention in VisualDSP. The device drivers libraries for all the on-chip peripherals are prefixed with LIBDRV which stands for the library for the device driver. The next three characters describe the processor variant that that library pertains to. For example if the next three characters are 532, that pertains to the BF531, BF532, and BF533 Blackfin processors. The prefix 534 supports the BF534, BF536 and BF537 processors. And 561 supports the dual core BF561 processor. The next three characters are omitted today, when we have a driver that requires some support from the operating system. You'll see that the next three characters define the operating system that's appropriate for that particular driver. In this case here, with all the device drivers that we've written today, we don't have any specific versions of the Library that require anything from the RTOS. Today those next three characters that show in the slide, the "bbb", are blank. The drivers run in both stand alone and in the VDK version today.

The next two characters define any special conditions for the library. The combinations, or the letters used in those two characters are either "d" and or "y". If there's just a "d" there that means debug version of the library. If there's no "d" there then it's a release version of the library. The letter "y" stands for work-arounds for silicon anomalies. If the letter "y" is in the library's name then that means we have the work-arounds for all the various silicon revisions built into that library. And a combination "dy" means it's a debug version, plus all the work-arounds are in there. If it's blank it means it's not debug, which means it's the release version and we don't have any work-arounds.

To find all the information on the device drivers, the include files for all of our drivers are kept in the directory called "Blackfin\include\drivers", and on this slide here I've shown the default location when you install VisualDSP this is where we'll find all the include files for the device drivers. The sources themselves provide source code for all the device drivers that we have today, those are located in "Blackfin\lib\src\drivers". The libraries themselves are located with the rest of the VisualDSP libraries in the directory "Blackfin\lib".

Examples, we have examples for the BF533 EZ-Kit, the BF537 EZ-Kit, and the BF561 EZ-Kit. And those examples are found in the directory "Blackfin\EZ-KITS" and then you look in the appropriate sub directory for the processor that you're targeting.

We have two pieces, two major pieces of documentation I should say on the device drivers, one is the manual itself, that's located in the Technical Library section of Analog.com. And then we had an addendum that we published in September of 2005 that includes information on some of the newer services and some of the newer capabilities of the device driver. That can be found on Analog's FTP site as shown on the slide here.

Chapter 3: Design Considerations

Subchapter 3a: Memory

No dynamic memory is used in any of our device driver. Any memory that's used by the driver is allocated statically. However the device manager itself needs memory to manage devices. The more device drivers you want to run concurrently, the more memory is needed to be provided to the device manager. physical drivers, the low level drivers for the PPI, SPORT, UART etcetera, they use static memory for all their internal data so there's no dynamic memory allocations that go on in those drivers. In general no dynamic memory allocation in any of our device drivers, all of it is either allocated statically or provided by the user at application time. This allows the application to tell us what memory to use, whether it's internal memory, whether it's in external SDRAM memory, we leave that up to the user to decide. As a result we have no restrictions on any memory placement, be it code or data, they can be wherever the application wants it to be.

Subchapter 3b: Handles

We use Handles to communicate between, or to identify the various components within the driver model itself. When a device driver is opened, the application is passed back a handle to that device driver. Every time the application wants to reference that device driver from that point forward, it needs to pass the device Handle so that the system knows what device driver's being accessed at any point and time. It's really just a unique identifier and really points to the address of a location in memory that we use to keep all the information that we use to manage that particular device driver.   In turn, when a device driver is opened, the application provides us with a handle, this Client handle. The Client Handle can be whatever the client or application wants it to be, it has no significant to the device drivers, it can be anything that is relevant to the client. And we use this handle whenever the device driver communicates back to the application. Let's take the case of an asynchronous event that occurs when the device driver wants to notify the application of that event, say an error occurred or a buffer has been filled or what not, the device driver passes that client handle back to the application so that the application has a means to identify where that notification is coming from.

Subchapter 3c: Result code

Almost every API function returns a result code. There are a couple of exceptions that are very obvious, but by in large every function returns a result. Zero is the value for universal success. If a device driver, or rather any of the device driver API calls return a zero, that means everything was successful in that execution of that call. If it's non-zero that means some type of error has occurred or some type of informative result is being turned back to the application. Each driver has its own set of return codes so they're unique. Each driver's return codes are unique from the other drivers in the system. They return a u32 or an unsigned 32 bit value. This value that's passed back can be examined and depending upon the value it indicates what error or informative event is being returned to the application. For example let's say that we open up a serial port driver and that serial port driver is required to hook into an interrupt, say the error interrupt for a serial port. The device driver opens the interrupt and hooks the interrupt properly and everything happens as it should, a value of zero is returned back to the application. If for example, maybe there's some reason why that interrupt could not be hooked, then the device driver returns back and error code indicating that the interrupt can not be hooked and the application needs to take some action.

When a non zero value is returned back to the application, the easiest way to find out what's wrong is to look in the "services.h" file and you'll see in there a unique set of numbers that each System Service or each device driver can return. And then look in the appropriate .h file for that particular value and that will indicate what either the error code or the informative value means.

Subchapter 3d: Initialization

There's a specific sequence that the application needs to perform in terms of initialization. Because the device drivers are built on top of the System Services, the System Services need to be initialized first and then we initialize what's called the device manager. The device manager provides the API for all the device drivers, so he's kind of in control of all the various device drivers in the system. But because the drivers layer on top of the services, the services should be initialized first followed by the device manager. The specific sequence that should be followed is shown in this slide. Interrupts, you want to initialize the Interrupt manager first, and then we do the EBIU which is our external bus interface unit, Power, Port Control, then our Deferred Callback Service, DMA manager, Flag Service, Timers, and then finally the device manager itself. If any of these services are not used they can simply be omitted from the sequence. This specific sequence ensures that everything's initialized properly so that we don't have any conflicts, any service or device trying to be accessed before something has been initialized.

Subchapter 3e) Termination

Conversely there's a specific termination sequence that the application must follow. Often embedded systems applications never terminate, they run perpetually, but if there is a need to terminate, the stacks should be terminated in a specific sequence, that sequence is shown here.

The sequence is the exact opposite of the initialization sequence, so the device drivers should be terminated first, followed by the termination of the System Services in the exact opposite order that they are initialized. In this case here terminate the device manager itself, and then we go through the Timer, Flags, DMA, Callbacks, Port Control, Power, EBIU, and lastly the Interrupt manager.

Subchapter 3f: RTOS Considerations

Within the device drivers that we have today, there are no device driver dependencies on the RTOS. For example the VDK RTOS from Analog Devices, we do not have a specific device driver Library for the VDK because the device drivers run in both the stand alone mode and with the RTOS. Any of the RTOS interactions, the areas where the services or drivers bump into the RTOS, they're all isolated in the System Services which is why we have different System Service Libraries for stand alone or RTOS versions, but in the device drivers themselves there are no dependencies. One thing to note is that the API is identical regardless of whether we're running in RTOS environment or in a stand alone environment, there's no changes at all so the application doesn't have to change any of it's calls if it decides to move from a stand alone environment to an RTOS based one or vice versa.

