Presentation Title: Introduction to VDK

Presenter Name: Ken Atwell

Chapter 1: Introduction

Sub-chapter 1a: Overview

Hello I'm Ken Atwell, Product Line Manager for Analog Devices. I'm here today to give you an introduction to the VisualDSP Kernel, better known as the VDK.

In this module we will discuss the VisualDSP Kernel, some of the concepts underlying it and the capabilities it has. If you're going to be listening to this module we do recommend that you have a basic understanding of software development concepts. It also is extremely helpful to have exposure to operating system concepts either through another commercial RTOS that you've used in the past, or perhaps your organization has developed an in-house operating system that you're beginning to outgrow, you're now looking at a VDK as a possible replacement.

The outline for this module is as follows: we'll introduce the operating system choices that are available for Blackfin before introducing the VDK directly. We'll then at some length talk about the capabilities of the VDK and take a look at some of the APIs that it has. I'll then switch to VisualDSP and go live and show you some of the windows that VisualDSP provides you both for creating, editing, and debugging VDK-centric applications. And finally I'll wrap it up with some measurements, both cycle counts and application footprints for VDK applications.

Sub-chapter 1b: Blackfin OS Choices
First it's worth noting there are number of third party operating systems available for the Blackfin processors. This slide here is an incomplete list of what those are. They're available at a number of price points, a lot of different capability levels, lot of different support models. Perhaps you're already using one of these on some other platform today so in this case the migration to Blackfin is probably going to be straightforward because your operating system is already supported on the Blackfin. If you need more information on any of these operating systems, the link at the bottom right of this slide does give you the URL – that's a jump off point to get information on these third parties.

Sub-chapter 1c: Introducing the VDK
That said, if none of these operating systems are a good fit for your application, VisualDSP does come with its own kernel which again we call the VisualDSP Kernel, or the VDK. It is a very small robust kernel that ships with VisualDSP and is an integral part of VisualDSP. It is updated along with VisualDSP so if you move to the next major revision or if you take an update or patch you can be sure that the VDK will mature along with the rest of the product. So it's very easy from a maintenance point of view, it goes lock step with the rest of the tool chain. It's available at no extra cost, be it at purchase time or at run time, there's no licensing fee or per unit fee that you need to pay to use the VDK, it's completely royalty free. It does support all Blackfins that are available, be it today or those in the future. So if you're going to be, if your application demands that you're doing to be moving to different Blackfin family members over time. VDK is something to think about because the API is consistent across all those Blackfins as you move forward and through the Blackfin roadmap.

Finally VisualDSP Kernel is designed to complement the System Services. There are other BOLD training topics on this website that detail the capabilities the System Services, its Device Driver model. It is really designed to complement the VDK Kernel and not compete with it.

Let's move on and talk about the concepts and some of the concepts we'll be looking at with the VDK. We'll be looking at threading, we'll be looking at then priority and scheduling. There are three types of scheduling available within VDK. We'll be looking at pre-emptive, cooperative and time slicing or round robin type of scheduling. We'll be looking at critical and unscheduled regions and we'll look at then at two ways of coordinating traffic in the system, that being semaphores, including periodic semaphores and messages.

Beyond discussion today, there are a handful of facilities that we didn't have time to talk about today, but they are available and fully document should you prefer to use those. We have events and event bits, which is similar to a semaphore but has more complicated decision making behind it. We have multiprocessor messaging so if you're using either multiple Blackfins on a board or if you're using a dual-core solution like the Blackfin 561 processor, MP messaging is available for passing messages between those cores or between processors as the case may be. There are memory pools which you can think of as sort of being like a low fragmentation heap. There's also a device flag which is a way for a device driver to communicate status or certain events back to your application. All these things are fully documented in the VisualDSP Help and PDF manuals if you require a little detail beyond what's available from what I deliver today.

