Presentation Title: Basics of Building a Blackfin® Application
Presenter Name: Joe Beauchemin

Chapter 1: Introduction

Subchapter 1a: Agenda

Hello, and welcome to the basics of building a Blackfin application. My name is Joe Beauchemin, and I'm a Blackfin applications engineer at Analog Devices.

The module that we're going to discuss today builds upon the introduction to Visual DSP++®. In this module, we're going to discuss what it takes to take your existing C Code and port it over to

run on a Blackfin processor. While we talk about that, we're going to have to discuss the software development process, the build process, which goes through the compiler and linker to generate the executable code that is needed to run on the processor. We'll discuss some "gotchas" in programming on the Blackfin specifically, and offer some basic code optimization strategies to be used for system optimization level testing later on in the process. We'll conclude with a simple, two-step, nearly zero effort optimization strategy using just the optimizer and enabling the instruction cache. In terms of an agenda, we'll start off by walking through the software build process in Visual DSP++. In it we'll talk about the linker, which is probably the most important part of the process, because understanding the linker is going to enable you later on to take advantage of system level optimizations by moving code and data around in memory to suit your needs. To do that, we'll have to discuss the linker description file. After that, we'll get in to some programming Blackfin processors in C techniques, things that are available to you as a developer to program on the Blackfin processor specifically. After that, we'll take you through some ways of creating efficient C Code and conclude with a real life example, which was code that was taken right off the web and built using Visual DSP++.

Chapter 2: VisualDSP++ Build Process

Subchapter 2a: Porting C Overview

To start things off, we'll go with a software build process in Visual DSP++. When you're porting C Code to Blackfin, the code, as long as it's ANSI C compliant, will build and execute "out of the box". Because you're dealing with a limited memory space, the code could be large and slow. If the code is too large to fit into your internal memory, it's going to spill out into the external SDRAM, and that's obviously a slower interface. So the code may run slower then you expect. By default, when you start off, the optimization switches are turned off. The reason for this is because it's very difficult to debug the code with the optimization on. So, the code generated will be un-optimized, but functional. In addition to that, the default clock settings are used on the processor and the instruction cache is off.

In terms of the software development flow, this slide is an introduction of the types of files that are involved here. Starting from the left, you have your source files, which are either C or Assembly, and those will go into the compiler along with the C Run-Time Header, called basiccrt.s. We'll discuss this run-time header in a moment. Through the compiler and assembler, object files are generated, which are given the doj extension. These object files contain all the information that the linker needs in order to resolve the application into memory. It's code and data with tag information, essentially. The linker takes those object files, plus any third party libraries or libraries that you generate on your own, called DLB files, and uses the linker description file, which we call the LDF, to resolve all that information into your executable file, which is called the DXE file. That executable is where we'll actually conclude today because we're using it on the EZ-KIT Lite™. The executable runs on the target or in the simulator, but in an "end application"

the executable is just the beginning because you need to have something that takes that executable from external memory and maps it into the appropriate memory sections. And that's called the Boot Image, which is the loader file. The loader file may require this other thing called the Boot Code, which is a separate DXE, because if you spill into the external memory, then the hardware needs to initialize that memory before it can boot things into it. We call that Initialization Code or Boot Code, which needs to be pre-pended to your executable code in order to get the end product. Like I said, we're going to concentrate from the source files to the executable today.

Subchapter 2b: Compiler Fundamentals

In the previous slide, I mentioned the C run-time header, which is called basiccrt.s. What's this do for you? It sets up your C run-time; it sets up your stack pointer; it enables your cycle counters; and it gives you the ability, through a start-up wizard, to configure cache if you want to use cache memory, data and/or instruction and to change the clock and voltage settings that are run on the chip. It can be modified through the "Project Options" window later on, but it is also fired off at the beginning using the project wizard, which you see here on the left. If you select "Yes" here and click "Next", then it'll take you through the optional pages of configuring the cache and all that stuff.

The software build process, where does this begin? You start with your C file on the left here, which is the cFile1.c, and in it I'm just defining a simple main along with a function called func1. We feed that into the C compiler and the compiler generates an intermediate file called the .s file. All that is is the compiler's attempt to translate your C source code into the assembly code needed by the Blackfin processor to run on the part. It is fed through the assembler and it generates the object file on the right. In this, you'll see the tagging information. You have the object section called "program", which is the default section name given by the compiler to all code not explicitly specified by the user as a different section of code. So "program" is the default, and you see the assembly code in there for the main and the func1, which we defined over here. In addition to that, you see the stack section down here. The stack holds your local variables and other frame pointer and stack pointer information, as well as registers that are backed up and restored by the C run-time environment on its own.

