Presentation Title: Blackfin® System Services

Presenter Name: David Lannigan

Chapter 1: Introduction

Subchapter 1a: Agenda

Hi my name is David Lannigan. I work in the platform tools group at Analog Devices. And today I'll be talking about System Services for the Blackfin family of processors.

I'm going to cover today what the System Services are, what the benefits of using the System Services are to users, and for each of the services that we have, I'm going to be going through some high level functionality of each of the services, and a quick over view of the API into each service. Intermixed throughout the presentation I have examples that I'm going to show you the System Services doing power, DMA, flags and callbacks.

Chapter 2: System Services

Subchapter 2a: Overview

So what are the System Services? Really they're nothing more than a software library that provides functionality that's common to embedded systems. It provides simple efficient access into the hardware subsystems of Blackfin, specifically the PLL, DMA, interrupt controllers, flags, timers and so forth. We also have a service called Deferred Callbacks that allows us to improve the interrupt performance of embedded systems. The library is callable from both C and assembly. When in assembly the only restriction is that you must adhere to the C run-time calling conventions, but we make no restrictions on what the user code has to be, whether it's C or assembly. The API's are common across all of our Blackfin processors, specifically the single core devices, the ADSP-BF531, BF532, BF533, BF534, BF536 and BF537, and our dual core device, the BF561. The services are used by applications, device drivers, and they really are utilities that can be leveraged in any system. We operate in the stand alone environment without any RTOS, and we also operate in a VDK environment. If you're using the VDK real time operating system, our System Services are supported in that system and the API stays the same.

Subchapter 2b: Benefits

I'm often asked "What are the benefits in using the System Services?". The most important one to me is the faster time to market. Why reinvent code when you don't have to? The software in the services is tested and proven, it shortens the learning curves so there's no start up time, you can start using them right away. The modular software, using the services, modules can be integrated much easier and simpler then in the past. The services manage the hardware resources of the part, so putting two pieces of software together that were done by different groups is significantly easier. As I said the API is identical across the Blackfin processors, so whether you're using a single core device or a dual core device, the API stays the same. We have a pretty rich road map of Blackfin processors and more are coming. Because the API's are identical you can move from one processor to another with very little porting, if any, necessary. Also our device driver library is built upon the System Services. So if you want to access the device driver library using the System Services is a given. Full source code is provided so you can step into the code so you understand exactly what it's doing and in that sense it actually acts as a learning tool as well. You can step in and see how the DMA controller works, how to go and manage it, what the registers are and so forth.

Subchapter 2c: Architecture
Here's a block diagram of a typical application. Up at the top we have the user code, optionally there's an RTOS, some systems have an RTOS, and some systems don't. Device drivers are there to manage the inputs and the outputs of the system. And down at the bottom are the services. Again, services are specifically at the bottom of this diagram because we don't require the RTOS, the RTOS is optional. And the drivers leverage them, the applications leverage them, and even the VDK or any other type of RTOS can leverage the System Services.

Today we have eight, we have an EBIU, which basically controls the SDRAM controller on the Blackfin, Dynamic Power Management, Interrupt Control, DMA Manager, Flag Control for the general purpose IO's on Blackfin, Deferred Callbacks, I'll explain what callbacks are and what specifically what a deferred callback is in a little bit. Timer Control for the core, watchdog and general purpose timers, and port control for our BF534, BF536, and BF537 processors. I'll go through each one of these from a very high level. I'll give a very high level summary of each service, show you the API and work through some examples that show how easy it is to use the services.

Chapter 3: Dynamic Power and EBIU

Subchapter 3a: Power Mgmt Overview

Blackfin has a very powerful power management system. By controlling the Phase Locked Loop (PLL) in the internal voltage regulator, various power dissipation and performance profiles can be selected. Power dissipation of Blackfin, or of any processor, is a function of the frequency and the voltage squared. By controlling both the frequencies and the operating voltage we have significant control over the power dissipation profile of the part. Programming the PLL and the voltage regulator though is not quite as straight forward as one would think. There are registers that need to be programmed, interrupts that need to be generated, waiting for the PLL to stabilize, it's actually a fairly complex process, quite error prone as well. The Dynamic Power Management Service provides a single function call to go and change our system and core clock frequencies (SCLK and CLCK), change operating modes, and even change voltage level in the part. With a single function call you can change between the various operating modes of the processor such as full-on, active, sleep, deep sleep and hibernate. You can also change the core and SCLK frequencies. When the core and system clock frequencies are changed the power management service automatically adjusts the voltage level to maximize power savings whenever possible. So let's say you choose a low core CCLK and SCLK settings, we'll automatically go and lower the voltage level to the lowest level possible to run those frequencies at. Conversely if you want to maximize the performance of the part and raise the CCLK and SCLK frequencies will automatically go in and adjust the voltage level upwards. Conversely if you wanted to control voltage and then set our clock frequencies, you can set the voltage to a specific level, say 0.9 volts, and then with a single function call maximize the CCLK and SCLK frequencies to the maximum values that can be run at 0.9 volts. In addition when ever we go and change any of the clock frequencies we automatically make the calls to the EBIU Service to set our SDRAM settings accordingly.