Chapter 4: Device Driver API

Subchapter 4a: Overview

The API is quite simple, on this slide here I've shown the application on the left hand side, and the device driver on the right hand side. There are six functions in all in the API, and they're very familiar to anyone that has used device drivers in the past. We basically have the five common ones; open, close, read, write, and control, and then we have another one that we've added called SequentialIO. Let me go through each of these individually and explain a little bit about what they do.

Subchapter 4b: API Functions

The adi_dev_Open() function obviously opens the device driver for use. When the application wants to use a driver the first thing it does is call the adi_dev_Open() function. If the application ever decides that it doesn't need the device driver any more it can call the adi_dev_Close() function. The adi_dev_Close() function shuts down and releases any system services that were used, any hardware that was used, it shuts it all down in an orderly fashion. If a device driver has been opened for inbound dataflow, or for bi-directional dataflow, the adi_dev_Read() function is used to pull data in from the device. In other words to pull data in say, let's take the case of a PPI that is connected up to a camera, as we want to take data in from that camera we would use the adi_dev_Read() function to pull that data in through the PPI device. Conversely we have a write function. The write function is used to send data out through the device driver, so take that same PPI driver again, if we were going to take that video data that we recorded from the camera and play that back to a monitor, we would use the adi_dev_Write() function to send that data out through the PPI and out to the monitor.

We have another function called adi_dev_Sequential IO() and that allows the application to specify a specific sequence of reads and writes. Without Sequential IO, reads happen as fast as reads can happen, writes happens as fast as writes can happen, but there's no synchronization between reads and writes. If a device such as the case of a DSL modem, a DSL modem typically has a faster download speed than upload speed; reads happen at a much quicker rate then writes.

If there's a device that requires a specific sequence of reads and writes in a specified order, the adi_dev_SequentialIO() function provides a mechanism that allows the application to specify some number of reads followed by some number of writes or what not, but in a very specific sequence.

Lastly we have the adi_dev_Control() function which in the old days was kind of like the IOCTL or IOCTL functions in some of the earlier device drivers. This function is used to typically set or sense any type of parameters for the device driver. In the example I'm going to show later on we're going to use the adi_dev_Control() function to specify the baud rate for UART, the number of stop bits and so forth.

That's it. There are only six functions that are used when an application wants to communicate with the device driver. Coming back the other way, if the device driver needs to communicate back to an application, that communication happens asynchronously through the event mechanism and callback mechanism that we have in the System Services. When the device driver needs to notify the application of an event, say that event could be the buffer has been processed, or that an error has occurred on the device or what not, invokes the applications callback function. Right there, that is the entirety of the API between the application and the device driver. Those six functions, and then the applications callback functions so that the device driver can notify the application.

That's it. It's not any more complicated then that. And one thing to note is as I described earlier about the handles, when the application calls the adi_dev_Open() function it's given a handle for that device driver. Any of the other five functions; adi_dev_Close(), adi_dev_Read(), adi_dev_Write(), adi_dev_SequentialIO() or adi_dev_Control(), the application passes that handle as the first parameter into that API function. And that lets the device manager which specific device driver the application is addressing. When the device driver notifies the application of some event, by the callback function, the device driver passes back to the application that client handle value that was passed in. That client handle value is passed in again in the adi_dev_Open() function. That way the device drivers know how to communicate back up to the application.

These same six API functions are the same regardless of which device driver is being used. Be it a UART, be it a PPI, SPI, SPORT, TWI, Ethernet, USB, the API is identical. Even for our off-chip peripherals such as ADCs, DACs, Video Encoders, Video Decoders, again those same six functions are used. The API itself is described in the generic device driver .h file called "adi_dev.h". Each device driver can add extensions. It can add it's own custom commands, so for example if it has it's own IOCTL values or unique parameters that need to be set or sensed. Take the case of a DAC for example; a DAC you may want to set the volume of a specific DAC, each device driver has the means to extend the control function or the commands that can be passed through the control functions I should say, to the device driver. Also each device driver can create new return codes for the application to better convey should something go wrong or any type of informative type functionality through an API call. Each device driver can specify its own return values in addition to the ones that you can find in the generic "adi_dev.h" file.

Also events, the device driver can create any additional events that warrant notification back to the application in the standard generic "adi_dev.h" file we provide events for buffer completion, errors and so forth. But if there's some unique situation that a device driver needs to notify an application, the device driver can actually extend the API to describe additional events. For example an off-chip controller may decide to go into sleep mode, and it needs to notify the application that it's going into sleep mode, it can create an unique event that says such and tell the application whenever that event occurs.

Subchapter 4c: Interaction with System Services

As I said at the beginning of the presentation the device drivers are built on top of the System Services. That makes it very easy and very simple for us to create device drivers, and also gives us a nice way to manage the resources of the Blackfin part itself; for example DMA. The PPI device driver makes calls into the DMA manager whenever it needs to move data through the DMA; opening DMA channels, providing descriptors or what not, rather then have to put that in each device driver that uses DMA we only implement that once in the System Service. We implement it once, we get it working properly, and then all of the device drivers go and benefit. Same thing with Interrupt Manager, Timers, and Deferred Callbacks etc. The PPI driver uses all that functionality but simply makes calls into the appropriate system service to go and manage that functionality. From an application perspective the only thing the application needs to do is initialize the system services. Once the system services are initialized the device drivers make all the calls themselves into the system services. Again take the case of the PPI driver, when the PPI driver needs to use the services of DMA, provided the application has initialized the DMA service, the PPI driver just goes in and accesses the DMA service as it needs to. This makes it very simple for the applications, the applications only have to talk to the device driver API then all the system service activity is taken care of by the device drivers themselves.

Chapter 5: Transferring Data

Subchapter 5a: Buffer Overview

The means to move data through a device driver either out through the device or in from the device, we use buffers to describe that data. We have two types of buffers, inbound buffers, they're filled with data that comes in from the device and outbound buffers that contain data that the application wants to send out through the device. We have various types of buffers; we have one dimensional buffers, which are traditional linear buffers. We have two dimensional buffers that map directly to the 2D DMA capability of a Blackfin architecture, very important when doing video type applications where you want to get macro blocks out of a big frame for example. We have sequential buffers that are used for the Sequential IO functions. Today they're just one dimensional buffers, but linear buffers where we can describe a specific sequence of reads and or writes. And then we have circular buffers, they map directly to the auto-buffer capability of Blackfin. With a circular buffer, we give it one big chunk data and then the device driver continually iterates through that circular buffer.