On the previous slide, we discussed the compiler-generated object section name called "program". This slide here is a look-up slide for you. It's a reference slide to show you the types of labels that are automatically generated by the compiler. Anything that is code-related is going to be tagged as "program", anything that's globally declared data is going to be called "data1". Those are the two big ones for the purpose of this demonstration. These section names are important because these are the tags that the linker is going to use to parse through all of your input object files to create the executable at the end. You're not limited to those sections names. You have the ability to come up with any section names that you want, and this slide is a

demonstration of how to do that using alternate sections. In this source code on the left here, we're simply using the extension "section ("sdram0")" to tell the compiler that I want this integer array, instead of being resolved into the "data1" section that you would normally put it, place it out into external SDRAM, tagged by "sdram0". Similarly, you have the flexibility to do the same thing with code, and for this example we're tagging "L1_code" as the section name that we want to give for this function called bar. Again, it goes through the compiler and the assembler, and the object file is generated on the right. In it, again, you see the tag for "sdram0" and the integer array showing up here. And then the object section called "L1_code" with the assembly break down of the bar function, which is over here, and the stack is there as well.

The reason that we use the labels on the previous slide, "sdram0" and "L1_code", is because, in the linker description file itself, we've assigned specific section names that you can use as alternatives to the normal produced by the compiler. So, "sdram0" is any code or data that you want placed in external memory. "L1_code" is code that you definitely want to have mapped to on-chip program memory, which is faster and more efficient than executing out of external SDRAM. Finally, we have "L1_data_a" and "L1_data_b", which respectively map to the two on-chip data memory banks.

Subchapter 2c: Linker Fundamentals

Once we have an understanding of how the compiler generates these object files, the next step is to understand the linking process. This becomes crucial when you get to the system level optimizations later on, when you're actually placing things in faster memory because they're called more often or used more often than stuff that you want in external memory. What does the linker do? As I said, it takes in the compiled or assembled object files, the doj files, along with your linker description file and all the library code that you have, and it creates a fully resolved Blackfin executable file. That DXE can run in a simulator, or it can run in a target via the emulator or the debug monitor. The important file input to this is the LDF, which defines the hardware system for VisualDSP++. It specifies the processor that you're going to use for the project, and it also specifies all the internal and external memory available to the linker to resolve things to. If the user doesn't define the memory, then the linker can't use it. So, it's imperative that the user has an understanding of the memory map and flesh out all the memory segments for the linker to use to achieve optimal performance. While we're talking about memory, it should be noted here that the Blackfin memory is highly efficient and is a hierarchical scheme. So we have what we call L1 memory, which is simply on-chip level one memory. It's the fastest, and it runs at the core clock, so it's where you want your code that's called the most frequently to reside. After that, we have some processors, the BF535 and the BF561 dual-core, that have on-chip level two memory, which is slower than the L1 memory, in that it runs at core clock over 2 at the maximum. Finally, the furthest from on-chip is the external asynchronous SRAM or SDRAM external memories. It could be called just off-chip memory, or sometimes "L3 memory"; you might hear that term as well.

Each of these memory regions has its own address range, and that's all fleshed out in the LDF for you, which we'll get to in a moment. But, it's important to note that accesses to these memories are all automated by the processor. If you have a pointer set up to SDRAM, then all you need to do is access that pointer and the processor will take care of accessing SDRAM on its own, provided that you have it set up in the hardware.

Chapter 3: Controlling the Linker

Subchapter 3a: Link Process Overview

That brings us to the meat of the linker, which is understanding the linker description file itself. That's the bright blue box here, and we see the LDF governs the linker by taking in all of these input files and the object code and the library code, with all of these labels that you see here, and it takes it and resolves it into executable output sections on the right over here.

Subchapter 3b: LinkerDescriptionFile

What does the LDF do? The LDF expresses how programs are mapped into memory. It's the way that the user describes where in memory to place parts of a program. Every project absolutely has to include an LDF file. If you do not include one, then there are a bunch of template LDFs included in the VisualDSP++ install path that the linker will choose by default, based on the processor you're using and what language you're programming in, if you do not specify one on your own. In the LDF file, there are a couple of global commands that you'll see at the top. You have to first set the target using the ARCHITECTURE command, and then you need to define the memory layout using the MEMORY command. Now we're going to talk a little bit about the details of that. Again, the user has to declare all the available system memory. If you're going to declare it on your own, and not use the template LDF, if you refer to the system memory map in the data sheets, you can define the segments and not cross any boundaries or do anything that's going to result in illegal memory ranges. Again, here are your global commands. You have the ARCHITECTURE command, which is saying that, for this example, we're running on a BF537 processor. Then you have the MEMORY command down here, which is where we define the memory segments. On the left are the memory segment names, and these are the default names that you'll see in the LDF files that are in the templates that we provide. These names can be anything that you choose, but these are the ones that are in the default. After that, you define what TYPE of memory it is, which is always going to be RAM on the Blackfin processors. And then you provide a START and an END address. Again, this is all user-specified, but for the sake of demonstration, the BF537 ranges are shown here. Finally, you have a WIDTH command inside of this, and for the Blackfin processors, this is always going to be 8 because it's a byte-wide addressable memory space.