Subchapter 3b: EBIU Overview

So let me talk about the EBIU Service. Think of the EBIU Service as a controller for the SDRAM controller on Blackfin. It contains logic to calculate new values such as when we change our SCLK frequencies, we automatically calculate the refresh settings for the SDRAM. It works in concert with Power Management Service, so from the user perspective all the user has to do is go and change CCLK and SCLK and we automatically make the effective changes into the SDRAM settings. The take away for EBIU and Power Management is that an application should first initialize the EBIU Service then initialize the power management service, and then simply call the functions and the Power Management Service as needed, such as setting the frequencies, setting the maximum frequencies for a given voltage, or changing the operating modes of the processor.

Subchapter 3c: Dynamic Power API
Here's a summary of the API for the Dynamic Power Management. Now I don't want to linger too long on the API, but one thing to look at right up here at the top are the initialization and termination functions. Every service has an initialization and termination function. Before the service can be used the init function must be called. When you're finished using the service you call the terminate function. Often in embedded systems the system runs in an infinite loop, never having a need to call the terminate function. But we also have functions here in power management to go and manipulate the voltage and frequencies, and also to set and get the power modes.

Subchapter 3d: EBIU API
In the EBIU API, the most important ones again are the initialization and termination functions. When working in concert with Power Management the only thing that the application really needs to do is initialize the service, once it's initialized everything else is automatic.

Subchapter 3e: Example
What I'd like to do is demonstrate this and show an example of running on the Blackfin BF537 EZ-KIT Lite. I'm going to show a main program that basically just initializes the services and then go and change various frequencies, the SCLK and CCLK and toggles between the different operating modes; full-on, active and so forth. We'll see that it makes these calls into the power management service, power management in turn controls the PLL and voltage regulator, and also makes the calls into the EBIU service to adjust the SDRAM settings automatically. What I'm going to do is I'm going to bring up VisualDSP®, and here I've got the example. What I would like to do is just walk you through the example line by line. The first thing we're going to do is initialize the services. Once they're initialized we can go and use any API functions in the service. I'm just going to step over that. Over here on the right hand side we have local variables for the frequency of CCLK, the frequency of SCLK and the frequency of the voltage- controlled oscillator. The key ones here are CCLK and SCLK. I'm about to make a call into the Dynamic Power Management Service called GetFrequency, passing in locations where I want it to go and store the CCLK and SCLK frequencies. That's going to appear right up here in the right hand side of the stream. So when I execute it, it tells me we're running at 250 megahertz on our CCLK, 50 megahertz on our SCLK, and the voltage control oscillator is running at 250 megahertz as well. When I execute this next function we're commanding the Power Management service to turn our CCLK value up to 600 megahertz, or SCLK value up to 120 megahertz. So when I do a step it's gone and effected those changes and now what I'm going to do is just interrogate it and make sure it's gone and made those changes to what we had said. The CCLK is now running at 600

megahertz, the SCLK at 120. Next we're going to set our CCLK frequency to 500 megahertz and tell the Power Management Service to maximize SCLK. I want SCLK to run at the fastest frequency it can run at provided the core is running at 500 megahertz. Remember, we can't go and set each frequency individually, as there is a relationship between the two. When we set CCLK to a specific value, we only have certain values that we can get SCLK at. So here what we're telling the Power Management Service is maximize SCLK, but I really want CCLK to run at 500 megahertz. So I execute the function, and interrogate the service to find out what the frequencies are. It set our CCLK to 500 and it maximized SCLK out at 125. Depending upon the clock-in values and the crystals that are used will actually help determine what the specific combinations are. See the hardware reference manual for information on that.

Next we're going to go and change our voltage. We're going to command the power management service to turn the voltage regulator down to 0.85 volts and run at the maximum frequencies we can at 0.85 volts. So I'm going to call the SetMaxFrequency for voltage function. And now what I'm going to do is interrogate again to see what the CCLK and SCLK values are. At 0.85 volts our maximum CCLK value is 250 megahertz, which is what we're running at now, and the SCLK as been set to 83 megahertz.

Now to demonstrate changing the operating modes of the processor to go from full-on to active, to sleep and what not. What I'm going to do here is we're running at full-on right now, we're going to go and execute the function that's going to put us into the active mode. So now I've gone and affected that change into the active mode, I'm going to interrogate it just to prove that I've actually gone and done that. My cursor over power mode and you'll see we're now running in the active mode. Now I'm going to query the frequencies, the CCLK is running at 25 megahertz, and SCLK is running at 25 megahertz. Remember in active mode the PLL is by-passed so we're really running at our clock-in frequencies. Now that we're active let's say we want to go back on and do some high performance processing, so let's go command the processor back to the full-on mode, turning on the PLL, if I interrogate it you should see that we're back to full-on. Interrogate our frequencies; we're back up to 250 megahertz on the CCLK and 83.3 on the SCLK. Now I'm going to command it to run both CCLK and SCLK as fast as we can, maximizing the performance of the processor, interrogating, again 600 on the CCLK and 120 on the SCLK. So a very simple mechanism to go and change the operating frequencies of the part and to change the voltage level of the part.