Subchapter 5b: One-Dimensional Buffer

Let's look at a one dimensional buffer which is the simplest of the various types, this line here shows the fields that are within that one dimensional buffer. The first field in there is the pointer to the data. The data can exist any where in memory, so the buffers themselves can be physically removed from where the data is. This is very useful in that we don't have to move data back and forth, and we don't have to put buffers in the same place as the data, so we can move large amounts of data, say video where the video frame is almost a megabyte of data, we can put the buffers themselves in a different spot then the actual data that's contained within the buffer. There's an element count that says how many elements that pointer to the data, or how big that data is. Then there's the width of each element so that's the width in bytes of each element within that data.

Let's take a case of a piece of data that's 1,024 bytes long. We would typically say the element count is 1,024 and the element width is 1, meaning 1 byte wide. If we wanted 1,024 elements of 16 bit data, again the element count would be a 1,024, but the element width would be 2, because it's 2 bytes wide. Each buffer has a callback parameter. The value for that callback parameter indicates whether or not the device driver needs to notify the application when that buffer has been processed. If the callback parameter is NULL, then no callback is made when that buffer is processed. Let's take the case of an application sending a buffer out through a device. If when it creates the buffer it puts a NULL value in the callback parameter, the device driver will go and send that data out through the device, but will not notify the application that the data has actually been sent out. If the callback parameter is non-NULL, then that indicates to the device driver that the application wants to be notified when the data has been sent out to the device. The device driver will callback the application, letting the application know that that buffer has been sent out. And it actually passes back to the application callback function, the value that was passed into the callback parameter. This gives the application several choices, it can tell the device driver here's some data go and send out, or data to fill, to pull in from the device. You don't need to notify me when it's done. Or another option is here's some data that I want to go and send out through the device, or fill in from the device, and I want to be notified when that buffer has been processed. We give the application the option of being notified or not being notified.

Next in the buffer is a Processed Flag, and that gets filled in by the device driver, it's just a simple flag that says TRUE or FALSE whether or not the device driver has finished processing that particular buffer. If has processed it, there's a Processed Count Field that gets filled in. That Processed Count field indicates the number of bytes that have been processed by the buffer. Typically that's the same as the element count times the element width, but there are instances where that may not be the same, let's take the case of an Ethernet driver. In an Ethernet system there's a maximum packet size for each piece of data that goes out over the wire. But not all packets are that size. Some are much smaller than that. In the case of the Ethernet driver we may be passing a large buffer to the Ethernet driver saying fill this buffer up with data that comes in over the wire, but maybe only a small packet was received, maybe just a short packet, so the process count tells the application how many pieces of data were actually put into the buffer.

The next field we have is called pNext, which points to the next buffer in the chain. And that's an important concept and I'll describe that over the next few slides. We can chain buffers together so that one buffer can point to another buffer to another buffer, so we create a chain or a sequence of buffers for the device driver to process.

Last up we have a field in there called Additional Info, that's kind of a catch all that we put in there. None of our device drivers today actually use that field. In the example that I'm going to show later on we're going to put some data in that field just to show how it's used, for the purpose of illustrating the example. But it's meant as a means so that in some of our future device drivers if there's information that we can't anticipate at this point, being put into a buffer, this gives applications and drivers a means to include additional information that perhaps we haven't thought of yet.

Chapter 6: Dataflow Methods

Subchapter 6a: Overview

There are five basic dataflow methods. And the dataflow methods describe how the device drivers process the data. I alluded to chaining on the previous slide, we can chain both one dimensional and two dimensional buffers and that allows us to create one, two, three, or some combination of buffers. By the way there are no limits as to the number of buffers that can be provided to a device driver at any point at time, how many are queued up, how many the device driver is processing, there's no limit to any of that. You can provide as many buffers as are needed.

Chaining with loopback, I'll describe what that is in a minute, but that allows us to create a chain and then loop back through that chain over and over again. We have Sequential Chaining which is another chain but with a specific sequence as I described about reads and or writes in a specific order. We can loop that back so we can do that over and over again. We have a Circular dataflow method that maps directly to the auto buffer capability of Blackfin, where we give it one contiguous chunk of memory and the device driver just keeps iterating through that same chunk of memory over and over again.

We have these five different dataflow methods, but a device driver doesn't need to support them all, some of them are inappropriate for a device driver. But the device driver has to support at least one, so a couple of examples here, the PPI driver supports one dimensional buffers, two dimensional buffers and circular buffers. The UART driver, that I'll show in a few minutes, supports one dimensional buffers. TWI which is kind of an I^2C compatible type of protocol, supports sequential 1D buffers. For that interface, or protocol I should say, a specific sequence of reads and or writes is appropriate for that particular protocol.

Subchapter 6b: Chaining Method

Let me walk through an illustration of chaining and how that mechanism works. With the chaining method buffers are effectively queued to the device driver, and we keep two different queues, we keep one queue for inbound buffers, so that queue contains a list of buffers that are provided to the device driver by the adi_dev_Read() function. And the device driver is going to fill those buffers with data as data is pulled in from the device. We have a separate queue for outbound buffers. Buffers in the outbound queue contain data that the application wants to send out through the device. In both of these queues buffers are processed in a FIFO type fashion, they're processed in the order that they are received. And they're processed asynchronously. Go back to that case I talked about earlier such as the DSL modem, in there the read buffers may be processed at a faster rate then the write buffers, so the reads and writes happen asynchronously to each other.

Buffers can be provided to a device driver at any time, so they can be provided at application time, at interrupt time, there's no restriction as to when the application can provide us a buffer to process. Buffers can be provided one at a time or in groups, so you may choose to give us buffers individually, one after another, you can chain buffers together so you can say here's ten buffers and pass in a chain of ten buffers all at once. Each buffer can point to data of different sizes. You don't have to have each buffer point to a fixed size buffer, we may have some buffers that are provided that point to pieces of data that are 128 bytes in size, we might have other buffers in the chain that point to buffers or data that's 1,024 bytes in size. You can mix and match different sizes on a particular queue.

Any, all or no buffers can be tagged to generate a callback. Remember a callback is the mechanism that the device driver uses to notify the application that the buffer has been processed. In this slide here, if we start with the buffers moving from left to right, the first buffer would not be notified when it was processed, and the application would not be notified when that first process was processed. The next one with the check mark below it, when the device driver processed that buffer, it would notify that application, "Hey I've just gone and processed this buffer." The next two would be processed automatically but no callbacks sent back to the application. The second to last buffer, when that one was processed the device driver would notify the application that that buffer had indeed been processed. You can mix and match, any buffer can be tagged, all buffers can be tagged, often some applications just tag the last buffer in the chain so that they get notified when the last one has been processed so then they know that everything prior to that last one has been processed by the driver.

Once processed, the buffer is not used again unless it's resubmitted. Think of that as just a chain, when the driver gets to the end it stops. The next slide I'm going to show with loopback and I'll show a means to get around that. But with the basic chaining method, buffers are provided to the driver, driver processes them, and then after they're processed they're basically back in the domain of the application.

