

Presentation Title:

Using System Services Library and Device Drivers with VisualDSP++ Kernel in Blackfin® Applications

Presenter Name: George Kadziolka

Chapter 1: Module Overview

Hello, my name is George Kadziolka, president of Kaztek Systems. Today we are going to be talking about using System Services and Device Drivers with VDK in Blackfin applications.

Subchapter 1a: Why Use System Services Library and VisualDSP++ Kernel?

So, why use System Services in VDK? First of all System Services Library is there to abstract the low level hardware from the application, which makes it easier to develop code for the Blackfin processor. When an application starts to get a little bit more involved from the perspective of user I/O, device I/O, processing and control, a real time kernel is often considered. The real time kernel enables an application to be broken up into separate tasks or threads each of which can be dealt with a lot more easily. Then threads can synchronize between each other using mechanisms such as semaphores and messages. So, VisualDSP Kernel or VDK is a pre-emptive multitasking kernel, which is developed by and supported by Analog Devices. Both VDK, System Services and Device Drivers are also part of the VisualDSP tool suite.

Subchapter 1b: System Services Library Device Drivers vs. VisualDSP++ Kernel

Now, system service drivers versus the VDK drivers. The System Services Device Drivers do work with VDK and they supersede the VDK device driver model in Blackfin applications. System services Device Drivers exist for both on and off chip peripherals and they are actively developed and supported for Blackfin.

VDK does offer a device driver programming model. There are examples that are provided for how they are used, but otherwise they are not supported. Now, the VDK device driver model is still used to support I/O for the SHARC processor family, as well as, multi-processor messaging for the dual core Blackfin processor, such as the BF561.

Subchapter 1c: Demo Project Overview

In this module we will walk through the development of a Blackfin project that uses System Services Library and Device Drivers in conjunction with VDK. The demo project will be an audio talk-through application, which will use a pair of push buttons to provide a simple up / down volume control. The application will use a BF537 EZ-KIT Lite for the demonstration, which will come up later on, but we will demonstrate the key elements required in order to customize System Services for a custom target.

Subchapter 1d: Prerequisites

In order to get the most out of the module the viewer will need a basic understanding of several concepts, which you can find from existing online modules or from a multi-day Blackfin workshop. One is on VDK basics. We need to be able to know how to create VDK projects, how to set up threads, semaphores, events, enable message handling and also deal with thread synchronization using post and pend functions. The user also needs to understand System Services basics, being familiar with initializing or configuring the System Services managers and the related API functions. And finally, there should be some familiarity with System Services Device Drivers. In other words, using the Device Drivers, knowledge about the buffers, setting up buffers and using callback functions.

Chapter 2: BLACKFIN APPLICATION OVERVIEW

Subchapter 2a: SSL/VDK Demonstration Project

So, let's talk about our Blackfin application at a high level. First of all we are going to be developing an application that uses both System Services Library and VDK. So, this will be a multithreaded VDK project. We are going to be using elements including messaging, semaphores, and events for inter-thread synchronization. We're also going to be using the System Services Library and Device Drivers. So, several System Services managers are going to be used including the interrupt manager, the EBIU manager, power, port, flag and the device manager will be used in this application. We're also going to be showing how the EBIU and power manager setup can be customized in order to deal with a non EZ-KIT board, in other words, some custom board that you might be developing with. We're also not going to use deferred callbacks in this particular demo. Instead all the callbacks we use are going to be live. We're also going to be using a pair of device drivers, specifically the AD1871 and the AD1854, which correspond to the digital to analog converters that are available to us on the EZ-KIT Lite.

Subchapter 2a: VDK Setup

Let's talk about the VDK setup first. VDK is going to be setup with four threads and a main function. So, the threads are the input thread, the process thread, the output thread and the volume control thread. The names themselves are fairly explanatory. There is also going to be a main.c file, where we're going to initialize VDK and do the System Services setup.

In this particular application the threads are going to be instantiated at boot time. In other words, all these threads will be boot threads.