Chapter 4: Interrupt Manager

Subchapter 4a: Overview

Let's continue on with the presentation. Here's the slide that we were on, I'll go to the next slide. We'll talk about the Interrupt Manager Service. The Interrupt Manager Service is kind of central to the other services. Many of the other services make calls into the Interrupt Manager, because interrupts are a key component of embedded programming. The service itself maps pretty much directly to the core event controller and the system interrupt controller on the Blackfin. Remember the core event controller operates like any other general purpose type processor with a vector table, and when a vector or rather an interrupt is asserted, we vector to the address in the table. The system interrupt controller on Blackfin is there because we have a very rich set of peripherals. We have a lot more peripherals then we have interrupt vector groups. The system interrupt controller is kind of like a gate, it decides which interrupts are allowed to be passed back to the core and in turn, services. The Interrupt Manager manipulates both the core event controller and system interrupt controller depending upon what the user wants. We can hook and unhook interrupt handlers into the various interrupt vector groups. We also support handler chaining so if we have say a single interrupt vector group, we can install multiple interrupt handlers for that one vector group. Take the case of errors. Errors are something that we expect relatively infrequently. We don't really want to waste a whole interrupt vector group for each potential error source in the processor, so we may gang up multiple errors on to the same interrupt vector group; handler chaining facilitates that. So let's say we have a PPI device driver, SPI device driver, and a UART device driver all running in the system concurrently. We could map all of the errors from those peripherals to a single interrupt vector group and allow each device driver to hook its own handler into the chain.

Next is the system interrupt controller. As I said, it operates as a kind of gate, allowing the peripheral interrupts to be passed to the core event controller. The Interrupt Manager Service provides a C function call interface to allow the user to specify which interrupts it wants to pass from the system interrupt controller into the core event controller. You can change the mappings of the peripherals to the various interrupt vector groups. So if you want a peripheral interrupt to be a high priority you can change it's mapping to a high priority interrupt vector group. You can enable and disable each interrupt on whether or not it's even passed the core event controller, and also enable or disable whether the interrupts from the peripherals wake up the processor from one of its sleep modes. We also have utility functions that protect critical regions of code. Let's say you have a piece of data that whenever you modify it you want to be sure that no other software in the application modifies that while you're going and changing it's values. We have an enter critical region function and exit critical region function, and you would bracket the sensitive code with those calls. We also have an interrupt mask register control set of functions that allow you to set and clear bits in the interrupt mask register. We've isolated these into different functions, and isolated them into the interrupt manager service so that we can work in the various operating environments such as the VDK, stand alone system, or really any other type of RTOS. RTOS's don't like people changing the interrupt mask register without their knowledge. So by isolating it into a function we can actually notify the operating system when ever we go and change IMASK register values. Also critical region protection, various operating systems have different mechanisms that they use to protect critical regions of code. By isolating it into separate functions, you can port from one operating system to another very simply without affecting the application. Some RTOS's disable scheduling to protect a critical region, other disable interrupts. We really rely on what the operating system provides and that our calls to the services that are provided by the RTOS.

Subchapter 4b: API
Here's the API into the Interrupt Manager. Again we have an initialization and terminate function for this service. We have core event controller functions where can hook and unhook handlers from the interrupt chain. We have system interrupt controllers where we can change the mapping of peripheral interrupts to Interrupt Vector Groups (IVG). We can set the wake up conditions for the processor, and we can test whether or not a peripheral interrupt is even asserting an interrupt. And then down at the bottom we have the utility functions I talked about; the enter and exit critical regions functions, and the setting and clearing of IMASK bits.

Subchapter 4c: Example
Now what I'd like to do is show a very simple example using the Interrupt Manager and our real time clock. In our main program what we're going to do is program the real time clock to generate an interrupt every minute. Every minute that transpires another interrupt will be generated. We'll call the Interrupt Manager to tell the Interrupt Manager to allow that interrupt from the real time clock to be passed to the core event controller, and we're going to hook our interrupt handler into the core event controller. So a very simple example here that just demonstrates interrupt handling using the Interrupt Service. Again, let me go into VisualDSP, I'm going to close down the project that we were working on for power, open the Interrupt example and go and build it. Here we are at main. We're just going to talk through this step by step but first I'll just give a quick over view. First off what we're going to do is clear the prescaler so that the clock runs at or set the prescaler excuse me, so that the clock runs at the correct frequency. We're going to clear any pending interrupts, and we're going to enable the real time clock to generate an interrupt whenever the minute register changes, so every minute we'll generate an interrupt from the real time clock. We're going to interrogate the service to find out what interrupt vector group the real time clock is mapped to. If we wanted to we could actually go and change