Subchapter 6c: Chaining with Loopback

Let me talk about chaining with loopback because that's a very efficient method of providing drivers with a set of buffers and then having the device driver continue to loop through those same set of buffers over and over again. Let's go back to that same sequence that we had on our previous slide where we talked about the chaining method, and with simple chaining the device driver would stop when it got to the last buffer. With the loopback method what happens is after the device driver processes the last buffer, it automatically goes back and starts processing the beginning buffer over again. This allows the device driver an infinite loop of buffers to process. One constraint here is that buffers can only be provided when dataflow is stopped. That's a difference from the straight chaining method. The chaining method, as we said, buffers could be provided at any point and time. When loopback is used, buffers can only be provided when dataflow is stopped. That makes sense because if you're in the middle of processing a loop of buffers and you want to add another one, where does that go in the loop? What we do is we constrain it and say additional buffers can only be provided when dataflow as been stopped on the device.

Typically that's not a problem. For the most part when this chaining with loopback is used, typically at initialization time, the application provides the device driver with the buffers it wants to process, and then never needs to re-supply them later on, hence the next bullet. Once the driver has been given these buffers, the device driver just keeps cycling through those over and over again. And there's virtually no overhead at that point in the system, the application never needs to call the adi_dev_Read() or adi_dev_Write() functions, the device driver just keeps iterating through those over and over again. The device driver never starves for data if it's sending stuff out, and it never runs out of a place to store data if it's taking data in from the device. A strategic use of the callbacks, allows the application to go and manage those buffers very effectively.

Subchapter 6d: Sequential Chaining

Let me talk about sequential chaining. Sequential Chaining is similar to the simple chaining method that I talked about earlier except in this case we don't have two separate queues for reads and writes, we keep all the buffers in a single queue. There's a field in that buffer that indicates direction, whether it's inbound or outbound, so if the device driver has to send data out, if the buffer contains data and wants to send it out, it's marked as an outbound buffer. If a buffer is provided and the device driver is supposed to fill that buffer with data, that buffer is marked as an inbound buffer. We keep these inbound and outbound buffers in one queue. And the buffers are processed in the order they are received. Just like in the chaining method, buffers can be provided at any point and time, and again I'm going to build a little chain here down at the bottom of the screen. Buffers can be provided at any point and time as I said; they can be submitted one at a time or in groups. Again we only have one queue this time, and you'll see that as I've marked in the graphics here, each buffer is marked whether it's an inbound or outbound buffer. At this point time I have three buffers in the chain, the first one is marked for outbound traffic so it contains data that we want the device driver to send out. The next two are marked for inbound data so they contain data that we want to read in from the device.

Again like the chaining method each buffer can point to data of different sizes, so you can mix and match the outbound and inbound on the chain, you can mix and match the sizes of the buffers on the chain. You may have some inbound buffers that are short, and then put in some outbound buffers that are bigger in size, there's no restrictions here as to the buffer size, they don't have to be fixed in sizes. There's also no restriction as to the number of buffers that can be queued at any point and time. You're basically by how much memory there is in the system. That's true for all the chaining methods.

Likewise any, all or no buffers can be tagged to generate a callback. Once processed, a buffer is not used again unless it's resubmitted. In the case of simple sequential chaining, the device driver starts with the first one that was queued, it processes it whether it's an inbound or an outbound one, and then if the buffer is tagged for a callback, it'll notify the application when that buffer has been processed. If it's not tagged for a callback, the device driver simply moves on to the next buffer in the chain without notifying the application. Again, a very powerful mechanism. Today our TWI Driver, which is kind of like an I^2C compatible driver, uses this method. If you're familiar with I^2C, it's a simple protocol where you typically do some writes first, where you write out the address of the device that you want to go and interrogate or provide data to. Then you do some number of reads, let's say you want to read data from a EPROM, you would send out the address of the EPROM that you wanted to access and then you would do reads to go and read the data from the EPROM. Sequential chaining is used to provide that specific sequence of reads and writes. When we reach the end of the chain the device driver stops. If more buffers are provided the device driver automatically restarts and starts processing those buffers.

Subchapter 6e: Sequential with Loopback

But we have this loopback option with sequential chaining. That's just like what we talked about in the simple chaining with loopback method, but in this case here we can loopback this sequence of inbound and outbound traffic. Just like in the regular loopback case, when the device driver reaches the end, the last buffer in chain, it automatically starts back with the first buffer in the chain. Likewise buffers can only be provided when dataflow is stopped, and most of the time that's done at initialization. The sequence that the application would do if it wanted to use Sequential Chaining with loopback would be basically open the device driver, configure it, provide it with buffers via the adi_dev_SequentialIO() function, some number of inbound buffers, some number of outbound buffers, whatever is appropriate, tagged accordingly. And then turn on data flow, and then at that point the device driver would just keep iterating through that sequence of buffers over and over. Now this makes it very handy so that the application never needs to re-supply buffers, so once it's given the buffers the device driver just keeps processing that data over and over again. Likewise similar to the regular chaining method, the device driver never starves for data and always has data to send out or pull in, very little overhead again to the application.

Subchapter 6f: Maximizing Throughput

With any of the chained dataflow methods we can pass a command called the streaming command. The streaming command is useful for dataflow that is very sensitive to interruptions in the dataflow such as audio if you don't want to have clicks and pops in audio or glitches in video data, the streaming command is very useful. One thing with the streaming command is there are actually some assertions that the application is making to the device driver when it uses the streaming command. The application ensures that the device driver will never run out of buffers, guaranteeing that the driver will always have an inbound buffer to process if it's been opened for inbound or bi-directional traffic, and will always have an outbound buffer to process if it's been opened for outbound or bi-directional traffic. The second assertion is that if buffers with callbacks are used that system timing ensures that interrupts aren't going to be lost in the system. It's a very convenient way for device drivers to maximize throughput. If a device driver is using DMA it allows us to run full speed, never interrupting DMA, always keeping the DMA engines running as fast as possible. As I said very useful for audio and video, it eliminates clicks and pops on audio sides and glitches in video. The streaming command can be used with any of the chained dataflow methods and allows the device driver to say operate at full speed and the application will guarantee that it will never run out of buffers to process.

Subchapter 6g: Circular Method

The circular dataflow method is the last dataflow method that we have, and that maps almost directly back to the auto buffer capability of DMA. When using the circular dataflow method a single buffer is provided and that buffer is broken up into some number of chunks or sub-buffers. Imagine a continuous block of data and then we divide it up as shown here on the screen in sub-buffers. There are some constraints on a circular dataflow method, Blackfin only uses 16 bits to define a circular buffer, or the size of a circular buffer, so it's a limited to a 64K byte size. But it's a very simple and efficient means to provide data to a device driver. When a device driver gets told to use a circular dataflow method, then that buffer is provided, the device driver starts at the top, starts processing the buffer, as it reaches the end of each sub-buffer the application has an option to be notified at the completion of each sub-buffer, or at the end of the buffer completion so when it gets to the very bottom of the large buffer, it can notify the application. Or it doesn't have to notify the application at all if so desired. A very simple way of providing a contiguous chunk of data and then breaking it up into some number of sub-buffers, and being notified whenever is necessary, at the end of sub-buffer, or at the end of the buffer, the large buffer.