So, what we are looking at here is a screen capture of the kernel tab for this application. We're looking at the different threads. So, here is the input thread. We're using all the default settings so the priority hasn't been changed. The only change that we are making

is we are enabling messaging. We're setting this to true. So, this is going to be true for the input thread, the process thread, the volume control thread and the output threads. We're also showing here the boot threads have been created; In, Proc Volume, Out, basically the instantiations of the input, process, volume control and output threads respectively.

We're also going to be setting up several semaphores and events. So from a semaphore prospective there are two semaphores that we are going to be creating. One is called `ADC_DATA_READY`. This semaphore is going to be dealt with in the AD1871 callback function to let the rest of the application know the input buffers are ready for processing. There is also going to be a `DAC_DATA_SENT` semaphore set up and this will be dealt with in the AD1854 driver callback function.

We're also going to set up an event. This event is going to be called `VOLUME_ADJUST` and it will be based on two event bits, specifically the `UP_VOLUME` and the `DOWN_VOLUME` bits. This is going to be used to provide some sort of indication that there was a change made in the volume control level.

So again, taking another look at the kernel tab. This time we are looking at the semaphores and the events configuration. So, semaphores we're setting up for two semaphores. Here's the `ADC_DATA_READY`. Semaphores in VDK are generally counting semaphores. We haven't touched any of the default values; so, we see that the max count is set to a large value. So, there's the `ADC_DATA_READY`, `DAC_DATA_SENT`, and here's our event down here called `VOLUME_ADJUST`. There's the two event bits that are the dependent bits for the `VOLUME_ADJUST` event, `UP_VOLUME`, `DOWN_VOLUME`. The `VOLUME_ADJUST` is made to be of Type 'Any', which means either one of these two bits needs to be true in order for the `VOLUME_ADJUST` event to be true.

The last thing we want to talk about as far as VDK setup is messaging. We're going to be using message loopback in this application. This allows the thread scheduling to be driven mainly by data flow.

We're going to be using four message channels. The `IBUFF_PTR_CHANNEL` is going to be one, which is used to convey pointer information regarding the input buffer, which has the address of the input buffer that's ready for processing. We're going to use the `OBUFF_PTR_CHANNEL` in order to convey pointer information for the output buffer that is going to be filled when we do the processing. Then the `CONTROL_CHANNEL` will be used to convey volume control information on to the process threads. When ever there is a volume control change we'll pass that on to the process thread over that channel. The messages are going to be re-circulated rather than be continuously created and destroyed. The messages will be returned over the `RETURN_BUFF_CHANNEL` back to the center for reuse. So, this recirculation does two things for us., It improves reliability because we are going to know when the messages are initialized and if we have enough memory or not. But it also saves on efficiency because we are not constantly creating and destroying them.

Subchapter 2b: VDK Control Summary

So, we're going to take a look graphically at the interaction between the various threads and this is a legend that the graphics will be using.

So, here on this page what we see is for instance the `AD1871_callback` function posting a semaphore, the `ADC_DATA_READY` semaphore. The input thread is pending on that semaphore. What the input thread does in response is to calculate the address of the buffer that needs to be processed next and it sends that address in the input buffer pointer message over the `IBUFF_PTR_CHANNEL`; so, it posts that message. The process thread in turn is pending on that messages coming over that channel. Once the process thread is done with that particular pointer it returns the message back to the input thread so it posts over the `RETURN_BUFF CHANNEL` back to the input thread, which is going

to be pending on it. Likewise, the AD1854_callback function when it gets called in response to another data buffer being sent out will post a DAC_DATA_SENT semaphore and the output thread is going to be pending on that semaphore. And again when the semaphore has been posted what the open thread will do in response is to calculate the address of the next buffer that is supposed to be filled by the process thread and post that in a message, the output buffer pointer message over the OBUFF_PTR_CHANNEL to the process thread. So, the process thread is pending on that particular message as well. And again, once the process thread is finished with a particular message, once it has the pointer information; it will return the message back to the output thread over the RETURN_BUFF_CHANNEL. Again, the output thread is pending on that.