After you have the memory defined, the next thing that the linker description files does for you is it actually says where in memory to place these input sections, and that's done using the

SECTIONS command. You'll see here that the INPUT_SECTIONS command is underneath these two levels, and we'll get into an example of that on the next slide. But, the INPUT_SECTIONS command is what describes what contents from these input files get mapped into the current output section that we're in. So, the usage is INPUT_SECTIONS with this *file_source* and *input_labels*. And *file_source* is simply a list of files, .doj files, or an LDF macro that expands into a list of .doj files. What it's saying is to parse those .doj files and look for this input label, and *input_labels* is a list of these section names. An example would be the "INPUT_SECTIONS {main.doj(program)}". In this example, it's basically telling the linker to go into the main.doj object file, find everything tagged as "program", and map it to this section if it can. In terms of a physical LDF file, here's what it would look like in the default template. Here we have the processor with the SECTIONS command that we just discussed, and we're defining the DXE section names here. These names only mean something in the .map file, which is one of the debug tools that are provided with VisualDSP++. You can open up a map, and it'll say what the output section name is and list everything that's in there. The critical stuff for this is the stuff in green, the object sections. These come from the assembly, or the .S files, which were the compiler-produced assembly code. As you see, the "data1" is the default global data section that the compiler will produce for all global data. It'll take that "data1", it's going to look in all the object files, take that "data1", and place it into the memory segment defined here on the right. These memory segments are exactly the ones that we saw on the previous slide. One thing to note here is the ability to map input sections to multiple memory segments, which we'll discuss later on in the description of what the LDF file is. "data1" and "program", notice they have dedicated memory segments that are on-chip, MEM_L1_DATA_A and MEM_L1_CODE, but they're also being mapped down here to the external memory.

And I'll show you how that works next. But first I want to discuss macros.

Subchapter 3c: How the Linker Works

On the previous slide you saw the $OBJECTS here. $OBJECTS is simply a macro that's defined in the LDF, and it includes the CRT, which is the C run-time object file generated from basiccrt.s that we discussed early on in the presentation. And $COMMAND_LINE_OBJECTS, that's basically all of the object files that are produced from the input source code that is in your project. The ordering of these object files is important because of the way the linker works, and that order will be exactly the order that it appears in your project window. Just to show you quickly here, this is VisualDSP++, the session is open, it's for a BF537-EZ-KIT, and over here on the left I'm showing the "ModuleFileProject", function 1, function 2, 3, 4, 5 and "ModuleFileProject.c". These 6 source files would be in the $COMMAND_LINE_OBJECTS macro in this order that they appear here on the left. Why does that matter? You have the ability to flesh out macros on your own and map them into specific output sections, and we'll talk about that later on as well.

How does the linker work? As I said, the object sections can be mapped into many memory segments. The way that the linker works is it parses the LDF file from left to right and from top to bottom. It goes through all of these .doj files, searching for the tagging information that you have defined for it, and it says “OK, if this fits in the memory segment that we’re currently processing, we’re going to place it in there now. If not, we’re going to put it off to the side. And we’re going to go through the entire line and all the way through and see if everything fits”. If it doesn’t fit, then a linker error is going to be generated at the end. If there’s objects left over that spilled over from internal memory and also don’t fit into off-chip memory, or aren’t defined to go into other output segments of memory, then you’re going to have a linker error because you have some source code that hasn’t been mapped and isn’t going to be used by your end application.

Just to elaborate on how the macros work, if you had “main.c” and “file2.c” input files, and you have the $OBJECTS macro here, which comes from the default template LDF that I showed on the previous slide, then, to the linker, it’s going to see that as this: where “main.doj” and “file2.doj” are individually parsed for these labels. The linker is going to go into “main.doj”, take everything tagged as “data1”, and try to place it into MEM_L1_DATA_A. If it’s successful, it goes in; if it’s not successful, it gets pushed off to the side, and then the “file2.doj” is parsed to try and locate the same “data1” sections. It concludes with MEM_L1_DATA_A, moves on to MEM_L1_CODE, and performs the same tasks for the code for the “program” label. And then, when it’s done with that, it moves on to the SDRAM section here, and in this particular example, we’re saying we want everything that overflows from “data1” and from “program” placed into the large external memory that we have available in the SDRAM. So, the linker is going to take everything left over from these two functions and place it into external memory. And that’s why, if you have one giant C file that you want to have ported over to Blackfin, you may have a large enough file such that it only generates one “program” section with all of your code in it, and it just doesn’t fit in on-chip memory. So, it’s going to be placed entirely in external memory and you’re going to be executing entirely out of the SDRAM. Understanding the linker description file is going to give you the ability to break your code up into modules and to move things around in memory so that you take as much advantage as possible of the memory structure to get the best efficiency out of the part.