Subchapter 6h: Deciding on Dataflow Method

How do you decide on which dataflow method to use? Well the ones that we have here, let's start with the one I just went through, the circular dataflow method. If your data fits in a 64K byte continuous block and it's streaming type data where you're processing typically audio, the circular dataflow method is a good candidate, dataflow method to use. Chained with loopback, if it's packet based data, typically bursty dataflows such as Ethernet, UART, USB using Chaining without loopback is the way to go for the most part with those types of devices and types of dataflow. Chained with loopback, that should be used with steady dataflow, typically streaming type video or streaming type audio when using chaining with loopback and you have the streaming command enabled, it avoids clicks, pops, glitches, and would basically always have a place to either store data into if we're taking data in from the device, or we always have a buffer to send out if we're sending data out through the device. You can imagine that when streaming audio or video, chained with loopback is a good method to use. SequentialIO with and without loopback, that's typically used in half duplex serial type devices. As I said earlier an I^2C compatible device like our TWI port uses sequential IO. That's very good if you need to specify a specific sequence of reads and or writes, and have those happen in a predefined order.

What's the typical programming sequence that an application uses to use the device drivers? Well going back to the slides that I showed earlier with the API, we've got the application on the left side and the device driver on the right. The first step that's used, after everything's been initialized, is the application opens the device driver. In the adi_dev_Open() function, the application specifies which device to use, so if there's three or four devices in the system, which one of those three or four to use; what direction it wants the device driver to open the device, be it inbound, be it outbound, bi-directional, whatever. We do the handle exchange where the application provides us with the client handle. That client handle again is used whenever we callback the application, whenever the device driver needs to notify the application of an event, it passes back the client handle. Also the device driver is part of the open function, returns to the application the handle to the device driver. All subsequent API calls that use that particular device, the application would include that device driver handle.

Chapter 7: Programming Sequence

Subchapter 7a: Chaining Method

After the driver's been opened, it's typically configured. One mandatory configuration is to specify which dataflow method of the various dataflow methods that I talked about earlier, chained, chained loopback, sequentialIO, etc... Any other parameters, default parameters or configuration parameters would typically be done immediately after the device was opened. In the example I'm going to show in a few minutes with our UART, we're going to configure the UART the specific baud rate, number of data bits, soft bits, and so forth. After configuration it's usually good practice to provide the device driver with buffers to process, particularly if the driver has been open for inbound traffic or bi-directional traffic. You want to be sure that the driver has a place to store data when we actually turn on dataflow. The application would provide buffers to either the adi_dev_Read() function or the adi_dev_Write() function, or adi_dev_SequentialIO(). No data transfer would actually take place at this point; all we're basically doing here is providing the driver with buffers to process. After we've given him the buffers to process, and again that's an option, in some cases we may not need to provide buffers before we enable dataflow, but in many cases we do so I've illustrated it here for the majority of cases where we would give it buffers. But after the buffers have been provided, we would then go and turn on data flow. Once we turn on data flow, the device driver actually starts moving data, it starts filling buffers as data is received from the device if it's been open for inbound or bi-direction traffic. It also starts trying to send data out if the driver has been opened for outbound traffic, or bi-directional traffic. So it basically starts processing any data that was provided in the previous step.

When buffers are processed, if they've been tagged for a callback, the device driver will notify the application, it'll invoke the application's callback function notifying the application that the buffer has been processed. Often the application will then go and say okay here's some more buffers to go and process, it can go and call the adi_dev_Read() function, the adi_dev_Write() function or adi_dev_SequentialIO(). Again no restrictions on when that is called with the exception of loopback, and with loopback obviously you can't go and provide additional buffers at that point. But this is the basic sequence that applications would go and follow. Open the device, configure it, give it some buffers, turn on data flow, when the device driver processes the buffers, it notifies back the application that they've been processed, the application then optionally goes and gives the device driver additional buffers. And that sequence iterates through over and over again.

Chpt 8: UART Example

Subchapter 8a: Overview

Let's map that sequence to a specific example. In the example I'm going to show, it is just a basic UART example. It's just a very simple talk through program. I have a BF537 EZ-Kit and I've connected the serial port on the EZ-Kit to the serial port on the PC that we're using here. I'm going to start up HyperTerminal from Windows, we're going to configure it to run at 57600 for a baud rate, 8 data bits, 1 stop bit, no parity. And all we're going to do is just echo those characters back. As we type a character into HyperTerminal it's going to be sent down to the EZ-Kit, the device driver is going to receive that piece of data through a buffer, pass it back to the application. The application is simply going to send it right back out the UART and back out to the PC where it will be echoed back on the screen. Very simple example, but it demonstrates the sequence of how to go and use the device drivers, we're going to use the chained dataflow method, so we'll see how that works, and we'll actually use callbacks because we're going to be notified when the UART driver has processed these buffers.

Subchapter 8b: UART Programming Sequence

Let's go back to the sequence that we're actually going to use and we'll walk through the steps that we're going to go through in the example. The first thing the application is going to do, it's going to open the UART driver, it's going to specify that we want that driver to be open for bi-directional dataflow. In other words we want to open it for both input and output. We're going to go and configure the UART driver, we're going to tell it we want the chained dataflow method, we're going to set the baud rate to 57600, we're going to set 1 stop at 8 data bits, we're going to configure it for how we want it to operate. We're going to provide some buffers for the device driver to fill. We're not going to send anything out right off the bat, but we're going to provide some buffers through the adi_dev_Read() function so that the UART driver will have a place to store data when data is received from the PC, from the HyperTerminal. We're then going to turn on data flow and once we turn on data flow then what's going to happen is we're going to switch back into HyperTerminal. We'll enter a character and we'll see that character get received through the UART driver and brought into the application. When we go and type in a character, the driver is going to make callback to the application saying, "Hey I just received some data." The application in it's callback function is simply going to send that right back out using the adi_dev_Write() function, echo it back out to the HyperTerminal. Once that buffer has been sent out, the driver again is going to notify the application saying, "Okay I've sent that buffer out." The application is then going to take that buffer and put it back on the read channel again. Effectively we're just going to have this sequence operate over and over again. Where the application is going to give the driver a buffer to fill, the driver is going to tell the application when it has processed it, the application is then going to send it back out saying, "Here go and send this back out to the PC." The UART driver is going to come back and say, "Okay I've sent that one out." And the application is going to go and re-queue that buffer back on the read channel. We'll have this loop echo or talk through program.