Chapter 2: Capabilities of the VDK

Sub-chapter 2a: Threads and Priorities

Let's move on and talk about the capabilities of the VDK. Threads and priorities, for Blackfin processors there are 31 priority levels that any thread can be running at. The number of threads that can be running on the system at any one time is completely arbitrary, can be greater then 31, and in fact it's probably likely that your application will have more then one thread running at certain levels of priority. Scheduling amongst priority levels is very straight forward, that is priority dictates and preemptively what the operating system is going to do. That is the thread with the highest priority that is available to run will always get the complete control of the processor until something happens that forces that thread to be set aside.

Where decision making becomes somewhat more complex and more interesting is when you have multiple threads running at that same priority level. In this diagram I have here on the right this is the second set of boxes. What I have there is something demonstrating what I'll call 'cooperative multiprocessing or multitasking'. This is where I have two threads that are working in conjunction with each other, one has total control of the processor until it is done with whatever it is doing, at which point it purposely hands the control of the application over to that other thread. So it's coordinated, it's completely coordinated between the two, or cooperative between the two, so it's passed back and forth usually be yielding the processor between two different threads.

Lower down the diagram I do have what I would call a 'time slice' or sometimes referred to as 'round robin style' priority. That's what you might be familiar with when using operating systems like Windows and Unix where each application is getting a little bit of the CPU time without too much attention from the user. What this does is the operating system or the VDK will automatically pass control the operating system around all those threads in a round robin fashion or in a time slice. The length of time that each thread gets control of that processor is defined by the programmer. Priorities can be statically or dynamically assigned, statically meaning at the time the application is built you declare what the priority is going to be. Or they can be changed at run time, either when you instantiate a thread or later on at run time with a thread.

Moving on there's a little more information about threads here. Threads can be instantiated either at boot time or a run time later on. In fact you must have one thread instantiated at boot time to get the system running otherwise the system would have nothing to do. VisualDSP will enforce that you have created at least one boot thread. After that you can create as many threads as you would like. You are, of course, practically limited by the amount of memory that's available in your system, but there's no hard limit imposed by the VDK itself.

Each thread does get its own stack, that is the place where the C compiler will put local variables and put its call stacks. So you don't have to worry about two threads accidentally sharing a local variable or anything of that nature. They do get their own copies of the stack, their own stacks. That's how each thread is, really the most simple level is the implementation of four functions, there's a create function, a destroy function, a run function and an error function. If you're versed in C++ you can really think of the create and destroy functions as being like C++ constructors and C++ destructors respectively. That is, they're usually small little functions that are run right as the thread is created to do some sort of static initialization. Likewise the destroy function or the destructor does whatever small amount of clean up is required to destroy the thread. The bulk of the time is spent in that run function. In fact it's very common for the run function never to return, it will be implemented as "while (1)" loop or some other type of infinite loop that application never breaks out of. If the run function ever does exit, the thread is then automatically destroyed.

Over here on the right you can see an incomplete list of the APIs that are used. Some are frequently used like clear thread, or create thread, excuse me. Others, especially when it comes to error handling, are really there for debugging purposes and probably wouldn't be of significant use in a final deployed application. It's worth pointing out the last two APIs listed on this slide, Sleep() and Yield(), those are ways of putting an application to sleep for a specific amount of time, at which point the VDK will automatically wake it up to begin running again. Also Yield() would be used in that cooperative multi-tasking situation that I talked about earlier where you have perhaps two threads that are passing control of the application back and forth between each other.

Sub-chapter 2b: Critical/Unscheduled

So critical and unscheduled regions are mechanisms to do small amounts of work during which time it's important that that operation is not interrupted. Anyone whose worked with an RTOS has certainly hit a situation where there's some small operation you need to do and it's critical that the operating system is withheld during that period of time, and so that the operating system doesn't step in and take control away during that critical moment. Critical regions and unscheduled regions are the mechanism that VDK provides that do that.