Subchapter 3d: Linker Optimization

How does optimization work using the LDF file? You can optimize for size using a feature called the ‘eliminate unused objects’ button; and that is, if you open up your Visual DSP++ session; you right click on your project name; you can bring up the “Project Options…”; and, under the link tab here, you’ll see the “Elimination” sub-tab; and, in that, if you click on “eliminate unused objects”, that’s going to say everything that was linked in by the libraries - because you used something from the libraries or what-have-you - anything that you don’t use from those libraries is now going to get taken out and not mapped to waste your memory space. That’s how the “eliminate unused objects” feature works. In terms of performance, you have the ability, using the linker LDF as I

discussed, to move stuff around in memory so that it best fits your needs. If you wanted to put a function in L1, you would use this line of code: “section (“L1_code”) void my function”. If you wanted to place an array explicitly in on-chip Data A, then this is the code that you would use to do that. You have the ability to, and this will come up when you get into the system optimization techniques or Blackfin optimization techniques in future modules, if you have a vector multiply, for example, where you’re taking data from two different memory banks to avoid access conflicts, you can explicitly place one of the input vectors “mult1” into Data A and “mult2” into Data B, and this gives you the ability to do that.

Additionally, you have the flexibility to create a special section name, and for the sake of this example I called it “Map1st”; basically, I’m trying to say that I want to tag stuff as “Map1st” to tell the linker that, no matter what, this needs to go into on-chip memory and it needs to go in first. I’m going to show you an example of that in a moment. Finally, you have the ability to create special macro lists to prioritize how input files are parsed, and we’re going to go through a couple of examples of that right now. The project that I have open right now is called “SingleFileProject”; and, in it, we already have the source code open here. It’s essentially an infinite loop that calls these five functions, all of which are defined inside this single module. The header file here just prototypes all those functions. If you build and run this code, it’s going to take everything and place it in on-chip memory. This module isn’t big enough to cause overflow into external memory or anything like that. After it builds, you can take a look at the disassembly window by clicking on the browse button in the disassembly window, and it brings up all the symbols that are in your debug code. If you click anywhere in here and start typing, we saw the function names were function1, 2, 3, 4, and 5. I type “fun” and it gets to the “function1” label. You can see that the linker took function 1, 2, 3, 4 and 5 and mapped them continuously in on-chip memory. The 0xFFA0____ (start address) is on-chip Level 1 code, and functions 1, 2, 3, 4 and 5 are mapped in order because that’s the order that the linker parsed it in when it generated this executable file. If you have one large input file, that’s all well and good, but one thing that you can do, and this is a stepping stone to the next level of optimization using the linker, is set this to your active project. This is a module approach where I took the main file, did the same infinite loop delay with the same calls to the functions, but I broke those functions down into their own files. So you see “function1.c” here does everything that it did previously, only now it’s got its own dedicated file. And, if you build this one using the “build all” icon up here, the end result should be the same because we haven’t told the linker to do anything differently. It’s just parsing the same exact information, but it’s doing it continuously based on the files as they’re parsed here on the left hand side. Again, functions 1, 2, 3, 4 and 5 still reside in on-chip memory with the 0xFFA0____ prefix, and they’re still continuous in nature.

Moving on to the next example, after you have an idea of how the modules work, the next example project is called “AlternateSectionCodeProject”. I’m going to set it as my active project.

In this one, I've taken the main, which is this one here. It's the exact same main as the first one, with all the functions defined in the same source code, but instead of having them all resolved into on-chip level 1 memory, I've gone into the header file and, in the prototypes, I've added that "section("sdram0")" here to tell the linker that I want it to place function4 and function5 into external memory. If you go ahead and build this project, and we'll get rid of some of these source windows, and you go into the browse function here and again type the same thing for function1, 2, 3, 4 and 5. Now you see that functions1, 2, and 3 reside continuously in memory, but now 4 and 5 are mapped external into the SDRAM, which you can see with the 0x00_____ prefix here. So, they're continuous in external memory, whereas functions1, 2, and 3 are in internal memory. This is where knowing the linker file and having the ability to move things around come in handy for system level optimizations later on.