Subchapter 8c: Build/Run UART Example

Let me bring up VisualDSP. Here I've got VisualDSP, this is VisualDSP 4.0, and I have the December (2005) update loaded on this PC. I'm going to slide my window over so we can see what we have as far as files go. The only file in the example is this one little file that I'm showing here, uartexample.c, everything else is taken care of automatically by the default settings in VisualDSP. I go up to the top of my function and we'll just walk our way through the contents of this file. At the very front you'll see I have here the three includes that I talked about earlier in the presentation. I have our "services.h" file where we're pulling in all the System Services. I have "adi_dev.h" which are the device manager includes all the generic device driver information such as the API, common commands and etc… And then I have the UART device driver itself with all the specifics for the UART drivers such as commands for baud rates or for stop bits or what not, all the information that's unique to the UART device driver.

I'm just going to scroll down and I said we're going to use the chained dataflow method, we're going to use two buffers so I've defined this macro for the number of buffers. This first line here, Data, I'm going to highlight here, that's the actual data in memory where we're going to store the characters that are received. Remember I talked earlier that buffers and data can actually exist in different places. If we're doing an application where we're processing huge amounts of data say a video application, we may want to keep our data off in SDRAM, but we may want our buffers to be in a different memory space. On a dual core system with L2 memory, we may want to have our buffers in L2 memory for quick access, but keep our data off in SDRAM. This allows us to actually separate out the buffers themselves from the data that they go and process.

The next line here are the buffers themselves, and this construct here, ADI_DEV_1D_BUFFER, that's the buffer that I talked about earlier where I went through the different entries in the data structure, that pointer to the data, the element count, width and so forth. That's this data structure right here. Next is the handle, this is the handle that the UART driver is going to give us back, so when we open it, we're going to fill in this location in memory with that handle to the UART Driver. Then every time we talk to the UART Driver after that we're going to use this value to address it.

Scroll down, I've got three functions in this example, the main one that I'm going to show you in just a second, and then I've got two other ones; I've got the callback function and that's the function that the device driver is going to call when it needs to notify me of any events. And I've got this other function that initializes the system services. I'm not really going to go through the initialization phase of the system services, to learn more about the system services please review the system services module in the BOLD training. I think maybe I'll just step through that really quickly, but any of the details you should really look back at the system services module to see how those work. But basically all that's going to do is just initialize them for us.

Next we have our main program, and the first thing we have in our main program is this UART configuration table. And I talked earlier about once a device is opened it typically needs to be configured with some type of information. In the case of our UART drive we need to tell it what baud rate we want to run at, what dataflow method we're using, the number of data bits, the number of stop bits and so forth. We keep that just for convenience here in a table, we could pass these commands to a device driver individually or optionally with the adi_dev_Control() function, we can pass in a table of parameters that we want it to go and execute. That's what I've done here for simplicity, I just put all of our commands that we're going to go and pass through the driver here in this little simple table. Let me just walk through these and explain what they are. The first one, we're going to set our dataflow method and we're going to use the chained dataflow method that I talked about. We're going to set 8 data bits, so the UART data that we're going to process has got 8 data bits associated with it. We're not going to use parity, so I have that set to FALSE. We have one stop bit, and we're going to set our baud rate to 57600. And then this last entry just terminates the table so the driver knows the end of the configuration table.

Let me just step over this, and now we're going to go and initialize the services. I talked earlier about the initialization sequence that we go through; the specific order in which it has to occur.

I'm going to step into this real quickly and just show you this example here. Again I'm not going to go through all these values here. This is how we initialize the SDRAM on the part which is used for our EBIU service. And then we've got power, where we're specifying what the actual processor is that we're running, what package it's in and so forth. That's so that we can actually go and manage power effectively so the power services knows what frequencies are allowable to run at different voltage levels and so forth.

Let me step through this function here. Here we're going to initialize the interrupt service, and if you think back to that initialization screen, that's the first one that we needed to do. So we're going to step over that. Now we're going to initialize our SDRAM, and the way we do that is we call adi_ebiu_Init() function and we pass in that table just above that we looked at. And again for the details on EBIU initialization, power initialization and so forth please review the System Services module in the BOLD training. We'll just step over this, and now we're going to initialize our power service. Those are the only three services that we're using in this example. We're not using DMA, we're not using any of the other services, so those have been left out of the sequence here. So just these three services we're going to go and initialize.

Now I'm back in my main program, and remember after we initialize the services, then we can go and initialize the device manager, so that's what I'm going to do now, I'm going to go and initialize the device manager. And all I'm going to do is tell the device manager that I'm going to be using three, or in this case here, one device driver. In order to conserve memory, memory is a very precious commodity in embedded system, so rather than rely on a dynamic allocation scheme or predefined some fixed number of device drivers to use, we allow the application to specify how many device drivers are going to be used simultaneously. And then the application gives us only the amount of memory that's necessary to go and manage that number of device driver simultaneously. That's what the initialization function is for the device manager. All I'm going to do is give the device manager the memory that it needs to manage this one UART device.
Before I move on, I want to point out that for each of these API calls, let me just scroll back up to that one I just made, I'm storing the result in this variable here called Result. I talked earlier about the return codes, that every API function in all the services and in the device drivers returns a result. If that value is zero that means everything worked properly. If it's non-zero that's an indication of some error or some informative event. Typically what happens is an application needs to test for zero, if it's zero then everything worked fine, if it's non-zero then the application needs to take some action to figure out what occurred. In this example here hopefully everything returns as a zero. And if I put my cursor over the result you'll see that yes indeed, there were no errors when we initialized the device manager.

Once the device manager has been initialized, the next step I want to do is open the UART driver, remember that's the first step in that sequence slide that I showed just few moments ago. We're going to open the UART driver, in the comments up here you can see the parameters that we're passing into the UART, to the adi_dev_Open() function. We're passing in the handle to the device manager, so we know which device manager to go and use. We're passing in the entry point of the device driver that we want to open. This identifies the UART. There are different entry points for each device driver that we have, so there's a UART entry point for the UART driver, there's a PPI entry point for the PPI driver, our AD1836 audio CODEC, has an entry point for that. That's how the system knows what specific device driver is being accessed.

Next up is the device number itself, and in this example here we're opening the zero-th device, or the first UART in the system. If we had a system with multiple UARTs we could pass in at zero or one or two or whatever device number it is that we wanted to go and open. In the device drivers in System Services all indices start at zero, so zero is actually the first UART. The next parameter is our Client Handle. And remember I talked about in the adi_dev_Open() function we exchange handles, well this is the Client Handle, so every time the device driver wants to callback the application it's going to pass back this Client Handle. Again this value has no meaning to the device driver, supposedly it's of some significance to the application, typically if an application is using multiple device drivers it may use different Client Handles for each of those device drivers so that it can uniquely identify which device is calling it back. In this case here I'm just using 0x12345678, so when we look at our callback function we'll see that value being passed back.