that mapping, but I'm just going to use the default mappings. Here I'm going to call the hook function to hook our handler in. And then we're going to go in and enable the interrupts. Through those functions, those simple four functions, we can actually go and enable the interrupts and go and process them. Let me just skip down to our interrupt handler. First off what we're going to do is make sure that the interrupt is for us. Remember we talked about interrupt handler changing. One of the things that you need to be aware of when using chaining is that you may be called when an interrupt is not really for you. You have to go in and interrogate the device that you're supporting and see if the interrupt is for you because it may be for someone else in the interrupt chain. So we're going to check to be sure the interrupt is for us, we're going to clear it and then we're just going tell the Interrupt Service yes we went and processed that interrupt, that interrupt was indeed for us, so you don't need to call anyone else if there was anyone else in the interrupt chain.

Let me go back up to top, and again I'm just going to skip over the initialization. Now I'm going to tell the real time clock, I'm going to enable the prescaler so that it operates at the correct frequency of changing every second and updating it's minute every 60 seconds. I'm now going to clear any pending interrupts because we want to be sure that there's nothing else already set there. And then we're going to tell it to generate an interrupt every minute. Now I'm going to query the Interrupt Service to find out which interrupt vector group the real time clock is mapped to. I step over this function, I see the IVG or the Interrupt Vector Group that the clock is mapped to is 8. Now what I want to do is I want to hook my handler into IVG8. This is the call to go and do that. Here's the interrupt vector group that we want to go and hook into, here's the address of our handler, here's something we call the client handle. This is just a value that's passed to the handler whenever it's invoked so that we know who is actually going and calling us. This last parameter says whether or not we want interrupt nesting enabled. Interrupt nesting is a feature that allows higher priority interrupts to interrupt a lower priority interrupt. That can be true or false in this example we're just making it false. We're going to step over that, so now we've hooked our handler into the chain, now all I need to do is tell the system interrupt controller to allow that real time clock interrupt to be passed to the core event controller. Before I do that I'm going to insert a break point in our interrupt handler so we'll stop the processor when that interrupt goes off. I'm going to press F5, which is run and we've immediately hit our handler. The minute interrupt actually expired while I was stepping through those other functions. What I'm going to do now is I'm going to check to make sure it's a real time clock interrupt, it is, now I'm going to clear that interrupt out of the real time clock. I'm going to just clear that interrupt pending, and return to the Interrupt Manager that we've indeed processed that interrupt. I'm just going to hit run here and now our processor is running. Now it's just running, it's sitting in the loop not doing much of anything, it's waiting for that next interrupt to occur. Hopefully within the next 60 seconds or so we'll see that interrupt go off again and we should hit that break point one more time. That was about 60 seconds since the last time. Again check to be sure that's our interrupt, indeed it is, I'm going to go clear the interrupt and return that we've gone and processed it. That's it. A very simple overview on how to use the Interrupt Service; how to tell a peripheral to go and generate an interrupt, how to pass it to the system interrupt controller, how to hook a handler into the core event controller. Again, very simple, very easy to use. We did it in 3 function calls, and we hooked the handler in, enabled it to be passed from the system interrupt controller to the core event controller. Very simple process.

Chapter 5: Deferred Callbacks

Subchapter 5a: Overview

Let me go back to our presentation. But before I discuss our Deferred Callback Service, I think it's important to understand what a callback is. A callback is really nothing more than a function call that's made outside the normal flow of execution, such as in response to an asynchronous event. It's nothing more than a regular C callable function that takes some action based upon the event that occurred. There are two types of callbacks, live callbacks and deferred callbacks. Live callbacks are made immediately upon receipt of the interrupt, so a live callback executes typically at hardware interrupt time. A deferred callback executes at software interrupt time. The difference between the two is actually quite important. When executing a callback live, meaning at hardware interrupt time, lower priority interrupts are disabled until that callback function actually returns. In a deferred callback, lower priority interrupts are enabled because the callback service executes the callback at a lower priority. The Deferred Callback Manager provides a mechanism to defer callbacks from the high priority hardware interrupt time to a lower priority software interrupt time. When an event occurs typically what happens is that within the hardware service, we queue the callback, in other words make a note saying that we're going to make the callback later on, and we exit the high priority interrupt service routine. At a later point in time that software interrupt is serviced and then we go and make the callback at that point and time. The Deferred Callback Manager allows the user to create multiple callback queues. Each queue can operate at a different interrupt vector group level, which means inherently each queue operates at a different priority. Within each queue you can also specify relative prioritization for callbacks within that single queue. Let's say you have one callback queue that executes at say IVG14, you can have an urgent callback that will be called back first whenever that callback queue runs and have lower priority callbacks that also execute at IVG14, but just after that urgent one. We try and leverage anything that's available in the operating system so in a stand alone case we execute the callbacks typically at IVG14 right before normal user code is run. In a VDK based system we make use of the software interrupt thread to execute our callback.