Critical regions are the most drastic, they simply will shut off the scheduler and all interrupts, so that process regardless of its priority, whatever else is happening in the system, will have complete and utter control of the processor. So it's convenient as a mechanism to use when you need to do something very atomic and something that can not be interrupted. However, it does need to be used with a discretion because it is possible if you leave the processor in a critical region for a long time, the interrupts will be un-serviced, other side effects such as interrupt latency will increase commensurately with the amount of time that you're in a critical region. Typically that's a used in a test and set or read-modify-write type of operation your, perhaps your application is querying the value of a global variable, make some decision based on it and then changes the value of that global variable. If it's important that the system not be interrupted during that time, then a critical region is probably what you're looking to use.

Unscheduled reasons are less drastic but they are related. What that will do is it will disable the contact switching only, so essentially the VDK will step back and not, the schedule will not be, will not be allowed to enter, however interrupts will still be serviced. On the right you can see the API set we have for critical and unscheduled regions, it's fairly straightforward. You will note that it uses a stack style operation, that is you push critical regions and pop them. The reason for this is that if you are writing a piece of code that needs to enter a critical region that code does not need to concern itself with whether it is already in a critical region. You simply pop, push the critical region and when you are finished you pop it, and without any concern about whether the caller, whoever called the function that is now running, is in itself in a critical region.

Sub-chapter 2c: Semaphores

Semaphores: Semaphore is probably the most straight forward mechanism for synchronizing events between threads. Essentially it's a way of coordinating synchronization between either the threads or between ISRs to a thread. Semaphore is usually used to control some access some shared resource. Classic example is that I might have an audio buffer that's being filled by one thread and being drained by another thread and I want to ensure that the next audio buffer is not overwriting the buffer that's being written out by another thread. So I might have a semaphore that controls the access to that. The reading thread, the thread that's draining the audio buffer would say, "Please do not write during this time." And then the thread that's actually filling the buffer next would then honor that semaphore, would wait until that semaphore is cleared so that way again there's no contention between two different processes trying to access the same buffer or peripheral or some other resource.

Most semaphores are of a yes or no Boolean nature. Like an example I gave of a buffer, either it is available or it is not. There might be a situation where you might want to have a shared resource where there's more then one available, maybe two or three, or who knows how many. In that case you might use something called a counting semaphore which is implemented essentially exactly the same way as a Boolean semaphore except the share count is simply greater than one. Finally it's worth noting that semaphores may be made periodic. That is the operating system will automatically at a time period defined by the application, post a semaphore. So this is a way of saying you have a background animation that you want to update at 24 times a second or something like that. You might want to implement that with a periodic semaphore, having a semaphore posted every 1/24 of a second to update that animation rather then say doing something somewhat less graceful like putting a thread to sleep. Over on the right is the API set for semaphores, like I say it's fairly straight forward to create and destroy. You can also create semaphores statically, that is, at build time when you create the application we'll take a look at that later on.

Sub-chapter 2d: Messages

Messages are a somewhat more sophisticated way of coordinating between threads. A message is a targeted transmission from one thread to another. So essentially you'll have a sender thread and a receiver thread. That sender thread will wrap up a message of some sort, which is then passed over to the receiving thread which then unpacks that message and does whatever action is that your application requires. It's implemented through the void pointer, sort of back door on the C language, if you're familiar with that, where you can take an arbitrary data type, and something of arbitrary length and cast it to this void pointer and there by be able to pass it around the system some what arbitrarily. Also it's very convenient – you can essentially pass anything between threads. The flip side of that is that does assume that the receiving end of the receiver of that message does know something about the content of that message so it's able to cast it back to whatever the proper data type is.

And beyond discussion today but I'll touch on it briefly is facilities for multicore and multiprocessor messaging, essentially sits on top of this messaging API so you can marshal messages say across a serial port or through shared memory in the case of dual core system like the Blackfin 561.

Over on the right is the API set for messages, fairly straightforward you see there are APIs for getting the message payload and getting other sorts of information about the message that a thread may have just received.