We also have some volume control information which is being passed on. What we're using is we're using a pair of pushbuttons and these pushbuttons are tied into the flag pins. We're going to be setting up a flag_callback function. So, the flag_callback, what it will do is either set the UP_VOLUME event bit or the DOWN_VOLUME event bit depending on which pushbutton was pressed. And the OR of those two event bits will be used to generate the VOLUME_ADJUST event. So, the volume control thread is going to be pending on that VOLUME_ADJUST event. So, if either one of those pushbuttons are pressed, then the volume control thread will, based on which of the event bits was pressed, either adjust the volume up or adjust the volume down and send that new volume level over the volume message to the process thread over the control channel. The process thread in turn once it receives the volume change information will take it, store it and return the message back with the RETURN_BUFF_CHANNEL to the volume control thread. So, that's how the threads interact at a high level.

Subchapter 2c: System Services Library Setup

We're going to talk a little bit about System Services setup now. In main.c what we are going to be doing is to initialize all the System Services managers that are going to be required by the application. This will include the interrupt manager, the EBIU manager,

which is where we specify the SDRAM timing information,. We're also going to set up the power manager and this is where we specify the clock-in and the maximum frequency of operation information. We're going to be using DMA in our application; so, we need to setup the DMA manager. Port control, flag control will be set up. And finally after all the required services are initialized we're going to set up the device manager.

The application will be using a pair of device drivers; so, we're going to be opening the AD1871 and the AD1854 device drivers in the input and output threads, respectively.

We did mention earlier that all callbacks in this application are going to be handled live. This means that when we open up the drivers there's going to be a place in there where we normally would pass the handle for the deferred callback manager; so, instead we are going to place a null in that position, which will make the callbacks live, in other words, dealt with at interrupt level. Now in this particular application we're also going to be using circular buffers for data. So, we're going to set up the callbacks to occur whenever a sub buffer has been processed. As we mentioned before, within the callbacks we're going to be posting these semaphores either the ADC_DATA_READY in the ADC callback function and the DAC_DATA_SENT semaphore to be posted in the DAC callback function.

We mentioned that we're using flags in this particular application; so, we need to deal with them. On the BF537 EZ-KIT we're going to be using pushbutton 1 as the UP_VOLUME control and pushbutton 2 as the DOWN_VOLUME control and these are connected to the PF2 and PF3 flags, respectively. What we are going to be doing is registering a live callback with these flag pins and then what's going to happen with the callbacks is that if, for instance, PF2 is pressed we're going to set the UP_VOLUME event bit and if it's the PF3 will set the DOWN_VOLUME event bit.

Chapter 3: THREAD OVERVIEW

Subchapter 3a: main.c

Now, what we're going to do is get into some details on the threads themselves. So, the first stop is the main.c module. Technically this is not a thread. This is actually the same main function that gets called after the C runtime header is done setting up the runtime environment. What we are going to do within main.c is to initialize VDK and setup System Services. So, we're going to make a call to a function called the `adi_ssl_init` function. Now, this is part of a pair of modules called `adi_ssl_init.c` and `adi_ssl_init.h`. These are found where the tools are installed under the Blackfin/Examples/Common Code folder. So, what you'd do is take these files, copy them into your project file, and then you could adjust them as required for your particular application. What we typically do is to edit the .h file. There is a number of #defines in there that are used to specify the level of System Services that are required in a given application. So, in our case here we have these #defines, which we have just pulled out of the header file.

`ADI_SSL_INT_NUM_SECONDARY_HANDLERS` we're setting to four. If you recall from an earlier presentation we know that we have many more peripheral interrupt requests available than we have interrupt levels at the core. So you have multiple interrupt requests mapped at the same level. So, anything beyond one handler we need to specify how many additional handlers are required. So, this just allocates memory for four additional interrupt handlers.

We're not using deferred callbacks in this application, as we mentioned before. So, we set the number of deferred callbacks service to zero. We are using DMA. We're using two DMA channels one to bring data in from the ADC and one to send data out to the DAC. So, two DMA channels are being reserved. We are also using two flag pins to trigger a callback. So, the number of flag callbacks is being set to two.

Finally, `ADI_SSL_DEV_NUM_DEVICES`. How many devices are going to be open at any given point in time. We're going to set this to four. So, even though we've got

those two device drivers that we mentioned the AD1871 and the AD1854. These drivers are stacked on top of the SPORT drivers. You basically need memory allocated for each of the four drivers in total that are going to be used.