Subchapter 5b: API

The API into the callback service is pretty simple. Again there's an initialization and terminate function, and then there are functions to go and open a queue, close a queue, control a queue such as mapping it to a specific interrupt vector group, and then here is a call that's used to actually post a callback to the queue. And then remove after posting one you want to remove it from the queue it can be removed. I have an example coming up of callbacks that we use with the Flag Service, so I'll defer that example until we talk about the Flag Service.

Chapter 6: DMA Manager

Subchapter 6a: Overview

Another service that we have is the DMA Manager. Blackfin has a very powerful DMA controller. It supports both peripheral DMA and memory DMA. Peripheral DMA is used primarily in device drivers as device drivers take data within the Blackfin and send it out the device, or conversely take data from the device and store it into Blackfin memory. The DMA Manager controls and schedules both peripheral DMA and memory DMA. Device drivers make the appropriate calls into the DMA Manager when they're moving DMA data. Also applications can use the DMA Manager to move data from one memory space to another memory space, or from one location to another memory location with memory DMA. We also allow control of the various DMA channel mappings. Each DMA channel has an inherent priority associated with it. And we can change the mappings from one peripheral to another peripheral. In your system you may want the PPI DMA channel to be higher the UART priority channel, we can go in and affect those mappings for you. We also provide a mechanism for traffic control which manipulates the arbitration logic between core access to the DMA busses or the memory busses, and the DMA controller's access into the memory bus. The DMA Manager supports all the major modes of the controller such as descriptor chaining for both large and small descriptors, auto-buffering, which we call circular buffers in the DMA Manager, and then we also have the higher level memory copy functions that allow DMA transfers rather than core access transfers to move data from location in memory to another. We can leverage the one dimensional and two dimensional transfer mechanisms of the DMA controller so we can do both linear and XY type data transfers. And we have callbacks on completion. This is an example of where you would use a callback, let's say you want to schedule a memory transfer and you want to be notified when that transfer is complete. The callback mechanism allows that notification to happen, and that notification can be either live or deferred, depending upon what the user wants.

Subchapter 6b: API
Here's the API of the DMA Manager. Again, like all the services we have an initialization and terminate function. We have a facility to manage each individual channel using open, close, control, queuing descriptors on to a channel, and then higher level memory control functions where you operate on a stream. A memory stream allows you to go and move data from one memory space or one location in memory to another memory space in another location in memory. And then the channel mappings that I talked about to change the mappings of which channels are mapped to which peripheral.

Subchapter 6c: Example
Let's work through a memory DMA example. The example I'm going to show you has a main program that's going to copy memory from the source location to a destination location. The way it's going to do that is it's going to open up the memory stream and command the DMA Manager to go and move that data from this location in memory, to that location in memory. The DMA Manager is going to go and affect that move, programming the DMA controller on Blackfin as necessary to go in and do the transfer. Then the DMA Manager is going to call us back and let us know when that transfer has been completed. In the callback function we're going to schedule another memory transfer. This is a typical scenario that's used in such things as video processing. In video processing we're typically dealing with large amounts of data, that data is stored in SDRAM and then brought into level 1 memory where it's processed, say compressed, and then gone and sent to say a recording device or something. To go and bring the data in from external memory or from SDRAM into internal memory, typically we use the DMA memcopy function.

Let's bring up VisualDSP and we'll walk through that example. I'm just going to open the project and then I'm going to build it. What I'm going to do over here on the right hand side is I'm going to display some memory. And the memory we're going to display is the source and destination. Let's make this window bigger so we can see. Up here in the top corner is our source memory and down here in the bottom is our destination memory. What we're going to do is we're going to use the memcopy function to go and transfer the contents of this memory to this location down here. So let's step through the code and see what happens. The first thing I'm going to do is initialize the System Services and again now we can use all the functions in the services APIs. Now what I'm going to do is I'm just going to create some dummy values in this source data right up here. We're just going to count from 0 to 32 I believe, and just prepare that memory so that we'll now know what those values are when we copy them into the destination location. I'm just going to run down and skip over that, there we go. Here you can see we've populated this location of memory with some values. I'm just going to slide this over so we can see it a little bit better. Now we're going to make a call to the DMA Manager and tell the DMA Manager we want to open up a memory stream. In a memory stream, there are two memory streams on the BF531 and BF532 and BF533 devices in addition to the BF534, BF536 and BF537. I think we have 4 memory streams on the BF561, the dual core part. What we're going to do is we're going to tell the memory, the DMA Manager to open the stream so that we can go and use it. I'm going to step over this function, actually I'll explain the parameters before I go on. The first parameter is the handle to the DMA Service. When we initialize the DMA Manager we have a handle to that service and that's how we identify that service. I'm going to open that memory stream using that service. This is a stream ID that we want to go and open which is memDMA index 0. This next value is a client handle, so whenever we get called back we're going to be called back with that client handle. This is a useful thing that applications can go and use whatever value they want for the client handle, it doesn't matter to a DMA Service what value that is so long as it's presumably some value that's of significance to the application. It can be any value what so ever. You'll see in the callback function that gets passed as a parameter back to the callback function. When we open the memory stream the DMA Manager gives us a handle to that stream. That handle identifies that stream to the DMA Service, so whenever we go and reference that memory stream with the service, we have to pass that handle back to the DMA Manager so that he knows what we're talking about.