Chapter 3: On-line Demo

Sub-chapter 3a: Building VDK Projects

So next I'm going to switch over to VisualDSP itself. I'm going to just take a couple minutes and run through some of the windows that VisualDSP has to offer. We're going to take a look at the project creation window where you set up essentially the threads that you would like to have run your system and other build time decisions that you might make. I'll take a look at the thread creation template. That is if you want to create a new thread, we'll take a look at what VisualDSP does to help you do that. And finally we'll look at two debugging windows. We'll take a look at the status window which gives you a snap shot of where you application is right now, as well as a history window which gives you a viewport on the last N events that happened in your application.

So in here I have maximized the project window that comes with VisualDSP. If you've used an integrated editor at all you've probably seen windows exactly like this before. This is essentially my set of source files that I've used in my application. What's worth noting down here at the bottom is this kernel tab. If you've created a VDK application you'll get this kernel tab automatically. When I click over here, this gives me my VDK settings. So we can just take a moment going through these settings. My system, I can tell how fast my system is running. Here's where I control how big my history window to be, I can make it as large as I want within the balance of, if I have available memory of course. There's also instrumentation decisions. Instrumentation is error checking code that the VDK puts into its APIs to catch errors during development. Generally when you're deploying your application, you'll turn instrumentation completely off.

If I want to create threads or examine my threads I can do that here. Again these aren't threads that are running system, these are threads that I'm defining within my application. So here I have a number of thread types, I've created four thread types in this application, and the exact nature of this application isn't particularly important, it's the, you know, the use of the VDK I'm trying to demonstrate here. Within each thread I have what the priority is, in this case my boot thread here got the absolute highest priority, Priority One being the highest. And I can put in other parameters about the nature of that thread. The stack size, this is an important one, this is again I said that each thread had its own dedicated C language stack this is where I would define how large that is. And I can do this: I can create an arbitrary number of thread types here.

Boot threads – as I mentioned earlier, you need to have at least one boot thread type, or boot thread, and so I've done that here. I've called it simply BootThread, I declared that there. And round robin priority, at any one priority level that priority level can either be cooperative or can be round robin. And so in this case this application I've left all the priority levels to be cooperative except for one – I've made Priority Six to be round robin. And the periodicity there is ten, that's ten ticks, and tick was set up here in the system settings. So that's how long the operating system will wait before moving control from each thread in that priority level.

Semaphores which we talked about earlier are defined here. In this case I said I wanted to have a maximum of five at any one time running in my application. And I've defined actually four static ones here. I can do this at run time but in this case it was more convenient to define them statically at this place. So here I have something called UARTAvailableSemaphore which is initially true -- available -- and its maximum count is one. If I wanted to have a count at setting four, that is something more than just a Boolean yes/no value, I can change that value here by specifying the value there. And if I wanted to make something periodic I can do that here, statically in the settings as well.

As you can see down here some of the other settings for the more advanced features that we're not talking about today, but essentially their creation is very much like the project things we've been looking at.

Let's take a look at creating a new thread. If I wanted to create a new thread I just right click there and I want to create a new thread type, I'll just call it, "NewThreadType". In this case I can choose which language I want, I'll pick a C language. Would I like to file as regenerate automatically? Yes I would in this case, so it's created this new thread. And in fact it created these two source files NewThreadType.C and .H. If I double click here I go to this thread and you can see this is the code that the VDK or the VisualDSP environment just created for me. As I mentioned earlier there are essentially four functions in any thread and those, they are here. Here's my run function, here's my error function, and my initialization and my destroy, my constructor and destructor and more C++-style parlance. So again all of this was created from me. I mentioned earlier that many applications the run threads never terminate and that's actually the assumption that's made here. The actual the while (1) loop that presumably would be in this run function has already been inserted for me. If that's not the behavior I want I can of course simply delete it and move on.