Proceeding on to the next one, the next section is actually modifying the LDF instead of the source code. In this one, if I open up the LDF file here, we can scroll down past all of the pre-processor stuff that the tools use. We can get to the "Map1st" section name. The "Map1st" section name is the one that I'm giving to the code that I want to have absolutely placed in on-chip memory at any cost, and only on-chip memory. In this one, I changed the INPUT_SECTIONS so that it parses all the object files that are on the input line, finds the "Map1st" label, and places it into on-chip memory in the MEM_L1_CODE section. How does that work in the source? If you open up the source code here - in the header file - you can see that now, in addition to placing stuff explicitly into the external SDRAM, I have instructed the linker to take the "Map1st" label and assign it to function2. When I build this code, we should see function4 and 5 in external memory, and function2 should be placed inside of level 1 memory first. Again, I browse and I look for the label where all those functions are defined. You'll see that function2 resides in memory at a location lower than function1. The linker went through, took function2, put it in there first, and then went through and grabbed function1 and function3, and placed those *after* function2 was mapped. Also, just like the last project, function4 and 5 reside in off-chip memory.

Finally, you want to have the ability to use macros to treat certain input files with priority. For this one, again, I'm using the module approach, where I have the main that calls all the functions that are defined in separate input files. If I wanted to define a macro, I would do it at the same place in the LDF that the default LDFs do. The $OBJECTS includes the C run-time and some other initialization object files that are needed for the C run-time to perform properly. But now I've defined two macros called $MY_L1_OBJECTS and $MY_SDRAM_OBJECTS. What I'm telling the linker here is: I definitely want function2, 1, and 3, along with my main, to be in on-chip memory. And I want function4 and 5 to be in off-chip SDRAM. This becomes a powerful tool because, if you have one or two files that are of the utmost importance to reside in on-chip memory, you just need to make the change once here instead of somewhere else, where you're

going to have to put explicit labels on every line of code and every piece of data that you want mapped. In addition to defining the macros here, you have to use the macros in the SECTIONS command where you map things; and, as you can see right here, here is where we are taking the $MY_L1_OBJECTS, looking for the “program” label, and placing it into MEM_L1_CODE. Similarly, down in the SDRAM section, you’ll see where I made the modification to place all the $MY_SDRAM_OBJECTS sections - the “program”, the “data1” and the “const” data - into off-chip memory. If I go ahead and build this project, we should see how the macro had an impact on the system. I’ll get rid of some of these source files and use the browse button to go to the functions. Again, there you go: the macro was defined, it defined function2 to reside in memory first, so the 0x_____174 is the lowest memory address for those functions; and then function1 and function3 reside after that; and, again, function4 and 5 are off in off-chip memory.

That concludes the demo portion of moving things around using the linker description file, which you can take with you when you start considering system optimizations in other modules later on in the series.

Subchapter 3e: Expert Linker

After going through the text file, which is the linker description file, it should be known that there’s also a tool in VisualDSP++ called the Expert Linker, which we’ll cover now. The Expert Linker presents a graphical interface for manipulating the LDF files. You can actually resize the memory that we defined in the MEMORY section of the LDF by using the grab and stretch mechanism. You basically grab on to one boundary between two memory segments and move it, enlarging one at the expense of the other. Additionally, you have the ability to take sections, using drag and drop, and place them into these output memory segments. With the Expert Linker, the left hand side is going to show you the object files that you have available, and the right hand side is going to show you the memory map. You can use drag and drop to get that performance the same way that we just did it manually by editing the LDF file itself. Another cool thing that the Expert Linker does is it provides watermark information for your stack and heap, two run-time considerations. The stack is needed by the C run-time environment, as is the heap. When you’re developing your code, you can run it, and then the Expert Linker will give you the ability to see how much stack space - and/or how much heap space - you actually used so you can get an idea of what the maximum amount of memory is that you need dynamically at run-time and modify the depth of the stack and heap segments to get more memory available for the other stuff, like your code and your data. The important thing to know about the Expert Linker is that you’re simply changing a text file underneath it all. You’re making all these changes graphically, but all it’s going to do is modify the text associated with it. When you modify or resize the memory, you’re going to be changing the START and END addresses in the MEMORY section of the LDF. If you change the drag and drop and move sections around, you’re basically modifying just the SECTIONS part, where you’re saying: take this object code and place it into this memory

segment. The Expert Linker isn't going to stomp on everything that was there previously, it's only going to change the things that you modified in that Expert Linker session. Changes that you make in the text editor will be reflected in the Expert Linker every time you fire it up. Things that you change using the Expert Linker are going to just modify the underlying text that we've already been through.