We're going to be using live callbacks here for the sake of simplicity, so I'm going pass in a NULL for the callback handle service. I'm just going to step over this function, and now all we've done really is we've just talked to the DMA Manager and said we want to go and use this service for ourselves, we've opened that and the DMA Manager has allocated that memory stream to ourselves.

Here's the function to go and affect the copy using DMA. Here's the handle to the memory stream that the DMA Manager gave us. Here's the destination location, the source location, the width of each element, byte wise 8 bit values, so our width is 1 byte wide. Data size is set to 32, which is the size of the source array and this destination array. Once I make this call the DMA controller is going to start to go and move that data. I'm just going to sit in an infinite loop after we command it to go and move the data. The DMA Manager is going call us back when it's finished that transfer. What I'm going to do is put a breakpoint in our callback function so that after the DMA controller has gone and moved the data, he's going to call us back and say okay I'm done with that. Now I'm just going to hit F5, which is run, and we've hit our callback function, and if we look over our memory window we see the destination now contains the same values as the source. It's gone and copied all that data that was in the source array to the destination array. And it did that behind the scenes using DMA, so that actually happened in parallel with the core code executing. Now we're in our callback function, the DMA controller has completed the move and is letting us know I'm done with that move so that we can do something else. Remember that the value that we passed in that 0x12345678 is the client handle, we see it right there as being passed back to us as a parameter in the callback. That's useful if we have multiple callback functions we can use different client handles for each callback function so that we know where this event is coming from. We're going to now test the event. All callbacks take 3 parameters, and the first parameter is that client handle so that's a value that's of significance to the client or the application. The next parameter is the event ID for the event that actually occurred. In this case here it's 0x30001 which is the event, the DMA event that a descriptor has been processed. And the last value is based upon what the specific event is. Depending upon what the event is, the last parameter kind of changes context based on that event. In device drivers for example in a buffer completion event, the last parameter may point to the buffer that has completed. In this case here with DMA, when a DMA descriptor has completed, the value that's passed back is a pointer to the destination address. If I put my cursor over here you'll see the address is 0xFF801144 and sure enough that's the address of the destination. That's a nice easy way to tell one what the event is that expired or occurred and specifically what the location of memory was that was affected. I just have a little test to ensure that it is indeed the DMA event that the memory has been moved.

And now in my callback function I can actually schedule another copy. And this is really a very handy thing to do, it demonstrates some of the capabilities of a callback function in that we're executing at interrupt time but we can actually go and affect another transfer at interrupt time. Take our example of processing video data, this is a nice way of being notified that a chunk of memory is now available for us to go and process. And we can go and schedule another chunk of memory to be moved while we're processing that original one. I'm going to schedule that same transfer one more time and just to prove that we're actually doing something, I'm going to change some of the values in our destination address. And after we do this and then copy yet again, the DMA Service, it's going to copy that source data back into our destination area. Let me hit F5, and there it is, it's gone and copied that same data in again. We have this loop that we've created where we're constantly moving buffers from in this case here from the source location to the destination location. A very simple mechanism to go and show how memory can be copied by using DMA. Very simply, we opened the stream and then gave it a description of where the memory was and how we wanted it, what the size was and to call us back when it was done. So very simple, two functions and we can go and use the DMA controller. Again simplicity is one of the key features of the System Services. The services are designed to make the application's life that much easier.

Chapter 7: Flag Control

Subchapter 7a: Overview

Let me switch back to our presentation and move to the next slide. Blackfin provides what we call general purpose inputs and outputs or general purpose programmable flags. The flags are very useful in embedded control systems. The flags are just really pins that we can configure as inputs or outputs,. We can set values to those pins so that we can trigger external devices or sense activity from external devices. The Flag Control Service provides a very simple interface into manipulating those pins, configuring them, sensing the values on the pins, setting the direction of a flag as being an input flag or an output flag. Setting the value to a 1, logical 1 and logical 0, or toggle the values. We also have a callback capability in the Flag Control Service. The callback function is invoked upon a trigger condition. It's up to the user to describe or to specify what the trigger condition is. The conditions that we support are level sensitive transitions, such as a flag pin being raised high or low, or edge sensitive, which actually detects the rising or falling edge, or you can cause the trigger to occur on either edge, either the rising or the falling edge. Where ever we use callbacks we can do live or deferred callbacks and the flag control service is no exception. We can provide a callback capability running either live or deferred whenever a specific trigger condition occurs on a programmable flag.