Next is the address of the UART handle, the device driver is going to fill this location in memory with the handle to the UART device that we're going to open. Every time we go and talk to that UART from this point forward we're going to use this UART handle to uniquely identify that device. We're opening it for bi-directional dataflow, so we want open it for both read data in and send data out. The next parameter is a NULL, we're not using DMA in this case, the UART driver that we have today does not require DMA so I don't need to pass in a handle to a DMA service or anything like that. The next is a handle to a callback service, to the deferred callback service I should say. And if you'll recall from the System Services module, callbacks can be executed in one of two ways, either "live", which means callbacks are executed typically at hardware interrupt time, or "deferred", meaning we defer that callback for a lower priority. For simplicity here I'm passing a NULL which means our callbacks are going to happen live. A for more information on how the callback service works and the specific differences between live callbacks and deferred callbacks, again please review the System Services module. The last parameter that we pass through the adi_dev_Open() function is the address of our callback function. This is the function that the device driver is going to call whenever it needs to notify us of some asynchronous event. I'm going to step over this, and I look back at our result; it's zero so we opened the device successfully.

Next up we're going to configure the UART, so remember that parameter table that I talked about at the top? We're going to tell it we want to use the chained dataflow method, 8 data bits, no parity, 1 stop bit, 57600 baud rate. We're going in and configure the device driver now to the adi_dev_Control() function. We're going to step over that. And you can see the result is zero, so that worked properly. There's that UART handle that we talked about, so that's the handle that we used to identify that UART for every API function that we make after adi_dev_Open().

Now I'm going to create our buffers. And if you recall back to the slide that I showed earlier about the one dimensional buffer you'll see these fields right here, that are shown here on the screen. There's the data field, element count, element width, callback parameter, pointer to the next, and this additional info parameter. What I'm going to do now is I have these buffers and I created two of them, so we're going to populate that buffer with the information. We're going to point the data field to the data that we declared for the buffers; that's where the driver is going to store data as it's received or send data out as we want to send it out. Each piece of data that we want to process is, there's only one element in that data, and each element is one byte wide. We've mapped our buffers to one character, so each buffer can contain a single character, a single 8 bit character. Our callback parameter is next, and we want to be called back whenever the buffer is processed so we have a non NULL value in this field. In this case what I'm doing is I'm passing in the address of the buffer itself. When our callback function is invoked, our callback function will be told the address of the buffer that has just finished processing. That way we'll be able to map it back very simply. I'm pointing to the next buffer in the chain for this pNext value, so I'm creating a chain of buffers, in this case here there's only two links in the chain. This is going to point to the next buffer in the chain. Additional info, this really isn't used today but it's there as a place holder as I said just in case a device driver needs additional information that we haven't thought of yet, we have a place holder for that should it occur. In the case of this example what I'm going to do so that we can identify whether these buffers are inbound buffers or outbound buffers, I'm just going to put a value in there for us to, just for illustrative purposes so we can see. What I'm going to do is put a zero in that field if it's an inbound buffer, and I'm going to put a one in that field if it's an outbound buffer. What I'm doing here is I'm creating these buffers for the UART to fill with data, so I'm putting a zero in the additional info. Again, not used typically but just for the purposes of this example.

I'm just going to loop and create these buffers, and the last buffer in the chain I'm going to point it's next pointer at NULL, so we want to have, when we pass in a chain of buffers that chain has to be NULL terminated, so the last buffer in the chain points to NULL. I'm just going to run through to this step here. Now what I've done is I've created these buffers, and I've created these two buffers, each pointing to one character in memory that we want to go and fill. And I'm going to tell the device driver here are these buffers that we want you to go and process. I'm going to call the adi_dev_Read() function to provide those buffers to the device driver. I step over, we can see it executed properly so no errors in the system.

The next function I'm going to do after we've configured it, we've given it buffers, a place to store data when data's received. Now all I need to do is just turn on data flow. I'm going to call the adi_dev_Control() function and say turn on data flow. Before I do that I want to scroll down and show you the callback function. This is the function the device driver is going to invoke when a buffer is processed. Let's look through that and see exactly what's going to happen, scroll up here so we can see. Here's our callback function, we're passed in three parameters, the client handle, remember that's that 0x12345678 that we used in the adi_dev_Open() function. That's this value here. Remember every time the device driver wants to notify the application of an event, it passes back that client handle. That's the first parameter in the callback function. Whenever our callback function is invoked, the client handle should be 0x12345678.

Next up is the event that occurred. And the event that we're going to be looking at is this one right here, the buffer processed event. The device driver is going to callback this function whenever any event occurs, and the one that we're actually concerned with is the ADI_DEV_EVENT_BUFFER_PROCESSED event. We're going to do a key, or a switch off of that event when it's passed back. The last parameter is actually event dependent, so depending upon the event that occurred; this last parameter is significant for that event. This is all described in the documentation, but for a buffer processed event the pArg value is the value that was passed as the callback parameter value right here. What we did if you recall is that we put in the callback parameter we put the address of the buffer itself. What's going to happen in our callback function, the device driver is going to call us back whenever it finishes processing a buffer. It's going to give us back our client handle, our 0x12345678, the event type that we're expecting is this ADI_DEV_EVENT_BUFFER_PROCESSED event, so the device driver is going to say I processed this buffer. And then the last parameter is going to be the address of the buffer itself (pointing to the buffer).

If we look through the code in the function, the first thing I'm going to do is I know my pArg value is going to point to our buffer, so I'm just going to, rather than have to cast it, I'm just, because we use a void* here, I'm just going to declare a temporary variable that is of that type so I can avoid any casting. When our function gets called we're just going to get the address of the buffer. We're going to make sure the event that we're expecting is indeed the buffer processed event. And then we're going to look at that additional info field. Remember I said that for inbound buffers we're going to put a zero in that field, for outbound buffers we're going to put a one in that field. When we receive an inbound buffer or when we receive a buffer we're going to make sure that it's, or test whether or not it's an inbound one. If it is an inbound buffer, we're actually going to change the tag and say make it an outbound buffer. And again, typically the additional info field doesn't need to be filled in; we're just doing this for our own purposes so we can illustrate the read buffers and the write buffers. We're going to make sure that it's NULL terminated and then we're going to go and send that buffer out to the UART driver. In other words when I've received a character, when I've received a buffer, when the UART tells me I've received some data from the HyperTerminal, all I'm going to do is I'm going to just go and send it right back out, submit it on the write channel and say I want to write this data back out the HyperTerminal. After the driver goes and processes that, it's going to call us back again after it processes the write buffer. Again we're going to look at that additional info field, we'll identify it as being an outbound buffer. Now we're going to change it and say okay now we want to go and put this back on the read channel on the read side, or the read queue I should say. I'm going to put a zero in there so that we make sure it's NULL terminated, and then we're going to call adi_dev_Read(). I'll come back to this and we'll step through this step by step, but just to give you a brief over view of how that works.