In addition to having the ability to change LDF files using the Expert Linker, you also have the ability to create them. This slide is going to show you how to do that. Under the "Tools->Expert Linker->Create LDF…" pull-down menu, you'll be prompted for a name of an LDF file and the language that you're going to be programming in. It's important to know that C++, C, and assembly are a hierarchy, if you will. If you mix C++ and C source files, that's fine, but in terms of the LDF, you would need to say that you're using C++ because C++ includes nuances like generating constructors and destructors, whereas C would not have any use for that. Similarly, if you have a mix of C and Assembly, you would want to choose the C template because the C template would define the C run-time, which is needed to run a C application. Template LDFs are available in this directory, which is in your install path - the \ldf directory under the Blackfin sub-tree - and they have the suffixes _ASM, _C, and _CPP for C++. For example, if you wanted to use the C LDF file for the BF537, you would choose this one here. Just so you can see how to do that - if you go under your "Tools->Expert Linker->Create LDF…", it's good because it's giving me an error saying that an LDF file already exists. You do not want to have multiple LDF files in any given project. If I want to generate one that's new, this is the LDF Wizard as it comes up. I'm just going to cancel out of this and not actually run through how to do it. But that is the Expert Linker.

Chapter 4: Blackfin C Programming

Subchapter 4a: Overview

Now that we've discussed the tools development process, the next section is going to concentrate on programming specifically for the Blackfin in C. Coming over from another architecture, a microcontroller, or a PC, you may not have had a rich set of peripherals to use like the Blackfin offers you. To make use of peripherals, you need to have an understanding of what a memory-mapped register – MMR - is. There's an MMR set dedicated to each peripheral for configuring and using that peripheral, like serial ports, UARTs, the CAN, etc. Having the knowledge of this will enable you, as the user, to do things on your own. You should know that if you're using system services or the device drivers that the things that are discussed over the next few slides are automatically handled for you. You don't need to worry about programming the MMRs if you use those features.

Subchapter 4b: Using Header Files

If you do choose the program on your own, we offer a couple of header files to make things easier for you. The defBFxxx.h header file essentially maps all of the memory-mapped registers to their associated addresses in memory. We're basically saying that this EMAC_STAADD register can be referenced by this explicit pointer right here. To have that ability is important because you want to be able to access these registers and have your code be very legible. It's a lot easier to say "OK, I'm writing to EMAC_STAADD" than it is to say "I'm writing to 0xFFC03014". That makes debugging kind of a pain. In addition to defining the memory space, it also defines all the bits that are in those registers that are defined in the Hardware Reference Manual. Instead of setting a register to a specific hexadecimal value, you can set it to an ORed result of a bunch of different bit locations, and your code would appear to be setting things expressly rather than implicitly by some hard-coded number.

In addition to defining the bits and the registers themselves, we also offer some useful macros for commonly used tasks. If you wanted to set the PHY address in this register - instead of having to write the number, mask it so that it fits in the field properly, and shift it into the right location within the register - you can just use this macro to do that for you. These header files are defined in the \include directory, which you can find right here. That's the low level header file. On the higher level, we have the C header file, which takes everything that we defined in that previous header file - the memory map, the addresses, and their bits - and it creates a useful macro that a C developer would use. Using the classic *p prefix notation, you can access the entire memory map simply by *p and then, in all capitals, the name of the register that you want to access. That becomes very important because, on the Blackfin, you have to have aligned accesses. If a MMR is 16 bits, then you need to access it as 16-bit memory. If it's 32 bits, then you need to access it as 32-bit memory. If you don't adhere to those rules, it'll generate a hardware error, and the write or read will not happen properly. Again, just a reminder that the system services and device drivers take care of this for you automatically, but if you ever wanted to program them on your own, you should know that that's the way that the hardware will work. That said, there's one special "gotcha" when you're programming in C, and that is with regards to the serial ports themselves. The serial port hardware is the only hardware in the Blackfin that can be either 16 bits or 32 bits, dependent on how you have the hardware configured. If you have it configured for 32-bit data, then if you access it as a 16-bit location, it's not going to work, and vice versa. If you use the wrong macro, the code will build and run, but it won't do what you expect it to do. So, if you're ever writing or reading the SPORTx_TX or SPORTx_RX data register, whether it's SPORT0 or SPORT1, you need to be aware that we provide the suffixes here. Because we can support up to 32-bit data over the serial ports, by default we have it set up to be a "long", which is a 32-bit access. If you wanted to program it for 16-bit data, you would be forced to know and use the 16 suffix to make it a 16-bit access, or a "short" access. If you were to use the SPORT device driver, the SPORT device driver is intelligent enough to know that, if you have it configured for 16-

bit, it's going to select the proper macro here to read and write. If you have it set for higher than 16-bit, it's going to use the 32-bit version. Here is a special case code: in this sub-routine, we're initializing the SPORT. As you can see here, we're using one of the frequently used macros - SLEN(15) - so we're setting the serial length portion of this register to 15, which is configuring it for 16-bit data. Then we're priming the transmit register, which is the TX16 - notice the 16 suffix - with a known data pattern that you can see easily on an oscilloscope. And then, finally, we're enabling the sport transmitter by doing an "or/equals" and setting the transmit serial port enable bit (TSPEN). In this routine, all we're doing is looping infinitely, waiting for the transit buffer to empty, and, when it empties, we're going to prime it with new data so that it'll send again once it gets to this part of the code. These two in red, this would be incorrect usage for 16-bit data. If you use the 32 suffix or no suffix, it's going to generate a 32-bit access. This is just to show you the wrong way to do it, the right way to do it, and when to use which ones.