Sub-chapter 3b: Debugging

So let us pretend that we've been developing our application, and we're running it and we're trying to debug it. And so what I did earlier is I let this application run for about 30 seconds or so, so I already have a good bit of history and status built up already, so let's go take a look at that. First I'll look at the status window. So what this is is a snap shot. When I halted the processor and I just did that by pressing the halt button, there's no special event there. This is a snap shot of the state of the system right now. So we can take a look at that. I have ten threads to find and four semaphores. So I can drill down to this for more information, in this case let me resize a little bit. I have a number of threads that are blinking various LEDs on the hardware board that I'm hooked up to. I also have an audio thread and a couple other ones that essentially run away and take all the CPU time. I can get further information on each thread, my audio thread perhaps being the most interesting, you can see that its status right here is that it's blocked on semphore ID 3. Remember that because we'll see that again later. So I can open this up and I can get all kinds of information about how much stack has actually been used, where that stack is located, what its priority is, how long it's been running, etcetera. If it encounters an error of some type, that will be reported down here in the last error type and last error value. And we see it's at status right now of semaphore blocked and ID of 3. So it makes sense if we take a look at our semaphores we'd see some matching information there. And in fact we do. Here are the semaphores I've created, here is semaphore ID 3 which is called kBuffer2 available. If I open that back I do see that there is a pending thread, there's one pending thread and that being thread ID 10. Which does mirror what we saw just a few seconds before. So you kind of either look at your application from a semaphore-centric type of view, or a thread-centric type of view. If I had messages in this application, which I do not, similarly I could do the same type of correlation there.

The final debug window that's worth looking at is the State History. So here is what the State History diagram looks like. It's, again the specifics of this application are not important, it is nicely color coded, if you, when you first start you might want to pop up the legend which tells you what all these colors mean. I am limited by the resolution of what we're working with here, so I'm going to turn the legend off for just now. Here are all my threads and this green line that moves back and forth shows the move of execution between my threads. And the orange dagger and all these other symbols indicate traffic or history in my application. I can if I want to, say, if I want to inspect this piece of activity further I can, sort of use my cursor to sort of drag and zoom in on here, perhaps I'll even zoom in a little bit further to get down to where I, I've gotten down to a meaningful subset of information. At this point I might want to take a look at these individual events so if I right click and go to "data cursor" I can now use my left and right arrows on my keyboard to move through the history of my application. If you keep your attention down here in the lower left corner of the window, along here at the bottom it actually displays more information about the actual event that I'm looking at. So I can trace my way, exactly what my application did. If the display is too busy, I want to focus on a subset of the type of activity that's available, I can do that here by right clicking and going to my filter. I can either deselect the events I'm not particularly interested in, or if I want to only look at, say the interaction between only two threads, I can filter that out here.

Chapter 4: Timings and Sizes

Sub-chapter 4a: Memory Footprints

So I'm going to switch back to the prepared material now. And we'll wrap up with some size and timing information. First of all it's worth saying that it's very difficult, the question we often get is, "Well what's the overhead of the VDK? Or how much, what's it going to cost me in either code size or performance to use VDK?" And it's always a very difficult question to answer because that is highly reliant on the nature of your application. So what I've done here is pull together some numbers I think are representative of what an average application might be, but again the individual answer for any particular application will vary depending on what it is your application requires.

So firstly I have a list of some static memory foot prints. This is the amount of code and data that is consumed in link time by the VDK in these scenarios. Again this is for the VDK only, this isn't the entire application, just for the VDK's portion of the executable. I use this, measured this using VisualDSP 4.5 and I have dead code and data elimination on. Again all these sizes are in bytes. So if we take the first row in this chart, I have an artificial best case scenario where I have one C language thread and there's no API calls at all. So I have about five kilobytes of code and one kilobyte of data. So that gives you sense of the overall overhead of VDK. And moving down, there's subsequent rows where I'm starting to add more and more functionality. If I have two threads and I have semaphore static semaphore passing used to coordinate between those, I get it to around seven kilobytes of code. You can see the amount of data, however, remains fairly constant, only increases maybe 10% as I start moving down this chart. If I had message passing, I'd get up to about nine kilobytes. Where I get the big jump is this last row in the column, and here's really a debugging scenario. And this is not something you can expect in a deployed application but when you're debugging and want to have full instrumentation and want to have a history window to enable those great features that we saw just a few minutes ago. In this case, my code went up to about 13 kilobytes and the data went up to approaching 10 kilobytes, however it's worth noting that that's 512 events times 16 bites per event, if you do the math that's 8 kilobytes just for the history window. So you can see that the vast majority of that data footprint is by the size of the history window. And that is application specific, so you can make it smaller or very large if your debugging task demands it.