So, here's a look at our main.c module and we see the first function we call is to VDK_Initialize. So, this initializes the VDK. Then we make a call to my_ssl_init. I typically take the adi_ssl_init edit it, customize it for the boards I'm working on, which we'll talk about shortly. Also, if you recall from the Device Drivers and System Services modules, all these API functions will return a result code and you are always looking to see that the result code is zero. We're doing the same thing here. In the event that the result code is non zero, what we're going to do is dispatch a thread error. We're going to call this a ssl_init_failure so this is the VDK system error that we're going to be indicating. We're going to pass the result code that gets returned. That way there we can basically use the resources within the VDK debug capabilities in order to help figure out what went wrong.

Next we're going to use the adi_pwr_SetFreq function. Notice the parameters being passed are zero, zero. All this does is sets the maximum core clock (CCLK) and S clock (SCLK) frequencies for our target. And again, if there was ever a non zero, we're going to throw another VDK error.

We're going to make a call to a function called reset_ADC_DAC. All this does is just causes the programmable flag pins that are tied to the reset pins on those two devices to toggle, thereby resetting those two peripherals. Finally, we make a call, well not finally, we make a call to the flag_init function and within this function what we're going to do is to set up the two flag pins that are going to be used for the UP_VOLUME and DOWN_VOLUME control and register callbacks with them as appropriate and likewise we check to make sure that occurred successfully. Last thing that we do, is we make a call to VDK_run. So, this function is what actually causes VDK to start up.

Subchapter 3b: System Services Library Customization

Let's talk a bit about customization of System Services. When you are looking at customizing System Services, there is really two managers that are impacted. One is the EBIU Manager, one is the Power Manager. The EBIU Manager is mainly responsible for configuring external memory. This could be SDRAM or DDR for some of the Blackfin processors. If you have asynchronous memory you can also configure that for the EBIU manager. What we need to do is to pass along the data sheet timing information for the SDRAM device, the main external memory device, to the EBIU manager. We'll see this in an example coming up on the next page.

The Power Manager is the other manager that needs to be set up. It is responsible for managing the dynamic clocking and core voltage control functions for the Blackfin. What we need to do is to describe what the clock-in frequency is going into our processor and also what the maximum frequency is that the processor can run at.

This is just an excerpt from the EBIU manager's setup. We'll take another look at this later when we look at the demo application. What's important to note that what we're doing is we're setting up a command value pair configuration table. In this case I'm calling it MyCustomTarget_ExtMem, and what we are doing is we are taking the information from the SDRAM manufactures data sheet and just setting it up in this table. In this case we are using a pair of Micron SDRAM devices which are the ones on the Blackfin EZ_KIT board. So, for instance, we're setting the bank size to this parameter, which is set up here. Same thing with the column width, the page size, and so on and so forth. So, we set up this configuration table. We could include async memory setups as well, then we make a call to `adi_ebiu_Init` passing a pointer to this table and again just checking to make sure that there was success by testing it here against the `ADI_EBIU_RESULTS_SUCCESS` macro.

That's really all there is to the EBIU manager setup. With this timing information what will happen is if you go and change the S Clock frequency, for instance, the power

manager will automatically make a call to the EBIU manager and have it adjust the SDRAM timing information based on the information provided.

Power manager, what we are doing is like wise setting up a command value pair table in this case I am calling it MyCustomTarget_Power. So, the first parameter is the processor variant. In this case we are indicating this is a BF537 processor running at 600 MHz. This is used to set what the maximum clock frequency is for the processor that we're using. We also have a package type MiniBGA, external voltage 3.3V. And the other parameter that we need to specify is this one here, the clock-in frequency, which we have it set to 25 MHz. So, again this is what the EZ-KIT is running at. So, these two parameters are what you would need to change or address in your custom setup. So, again we call the adi_power_init function with a pointer to this command value table pair. And again check to make sure that we had success with that.

Subchapter 3c: Flag Setup Details

We talked about that flag_Init function that was called from main as well. So, this is what we are looking at here is the first part of that function call.