I'm going to put a breakpoint right here in our first location so that way we'll halt when the device driver calls us back. We'll go back up to the main program here, I'm going to turn on dataflow so this is going to say okay UART, go, start moving the data. And then all I'm going to do is just sit in the loop not doing anything, just waiting for characters to come in. If I click run, now you'll see down at the bottom of the screen we're actually running. We're really going to sit in this wait group, in this while loop here waiting for characters to be received. I'm going to scroll this over so we can see some of the information here. I'm using VisualDSP, and I've got the Blackfin memory here, we're running so everything's grayed out. But I've got our data buffers right up here so you'll see, hopefully see the characters that we're going to type in to HyperTerminal, they should appear up here in this field up here. Let me go into HyperTerminal, here's HyperTerminal, I've got it configured to match the settings that we have for the device driver, 57600, 8 data bits, no parity, 1 stop bit, and I'm going to type in a character so I'm going to press the letter 'a'. And when

I type the letter ‘a’, HyperTerminal is going to send that character out to the EZ-Kit, our driver is going receive that character and is going to complete process, it’s going to store it in our buffer, and notify the application by calling the callback function, that’s where we have the breakpoints. When I press the letter ‘a’ here we should see VisualDSP stop and hit our breakpoint. I press ‘a’ and I go back into VisualDSP, see down here at the bottom of the screen we’ve halted, and we’re actually in our callback function.

The driver has said I’m notifying the application of some event, so let’s just step through the callback function, and we’re going to do a case of the event, now the event we’re expecting is this one right here, that the data buffer has been processed. Now we’re going to check and be sure it’s an inbound buffer, or test to see if it’s an inbound buffer. And it should be. In the top right here you’ll see there’s that letter ‘a’ that I typed in, so you’ll see that data that we talked about, so if I scroll back up to where we defined the buffers, I said for each of these buffers, here’s the data that we’re going to go and process so that’s the location of memory so you can see where it’s stored the data. Yes indeed, it’s an inbound buffer, so now I’m going to do just do our little trick so we see what’s happening, touch the additional info field and put a one in there so I’m going to send it out. I’m going to make sure it’s NULL terminated so there’s no other, there’s just one buffer in this chain, there’s not a whole chain of buffers that we’re sending out. And then I’m going to call adi_dev_Write(), and via adi_dev_Write() I’m just going to say UART driver here’s another buffer I want you to go and process. I want you to go and send this out, so there’s our buffer. When I hit run now what’s going to happen is we’re going to give that buffer to the device driver, it’s going to send it back out to HyperTerminal, and then it’s going to call us back again after that buffer has been sent out. We’ll just go back into HyperTerminal, and you’ll see that there’s nothing has been echoed back to the screen yet. Let me click run, and what should happen is that we’re going to go and send that out and we’ll immediately come back into our breakpoint because the driver will send the buffer out it’ll want to call us back again. I click run and sure enough we hit our breakpoint again. I go back into HyperTerminal, you’ll see up here that it’s certainly echoed back the letter ‘a’ that I had typed in. And it’s called us back. Now if I step through the callback function, buffer processed event. This time it’s a buffer that was sent out, our additional info field we had set to one. Now all we’re going to do is we’re going to set it to be a zero again because we’re going to go and give it back to the UART to fill with more data. Terminate it, NULL terminate the chain so there’s only one buffer in the chain. And then we’re going to call adi_dev_Read(). If I run again we should just basically start to see us just running because the driver was saying here take this buffer, go and fill it with more data and let us know when you’re done. If I click run, see we’re indeed running and the driver is now waiting for another piece of data to come in. And will call us back. This time if I go back into HyperTerminal and type the letter 'b' we should hit our breakpoint again. There we go. We're back in our callback function, if we look at the data you'll see it's a letter 'b' now. Again we're going to go through that same loop again, we're going to mark it as an outbound buffer, NULL terminate it, again we only have one buffer in the chain, and send it back out. Run again with adi_dev_Write(), we should echo the 'b' character back out to the screen. Sure enough there it is. Back into VisualDSP, and the driver is now telling us okay I've sent that buffer out, and notifying the application that that buffer has been processed. Again I'm going to step through, tag it as an inbound buffer, make sure it's the only buffer in the chain, and then put it again back out on the adi_dev_Read() function.

I take away our breakpoint now so we don't do any more halts, click run, if I go back into HyperTerminal you'll see that we just keep echoing characters back. A very simple example that shows the basic steps of using the device driver, how a driver is opened, how it's configured, how buffers are provided for inbound traffic, how buffers are provided for outbound traffic, how the device driver notifies the application when buffers have been processed. You can see it's a very simple example. Also another thing to note is that aside from initialization of the system services, everything else was taken care of by the device driver itself. UART interrupts have been hooked, if we were using DMA, we don't in this case, but DMA would have been managed appropriately. The UART device driver actually made calls in to the power management service, to determine the system clock frequency which is used to calculate the divider ratios for the UART so it could run at 57600 baud rate. All that happens within the drivers themselves, and again that's all built upon the system services. We're putting together a collectively between the services and the drivers, a very powerful and a very easy to use environment that applications can use to very quickly get code up and running and take their applications to production much quicker then in the past when device drivers had to be created for each device, each customer had to create their own device for their own specific application and so forth. **B**y providing these device drivers we can actually shorten our customer's time to market significantly.

Chapter 9: Conclusion

Subchapter 9a: Additional Information

Okay let me go back into the presentation. In conclusion I think we've shown that the device driver's allow very fast development time by providing a stable software base with the services, with the device drivers. Less reinvention, applications or users don't need to create everything from scratch they can build upon the System Services that we have, the device drivers that we provide. Very modular software so that compatibility the drivers work concurrently with one another. Integration, multiple software components are working concurrently, hardware resources are managed effectively. And portability; we could just as easily have taken that example that I ran on the BF537 EZ-Kit and run that exact same example on a BF533 EZ-Kit or a BF561 dual-core EZ-Kit. There's really no changes at all that are required to the application because the API to the device driver is the same, the services operate the exact same way, functionality operates the exact same way. A very efficient way for customers to move from a processor that exists today to in the future where we have better, stronger, faster processors to migrate their applications extremely quickly to the latest processors that we provide.

For additional information the pointers to the documentation, our Device Driver and System Services Manual is located in the technical library section on Analog's website, located here at this address. We've augmented that with an addendum in September of 2005 with some additional information about newer services and some of the additional device driver capabilities that have been added in. There will be more documentation coming, In the VisualDSP, installation directory, look in the directory Blackfin/doc and you'll start to see additional information as it becomes available, as more documentation is provided, as additional drivers are provided with the distribution, we'll have additional documentation, locate that in the documentation directory in the VisualDSP installation.

If you have any specific questions you'd like to ask please click the 'ask a question' button at the bottom of your screen, or send an email to processorsupport@analog.com . Thank you very much for watching the device driver module of the BOLD Training, hopefully it's been helpful to you.
Thanks again.