Subchapter 7b: API

Here's the API into the service, the familiar initialization and termination functions. The next functions here work on individual flags. You can open a flag, close it, set the direction, set the value, clear the value, toggle, or sense if it's an input flag you can actually sense the value. And then callback facility here where you can install callback on a flag, set the trigger condition, and so forth.

Subchapter 7c: Example

We're going to walk through a Flag Control example. In our main program we're going to configure a flag as an input and install a callback for that flag, when the condition on that flag changes. The Flag Control Service is going to manipulate the programmable flag sub system of the Blackfin. In this case here because we're running on the EZ-KIT Lite, the flag that we're going to control is mapped directly to a push button on the EZ-KIT Lite. So when I put the button on the EZ-KIT Lite, that's going to cause the trigger condition to occur and our callback function should be executed. Let me get into VisualDSP, and close out the project we were working on. Let me open up the flag project. I'm just going to build it. Here's our source file. Again, the first thing we're going to do is initialize the System Services, so I'm just going to step over them. I'm going to open flag PF2. PF2 is mapped to one of the push buttons on the BF537 EZ-KIT Lite. I'm going to open the flag, I'm going to tell the Flag Control Service that I want to configure this flag as an input. And now I want to tell the service to install a callback on this flag. The callback, scroll over so we can see this, so I want to install a callback on flag PF2. This is the interrupt that I want to trigger the callback on, which in this case here is Interrupt A for the flag. We're going to trigger the event that's going to occur that we want to cause a callback to occur on I should say is on rising edge of the flag. When this flag goes from a 0 to a 1 that's going to trigger us to have the callback invoked. The next parameter is true. We could configure this as either true or false, it's the wake up flag to the processor. With the Flag Service you can configure it such that if the processor is sleeping in either sleep or deep sleep modes, when the trigger condition occurs we'll actually wake the processor up out of that mode and put the part back in to the full-on mode.

Next up is just a client handle, again just like in the previous example with callbacks, that client handle is passed into the callback function. Our callbacks are going to be live, if we wanted them deferred we would simply pass in the handle to the deferred callback service, that we wanted the Flag Manager to use. And lastly, the address of our callback function. Let me just before I step over that function call, let me go into the callback function. When this is called we should be passed in the client handle, again that's 0x12345678 that we just talked about. The event will signify that the flag has triggered, and the pointer argument in this case is actually the flag ID that corresponds to that flag that has triggered. Now I'm just going to install a callback and go back to this function here, step over this function, and now I've gone in and hooked the callback. My callback is installed there and I'm just going to spin in the loop waiting for that condition to occur. What I'm going to do is in my callback function I'm just going to insert a breakpoint, and now when I push F5 or run, the processor is running and it's just sitting in that loop, that "while" loop that I just talked about here, doing nothing except waiting for that callback to occur. Then, when I push the button on the EZ-KIT Lite, that trigger condition will occur and our callback function should get evoked. I'm just going to push the button now, and now we've halted the processor at our callback. Sure enough, if we look at the flag ID we'll see that it's PF2, so PF2 was the flag that we configured to generate the callback. And again that flag is mapped to the button on the EZ-KIT Lite, so when that flag transitioned from low to high, which was our trigger condition right here, the callback occurred. A very simple way, in this case here 3 functions to go and configure a flag, or open a flag I should say, configure it as an input and install a callback. Again, ease of use, one of the number one goals of the services.

Chapter 8: Timer Control

Subchapter 8a: Overview

Let's go back to our presentation and move on to the next slide. Timers. Blackfin has 3 types of timers in the processor. We have a core timer, a watchdog timer, and several general purpose timers. The core timer is frequently used kind of like a heart beat monitor or a system tick type timer where it generates an increment every so often at a periodic rate. A watchdog timer is often used to signify some time out event. Sometimes it's used as a dead man switch so that it has to be reset every so often otherwise an event will occur. On Blackfin, that event can be either a reset event where we can reset the part, it could be just a general interrupt, or it could be just any other type of event, it just resets itself without causing any action. General purpose timers, we have several general purpose timers that can be run in either width-cap mode or pulse width modulation (PWM) mode. PWM mode is very useful for things such as motor control, where we facilitate motor control applications by simultaneously enabling or disabling groups of timers at one time. The Timer Control Service provides full access into the features of Blackfin for each of these timer families. In addition we have a callback capability just like callbacks work on the timers just like in any other services, when a timer expires you can configure the timer service to invoke a callback function. That callback function can be either a deferred callback or a live callback, up to the user to decide.

Subchapter 8b: API

Here's the Timer Control API, again initialize and terminate. The functions here to open and close and control each of the different types of timers, core control, watchdog, and general purpose, and a facility to go in and install callbacks on timers and remove callbacks on timers. Again, a very simple, easy to use API.