So, what we are doing here is setting up the flag pins that we are going to be using for the UP_VOLUME and DOWN_VOLUME control bits. So, we are first going to deal with the volume up pushbutton first. We first open the flag, which flag are we opening up, the PB_UP. So, PB_UP would have been #defined to the programmable flag pin. So, we first open the flag pin and then we set the direction as input, because these are going to be use as an interrupt source.

Then we install a callback. So, which flag pin are we using? It is the PB_UP flag pin. We're going to generate a PortFG_A interrupt request when this occurs. The interrupt request will occur on a rising edge. We're not going to be using this to generate a wake up; so, this is false. We have a client handle, in other words, a user defined parameter that we can pass as a void pointer. What we are passing in this case is the identifier, the VDK Identifier, for the UP_VOLUME event bit. We're going to use this in the callback

function in order to identify which bit we are going to set. I mentioned that the callbacks are going to be live; so, the next position is where we would otherwise specify the handle for the deferred callback manger. So, we're going to replace that with a null, which again will make this live. And this is the callback function that will be called, the `flag_callback` to deal with the flag pins. This is repeated for the volume down pushbutton flag.

Subchapter 3c: Thread Details

Let's go through the various threads now starting with the input thread. What we do in the input thread is to use the `adi_dev_Open` function to open up the AD1871 device driver. We're then going to be creating an input buffer, which is going to be circular. We define the buffer as being of type `ADI_DEV_CIRCULAR_BUFFER`. When we set up callbacks, the callbacks again will be live and they are going to occur on the `ADI_DEV_EVENT_SUB_BUFFER_PROCESSED` event, which means every time the sub buffer has been filled we will generate an interrupt request.

Other initializations performed within this thread are to set up a pointer table to sub buffers. What we'll do is set up a simple table with pointers to the different sub buffers that we're going to be working with and just index through that table as we go along. We also are going to use the `VDK_CreateMessage` function to create a pair of messages that are going to be use to re-circulate between the input thread and the process thread.

Under a steady state condition the following events occur. First, we pend on a semaphore, specifically the `ADC_DATA_READY` semaphore. So, when this semaphore is finally posted, we're going to figure out the address of the sub buffer to process next. We will use the `VDK_SetMessagePayload` function to set the message to this particular address. Then we'll use the `VDK_PostMessage` function to post this message to the process thread over the `IBUFF_PTR_CHANNEL`. Finally, we use the `VDK_PendMessage` in order to get the return message back from the process thread. Because we have a couple of messages circulating we generally do not wait at all under the `VDK_PendMessage`. Instead we will get the message right a way, go back to the top of this 'while(1) loop' while we just sort of sit there and wait for the next semaphore to be posted.

Next, we're going to talk about the process thread. The process thread will be pending on messages. What we have set up initially is a message mask. The message mask, as the name implies, is just a mask to identify which channels we're going to be waiting on. So, in the steady state condition we're going to be pending on the channels identified in this message mask. When a message comes in we're going to be using `VDK_GetMessageReceiveInfo` function in order to identify which message channel came in. And then we will switch on that message channel. If a message came in on the `IBUFF_PTR_CHANNEL`, in other words an input buffer pointer. We're going to use the `VDK_GetMessagePayload` function in order to get this input buffer pointer information. Then we will check to see if the output buffer pointer has already arrived. If it has, we will set a flag to allow us to proceed with processing the buffer, otherwise we're going to update the message mask to indicate that we've already received the input buffer and we'll go back to the `VDK_PendMessage` at the top of the loop to wait for another message. And again, if we did receive both we would have processed the input buffer specified by the `IBUFF` pointer message to the output buffer specified in the `OBUFF` pointer message. Then we are going to use the `VDK_PostMessage` function to send both of those messages back over the `RETURN_BUFF_CHANNEL` to their respective input and output threads and we will also reset the message mask to indicate that we want to look for all three messages again.

Next, is the `OBUFF_PTR_CHANNEL`. If the message came in over that channel we'd do the same thing. We use the `VDK_GetMessagePayload` function to get the output buffer pointer and then we will check to see if we already have the input buffer message. So, otherwise it is the same as the input buffer pointer channel.