Subchapter 4c: Writing Efficient C

That concludes the section of programming specifically for the Blackfin, and now we're going to take a look at a couple of tips for creating efficient C code for the Blackfin processors. Optimizing C code can be done in many ways. You can decrease code size or increase performance. You can do that globally by setting optimization switches in your "Project Options" dialog box. If you don't have a switch, then optimization is disabled, and that is the default for any new project in the Debug environment. These other switches here tell the optimizer whether to enable for speed - so make it the fastest code possible - or for size - making it the smallest code possible. Then the special one, the "-Ov num", is the switch that you would use to utilize the sliding scale of a balance between speed and size. If you were to do that in Visual DSP++: select any of the projects that you have open, go to "Project Options", go to the "Compile" tab, and this is where you have the ability to set graphically the switches that you just saw on the slide. Enable optimization 100 enables the optimizer entirely with respect to speed, so we're just doing the -O switch. If you wanted to change it all the way to size, it would be the -Os switch, etcetera, etcetera. That's how you would do it from the "Project Options" dialog box. But that's not the only place that you can control optimization levels. You can also do it dynamically in your C source code using #pragmas. These four #pragmas here are probably the most popular and most commonly used, and they are parsed dynamically at compilation time. You have a certain set of optimizations set on your command line, which is controlled by the global options up here. Throughout your code, you could do a "#pragma optimize_for_space" for one piece of code and it'll tell the optimizer: "OK, for this piece of code, I only want to care about how much room it takes up in memory. So, optimize it for space." Then you would go back and, on the next line after that module in the source code, you would do the "#pragma optimize_as_command_line", which sets it back to what it was by the global switches up here. Pragmas allow you to dynamically change optimization control on the fly.

Finally, you can also control on a per-file basis. You don't have to use global scope switches, you can do it on a per-file basis. Again, using Visual DSP++, I have the "MacroListProject" open. If you right-click on any of your source modules there, go to "File Options" and you can see that, by default, it's using the project-wide settings, which is what you had in your "Project Options" dialog box. If you want to change it to file-specific settings, you can click here and you'll see that you get the same sub-tree under the "Compile" options. On the first page - the "General" page - you have the exact same flexibility to control this on a per-file basis.

Now, in terms of general optimization guidelines, and, we'll discuss this furher in future modules, it's important to keep in mind that using mature data types are optimal, because the Blackfin is optimized to perform operations on the native data types. So, if you get a fixed point application that would be preferable to a floating point because the Blackfin doesn't do floating point operations as effectively as fixed point.

The second thing to keep in mind is that the Blackfin core consists of a multiply-accumulator, an ALU, and a barrel shifter. There's no native support for direct division, so in order to do division operations, you're going to use emulated library calls, which consume a lot of cycles. If you can get your division to appear like a divide by 2 or a factor of 2, then it'll appear as a shifter operation, which will be a single-cycle instruction. So, you want to try to avoid division if you can.

The third thing to keep in mind is that the memory architecture is going to pay dividends in the long run. If you use internal versus external memory, you're going have to analyze the ramifications of using external versus internal in terms of latencies involved. If you can use the cache when applicable - that will allow stuff that's residing in external memory to come in on-chip and execute from on-chip memory as if it were on-chip the entire time. Using cache, when applicable, will also help. Having an understanding of when to use DMA to move memory around without the core's intrusion will also help, in terms of getting system performance out of the Blackfin processor.

And finally, the last thing to keep in mind is that the C Libraries that are shipped with Visual DSP++ are already optimized to work best on the Blackfin target. So, it would be better for you to use a memcpy command, for example, than to use a linear copy of buffer A to buffer B, for example.

Chapter 5: Real World Example

Subchapter 5a: OggVorbis Overview

Now that we have concluded the section of understanding how the tools development process works and how to go about coding the Blackfin in C, we're going to conclude with an actual real world example. It's called the OggVorbis Tremor Example. What is OggVorbis Tremor?

OggVorbis is a flash-based decoder with an audio DAC interface. OggVorbis, in itself, is simply an audio compression format. It's fully open and patent- and royalty-free. You can download it off of the web free of charge and use it at your disposal. The Tremor implementation is a free OggVorbis decoder that has been ported from a floating-point decoder to a fixed-point integer arithmetic decoder. And that's the one that we used for the purpose of this demo. This slide is an overview of what OggVorbis is and, as you can see, the red highlighted area here is the decode process, which is the raw music data being parsed, and that it is what our demo today is going to be focusing on.