Sub-chapter 4b: Cycle Counts

So finally I'd like to talk about cycle counts for certain things that the VDK does that tend to be very performance sensitive. First about the environment in which I ran all of these, this was the entire application residing in L1. If you do have portions your application residing in L3, performance likely will not be as good, however if you manage your cache properly you can get numbers that are very close to this. This is for an application actually very similar to that diagram I showed you earlier where I had five threads running at a couple different priority levels. I was using a Blackfin 533, revision 0.5 silicon to get these numbers. Boot time was 15,000 cycles, we want to convert that to time, and you know the frequency which your processor is running at, you can do your math yourself to figure out what that means in milliseconds or microseconds. Ticks you see only took 67 cycles. If I was not changing where the thread was, however if I'm in a time slicing type of situation I do change to another thread, that came out to 722 cycles. Similarly posting a semaphore, if post a semaphore and there's no change of execution, only 76 cycles, posting a semaphore there was a change, it was close to 300. Critical regions, hhe act of pushing near critical region, incrementing a global variable, and popping, so the amount of time to do all three of those activates was about 200 cycles. And finally if I'm creating a new thread, in this case I am malloc()'ing it on the heap, it took about 2300 cycles to do all that.

Chapter 5: Conclusion

Sub-chapter 5a: Additional Information

So in conclusion VDK is provided at no additional cost to VisualDSP. It is royalty free. There is no additional maintenance fees with it, however there are also many commercial operating systems for Blackfin that should not be ignored as well. The facilities that we looked at today, we looked at threading, we looked at prioritization levels, we looked at semaphores, we looked at messaging and critical and scheduled reasons. As I alluded to earlier there are other facilities that are available that we simply did not have time to talk about today, but they are fully discussed in the VisualDSP documentation if you'd like to learn more about those things.

As we saw from the live demo VDK is well integrated into VisualDSP itself. There's facilities for managing static settings of your project, that is the threads and the semaphores that we were creating. The VisualDSP does provide templates for when I create a new thread type it gives me the skeleton code that I need so at least I have something that will link, will compile and link immediately. From the debugging side there's a status window and there's also the history window which gives me a detailed look at state of my application just now, in this case, the status window, and then the view of the recent past of my application using the history window.

For any additional information we do invite you to review other BOLD topics, especially the System Services, there's a couple of training sessions based on System Services, and its Driver model, it really does complement the VDK quite well. VisualDSP is available in test drive form, that's a 90 day free evaluation. You could get yourself an EZ-KIT Lite at low cost. There's the URL that you would use to get to the test drive that's here. There's also the path that you would browse to in the Windows Explorer to get to the VDK examples there. It's worth knowing that the vast majority of the examples that demonstrate VDK capability do not require hardware, so if you're not prepared to make a commitment to an EZ-KIT Lite, the free test drive will let you take a look at about 90% of what it is the VDK has to offer.

There is detailed documentation available. Within VisualDSP, that is you install the test drive, it will be on the online help system, it's also available for download separately in PDF form and it's also on the CD that ships with VisualDSP.

Finally again invite you to take a look at the third party operating systems that are available. You saw from some of the earlier slides there is quite a number of them at various price points and levels of capability. The URL here is the place to jump off to to go find more information on that.

Finally there is an "Ask a Question" button on your display. If you click that button, you can send a question directly to Analog Devices.

Thank you for your time, again my name is Ken Atwell, Product Line Manager for Analog Devices. Thanks for listening about the VisualDSP Kernel and have fun, thank you.