If the message that just came in, came in over the control channel, when we `GetMessagePayload`, we're going to get the new volume control information and we'll put that aside. And then we will use the `PostMessage` function in order to post this message back to the volume control thread. So, that's essentially what goes on in the process thread.

Now let's talk about the Volume_CTRL thread. There is going to be some initializations done at the start. We're going to use the VDK_CreateMessage function in order to create the two messages that we are going to be using. We're also going to set the MessagePayload with an initial volume control level setting and post that message to the process thread, just so it is initialized with an initial volume setting.

In a steady state condition what happens is we're going to be pending on an event, specifically, the VOLUME_ADJUST event, which is as we mentioned before is going to be based on the UP_VOLUME and DOWN_VOLUME event bits. Just one of them has to be true. So, once the VOLUME_ADJUST event has occurred the first thing we do is to suspend flag callbacks, which means we're going to disable flag interrupts to prevent other flag events from coming in. Then we're going to go to sleep with the VDK_Sleep function for a certain amount of ticks and the purpose for this is to allow some time for key switch debounce. We're going to use the VDK_GetEventBitValue function to test the kUP_VOLUME event bit to see if it is true. If it is, we're going to bump the volume up by one notch. Then we use the VDK_ClearEventBit function in order to clear the kUP_VOLUME event bit. If we didn't get an UP_VOLUME event, then we must have gotten a DOWN_VOLUME event bit; so, again we check to make sure that is true. If it is we'll decrease the volume and clear that event bit. Then we calculate what the new volume control setting is supposed to be based on the UP_VOLUME, DOWN_VOLUME bit just occurring. Then we will use the SetMessagePayload function in order to set the volume level in the volume control message. Then we will post that message using the VDK_PostMessage function to the process thread over the control channel. And again, as we have done before we will use the PendMessage function to wait for a return message over the RETURN_BUFF_CHANNEL and then once we get that we will resume callbacks and flags again with the adi_flag_ResumeCallbacks function. So, this will re-enable key switch interrupts. And then we go back to the top of the loop where we pend on another VOLUME_ADJUST event.

Finally, we have the output thread. So, what we do here is we open up the AD1854 device driver. Again we create a circular buffer using the `ADI_DEV_CIRCULAR_BUFFER` structure and we likewise set up live callbacks every time a sub buffer has been processed. Other initializations that we're performing, we also set up a pointer table to point to the four sub buffers in our circular buffer and we just cycle through those every time we get a callback. We also create a pair of messages that are going to be used to provide the address information for the sub buffers to the process thread and we're going to use the `VDK_PostMessage` to the process thread to give it a first buffer address to work with. Under steady state condition we're going to use the `VDK_PendSemaphore` again to pend on the `DAC_DATA_SENT` semaphore. So, whenever another buffer has been sent out to the D/A converter, the callback function will post that semaphore. When that semaphore has been posted, we're going to then calculate what the address is of the next sub buffer that needs to be filled, use the `SetMessagePayload` to place this address into the message and then finally (use) `VDK_PostMessage` to post this message to the process thread over the `OBUFF_PTR_CHANNEL`. As we've done before, we will pend on the return message from the process thread on the `RETURN_BUFF_CHANNEL`.

Chapter 4: CALLBACKS

So, we're going to talk about callbacks next.

Subchapter 4a: Callback Overview

This is the callback as how it is used with System Services. A callback function is called with three parameters. We have a void*pHandle, a u32arg and a void* pArg. pHandle is what's referred to as a client handle. It's user defined and, in many stand alone System Services applications, can be used to identify who generated the callback request. Did it come from a flag, from one of the drivers? Now in this particular application pHandle is going to be use to pass and identifier for a particular semaphore event that needs to be dealt with in that callback function. u32Arg contains the reason for why the callback function was made in the first place. So, what we do is we switch on u32Arg to determine what type of event occurred. Now, in this simple application the only event that we are really interested in is the fact that the buffer has been successfully completed. You might be also interested in say error results or things of that nature, that they can also add to your code. pArg is not used in this particular demonstration, but uses for it in other applications could be pointers to address buffers that need to be processed, for instance.