Chapter 9: Port Control

Subchapter 9a: Overview

The last service is called Port Control. And what Port Control does is it controls the assignment of mux-ed pins on the Blackfin. Blackfins have lots of peripherals and peripherals tend to have lots of pins associated with them. If all the pins were brought out the part we would have a very high pin count part which would mean a relatively high cost, so we use a technique called mux-ing where we would actually mux the signals of different peripherals on to a single pin. In other words a pin can be used as a PPI in one instance, you could also configure that pin to be used as a general purpose I/O or a general purpose flag in other instances. On the BF531, BF532, and BF533 single core devices and the BF561 dual core device, the mux-ing of the pins is handled automatically by the hardware. Newer parts with lots more peripherals such as the BF534, BF536 and BF537 provide a hardware subsystem that controls the mux-ing of the parts, so we have a port control service that in turn controls that hardware mechanism. For the most part operation of the service is largely transparent to applications. Applications typically only need to initialize the service and then at that point the other services actually make the calls into port control to go and manipulate the pins appropriately. Same with device drivers; for example a PPI driver that's being configured for external frame syncs will automatically make the calls into the port control service to enable those pins such as they are used for frame syncs services. Timers the same way. If you take say a PWM timer and configure it to generate a signal out, the timer service automatically calls the port control service to configure that pin as an output pin. Same with flags, when you open a flag and you set the direction on the flag the Port Control Service is automatically invoked by the flag control service to configure that pin as a general purpose flag. Again, largely transparent to the applications; once it's initialized the application typically doesn't need to go back into Port Control.

Subchapter 9b: API

The API just like the others has an initialization and terminate function. And then we have functions to enable the various pins of the peripherals, PPI, SPI, SPORTs, UARTs, CAN, timers and GPIOs. We also have a mechanism to go and sense the mux-ing of the pins in one fell swoop if you will. We can get the profile, in other words get the status of all the pins, save that in a configuration and then in turn go and set the profile. Rather then do them individually through the individual peripherals, you can actually specify a complete set of mux-ing for all the pins on Blackfin you can it and get that profile.

Chapter 10: Conclusion

Subchapter 10a: Locating the services

Where can I find more information about the System Services? Well, the APIs themselves and the include files for all the services, are located in the directory "Blackfin/Include/Services". In a typical default VisualDSP installation, it's in the directory "c:\Program Files\Analog Devices\VisualDSP 4.0\Blackfin\Include\Services". All the ".h" files for all the System Services are contained in there. The APIs are in the ".h" files and you'll see that dot H files map to each of the services. You'll see an adi_dma.h which is the API for the DMA service, adi_flag.h which is the API for the flag service, and so forth.

Source files I mentioned earlier that all our sources are available. They're located in the "lib" directory, or library directory under source, services ("c:\Program Files\Analog Devices\VisualDSP 4.0\Blackfin\lib\src\services"). Again, you'll see adi_flag.c, adi_dma.c, all the sources for each of the services. The libraries themselves for the System Services exist in the same directory as the rest of the run time libraries, which is in "c:\Program Files\Analog Devices\VisualDSP 4.0\Blackfin\lib". Examples, we provide several examples. The examples run on hardware on each of the EZ-KITs. If you look in Blackfin EZ-KITs, and the subdirectories below that such as BF533, BF537 and BF561, in each of those directories is a sub directory called services ("c:\Program Files\Analog Devices\VisualDSP 4.0\EZ-KITs"). You can find examples using the System Services there.

We have a pretty comprehensive manual. The Device Driver and System Services User Manual is available on the Analog Devices website in the technical library section of www.analog.com. In September 2005 we added some new services and updated the user manual so there is an addendum printed in September, that can be found on the Analog Devices FTP site at this address right here "ftp://ftp.analog.com/pub/tools/patches/Blackfin/VDSP++4.0".

Subchapter 10b: Summary

Hopefully today I showed you some of the benefits of the System Services. I've shown you that there's less reinvention. That we provide this code today so that you don't have to invent it yourself. It's a stable software base; it's been tested across the different Blackfin processors that we have. There's less reinvention needed, we write the DMA controller code once for example, it doesn't need to be rewritten several times by each customer. Modular software; software can be more modular and as a result is easier integrated. The services manage the hardware resources of the Blackfin, so because the System Services are there, it's very straight forward to go and take different modules that use the System Services and integrate them together into a single unified application. And portability, as I said earlier, the APIs are identical from one processor to another, so there's no real porting effort that's needed should one want to move say from a current processor to say a next generation Blackfin processor or perhaps an application needs additional horsepower, it needs to move from a single core to a dual core device; the APIs are the same, and the code really doesn't need to be ported, it just needs to be recompiled on the new processor.

Subchapter 10c: Additional Info

For additional information again consult the documentation. It's available on the www.analog.com website under Blackfin Technical Library manuals. I strongly recommend reading the introductory chapter and the first few sections of each of the individual services. If you have any more questions please click the "Ask a Question" button at the bottom of your stream.

I want to thank you very much for taking the time to hear about the System Services. Thanks again.