In our application every thread that sets up a callback or deals with a callback will have the code for that callback function local to the thread itself. This will end up making the code a lot more modular and allows you to extend the functionality by adding additional threads with their related callback functions at a later point in time.

Subchapter 4b: Input Thread Callback

So, what we're looking at here is the input thread callback function, AD1871_callback, which is going to be called whenever an AD1871 device driver callback event occurs. So again, there are those three parameters that are passed. We are switching on the u32Arg and the only event that we are interested in is the ADI_DEV_EVENT_SUB-

`BUFFER_PROCESSED` . That's the only one that we're checking. If this occurs we're going to use the function call `VDK_C_ISR_PostSemaphore` in order to post a semaphore. What we're doing is we're using `pHandle` in this case to identify which semaphore we want to post. So, when we register the callback we would have passed this identifier. So, this identifier is just a type `VDK_SemaphoreID` . And then we just break when that occurs. For any other event that occurs we just simply break. So, the idea is that the callback function is short and sweet. It's handled live. We go in, we set semaphore, we go out again.

Here we are registering the semaphore as a callback parameter when we open up the device driver. So we use the `adi_dev_open` this is in the input thread when we open up the AD1871 device driver, So again there's the handle for the manager, the entry point for device driver. The fourth parameter is where we have the client handle. In this case we're passing a `kADC_DATA_READY`, which is the VDK identifier for the `ADC_DATA_READY` semaphore. Then going along second from the end is where we would normally pass the handle for the deferred callback manager. As we indicated before we're not using deferred callbacks in this application. The callbacks will be handled live so we put a null in there to make that happen. Then this is the function that is going to be called as a callback function the **AD1871_callback** that we saw on the previous page.

Chapter 5: DEMONSTRATION

Now, what we're going to do is move to a demonstration. So, what we are going to do is demonstrate this code, show how it all fits together, and build it and run it on a BF537 EZ-KIT Lite.

So, we have our VDK project open here, which is just called `SSL_DD_VDK_BOLD`. We're connected to a BF537 EZ-KIT Lite via the debug agent. One thing that I wanted to do is just to show you the files here. We've got our source files. You probably recognize some of these names `input.c`, that's the source file for the input thread. If we take a look at that we'll see the `run` function. We see our AD1871 device driver being opened. There is the `null` which indicates that it is a live callback, and so on. So, that's where the source files are. We've got our main function that we talked about before as well. Again we've got our VDK being initialized. We're initializing the System Services calls with the `adi_ssl_init` again just posting an error, a system error in the event there was a problem, and finally causing VDK to run in the end. This is the stuff that we saw earlier.

The other thing that I want to point out, I guess, was here. This is something you probably would have seen in the Device Drivers module, but the System Services Library includes all the device drivers for the on chip peripherals but for any off chip peripherals, we must include the source files in our project that's why we're showing the AD1854 and the AD1871 source files being included here. There's our `flag_init` function we saw before. `MY_SSL_INIT`, I just want to take a quick look at this. We mentioned about customizing System Services for a custom target. So, this file actually started off as `ADI_SSL_INIT.C`. What I did was I... here is the part where we set up the SDRAM, I mentioned that before. This is that command value pair table that I mentioned, `my_custom_target_ext_mem`. So, this is where it is located. We also have the same table set up for the power. Again remember the processor variant being 600 MHz, clock-in frequency being 25. That is all located in this file here, which is called in this case it is `MY_SSL_INIT.C`. But it started off as I said the `ADI_SSL_INIT.C`.

Let's just take a quick look at some of these header files. Again, all the threads, the input threads, output thread, process thread all have their own header files. `Flag_init` – this is one that was created for the flag initialization. All I've done here I've just defined the `PB_UP` as the ADI flag `PF2`. `PB_DOWN` as the ADI flag `PF3` just to make it easier to identify the purpose for a particular flag pin. `MY_SSL_INIT.h`, this is the one where we reserve whatever services are required like how many secondary handlers, DMA channels and so on and so forth. So, that's found in this particular header file. Again this one started off as the `ADI_SSL_INIT.h`. The other thing was the `SSL_VDK.h`, this is something that I included just to define some parameters for our demonstration application. For instance, `sub_buff_size 128`, `sub_buff_num 4` this is just saying that we're going to be creating four sub buffers with 128 words in each. Then I also have the `#defines` for these message channels, the `CONTROL_CHANNEL`, `IBUFF_PTR_CHANNEL`, `OBUFF_PTR_CHANNEL`, which we've seen from the earlier screen captures. These tie into these particular VDK message channels. So, channel 1 for control, channel 7 for IBUFF pointer and so on. Two messages are being created any time we create a message which occurs in the input thread, the output thread, and the volume control thread. So, that's trying to put things in perspective showing you where the files are and tie it back to some of the information we saw on the slides.

If we go over to the kernel tab, this is where we get some additional information on the VDK setup itself. For instance, the thread types. There's our different threads that we saw earlier. Again the one thing that we wanted to make sure that we do is to set the messaging flag to true to enable messaging between the threads. And again the boot threads ... there are the boot threads defined there. 'IN' is an instance of the input thread type. You can also do the same thing check out the semaphores and the events and event bits.

Let's go on then to the demo. Let's build this, so we're going to hit the rebuild all button. It's going to go through and rebuild all the files. We've also enabled the elimination function in the Linker. As you see, it was able to eliminate about 27K of unused code.

There we go. It loaded on to the target and we're all set. It stopped at main where the first function is initialize the VDK. So, what we'll do is we'll just go on from here and just run this. There's the run button and we have some audio running in the background. Now I've selected two pushbuttons. Here's the volume up and volume down buttons. So, PB1 is the UP_VOLUME. PB2 is the DOWN_VOLUME. So, I can increase the volume by pressing the PB1 button ... and decrease the volume by pressing the PB2 button. So again, there is a callback for every time the button is pressed. I'm just going to stop the application here. Again just a reminder that when you when you are debugging VDK applications I just want to draw your attention to a couple of the windows that they have available to assist with VDK debug. One is called the VDK Status Window, which is shown there. Again we can just expand that and you have some information on the threads, which threads are present, which ones are running, what they are pending on, for instance, we have kInput block on a semaphore. If we were to expand further, we can get some additional information stack size, how many times it was run. I mentioned about, if there was an error created, we can see that if an error occurred within a thread, you can see that here as the last error type. Here there was no error but if there was it would be posted here and the result code would be appearing here under last error value.

So the status window is one, the other debug window we have available is the history window. So, here is our VDK state history. By following the green, we can see which thread is active at any given point in time. So here we have all four threads, we can also throw up a legend, just a color coded legend, to let us know, for instance, these arrows indicate events that cause a thread status to change. So, for instance, a message is being posted or maybe a thread is pending on a message, that's being indicated here. We could also zoom in just by dragging the cursor here and we can zoom in further. So, there is a detailed look at what's going on. We could say right click on here and introduce the data cursor and we can step along and just see, for instance, at the bottom messages posted at this particular tick number. There is the message ID and so on. So you basically can do some very detailed debug of your VDK application. And this concludes the demonstration.

Chapter 6: SUMMARY

Subchapter 6a: Summary

And to summarize, in this module we have demonstrated an approach to using System Services Library and Device Drivers with VDK when developing multi-threaded applications for Blackfin. It is recommended that driver callbacks are handled live and only then use the post semaphores or set/clear event bits within those callbacks functions. Callbacks can also be associated with their corresponding thread code in order to keep the overall code modular. If you are using messaging, consider re-circulating the messages in order to improve the reliability of your code as well as to improve the efficiency. Finally, we have also shown that configuring System Services for a custom target only involves the customization of the initialization for two managers, specifically the EBIU and the Power Manager.

Subchapter 6b: Resources

Thank you for watching this module on Using System Services Library and Device Drivers with VDK for Blackfin processors. For additional information, you can contact the Analog Devices website at www.analog.com/blackfin. Here you are going to find additional information on the Blackfin processors including data sheets, code examples, VisualDSP tools, knowledge base, frequently asked questions, and so much more. For additional information on training, you can contact the Kaztek Systems website at www.kaztek.com or email at info@kaztek.com. If you have any specific questions you can click the ask a question button on your screen. Thank you very much.