Subchapter 5b: Optimizing Strategy

In terms of compiler optimization strategies, again, this reference code - the OggVorbis Tremor code - compiles "out-of-the-box". The original code was based on host file I/O, and the Blackfin doesn't have a file structure, so we had to first modify it to work with flash I/O. So, it's getting the streaming file from flash rather than from a PC file system. From there, you can benchmark it using the simulator or the BF537 EZ-KIT Lite, which is what we're using today. Using that benchmark information, you can analyze your computation cycles and your data access cycles based on where stuff is executing in memory. The theory is that you could decrease computation cycles now by optimizing, and you can optimize your memory later when you get to the system-level section of optimization. If you profile your code, you can discover the 80/20 rule, where 80% of your code is only doing 20% of the work; and 20% of the code, which is the important stuff, is doing 80%; and you would want to have that code on-chip, if at all possible, to take advantage of the on-chip memory and the speed at which that can execute. Finally, the global compiler settings - we're going to have it optimized for speed and turn on automatic in-lining. This is the first of a two-step process to increase efficiency. Like I said at the top of this module, this is a nearly zero-effort, two-step process of turning on the optimizer and enabling the instruction cache. The optimizer enables the compiler to generate more efficient code. It does a basic C optimization analysis, produces the better code that can run on the Blackfin better, using multi-issue instructions and hardware loops and considering the pipeline that the Blackfin uses. So just with those two steps, we're going to see a significant efficiency increase. If I go over to Visual DSP++, I have a project set up that utilizes this Tremor code. So, it's a library called libtremor537, and these are all the source modules over here that you'll see. If I right-click and show you the project options, you can see on the general compile tab that we have no optimization enabled whatsoever. If I build that library - this is going to take a couple of seconds because it needs to parse all these files and create the object code and link it and resolve it out into memory. But, it's generating the code that's going to be used by my application code down here, which is called "TremorBF537". Now my library has built successfully, and I'm going to move over to my application code and set it as the active project. Again, look at the project options under the general compile tab, and you'll see that no optimization is enabled at all. If I now build my user code - which is basically just a bunch of printfs that take you through the tremor decode process

and tells you how many cycles it's taking and how many bytes are being decoded - here I am at the top of my code, and if I run this thing, you see the OggVorbis decoder demo showing up; and now it's reading in the input stream, which is called "chopin.ogg" - that's off in my flash memory - and it's loading it into the on-chip memory to read it. Some debug information there, and now it's going through and telling you which encoding process it's using; and now it's actually benchmarking the code. This is going to take a few seconds because, again, this is completely un-optimized code. This is the code as it would appear if you took it off of the web and just brought it into VisualDSP++ and built it and ran it. You see the number of bytes that was decoded and, finally, a cycle count. So, right here, you see the number of cycles in millions is 5,489, so about 5 and a half billion cycles to do this entire decode process using completely un-optimized code.

If I now go back and set my library code to be my active project, and then go into the project options, all I'm going to do is go to the general tab and turn on the "enable optimization" with automatic inlining. You see the 100 here - that means it's optimizing entirely for speed. So I select "OK" here and rebuild my project again with this new optimization standard selected in the project options. Again, because it's employing a different strategy for building the code, it's going to have to parse all the input code and generate the new assembly based on the optimization level that we just set. The library is now done building, so if I scroll back down to my user application and set that back as the active project; and turn on optimization by going to the general tab again, click that and that, and build it. Now I build this - it's taking my application and building the code based on the optimized library that I just built previously. So, again, it's now built and loaded, and I'll make this a little bigger so we can see it better. If I run this guy now, again, it goes into the decoder demo/printf, tells you that it's parsing the input file in Blackfin memory. Again, some more status stuff and now it's actually benchmarking the code as it's running on the board.

Now, if you remember, it was 5 and a half billion cycles the first time. Now it's done the same thing, it's decoded the same number of bytes, and now it's taken roughly half the time. So, you're looking at 2 and three-quarter billion instead of 5 and a half billion cycles, and that was simply by changing the optimization switch.

Now, as I suggested earlier, I'm going to show you that the next step is to turn on the instruction cache in this header file. It's just a pre-processor thing that's commented out currently. So, all I'm doing here is un-commenting this switch, which will also link in code that turns on the instruction cache memory just by writing a couple of registers. If I now rebuild just the user code using the optimized library, only now I'm also using cache memory, you should see another improvement over the 2.75 billion cycles that we just saw previously. That's now built and loaded; and now it's running; and now that it's taken advantage of the cache, now it's saying: "OK, all this stuff that's

already in external memory, we're going to bring it in on-chip for stuff that's used repeatedly and execute from on-chip even though it's actually resolved to external memory." That's what the cache is going to give you.

We're down to the benchmarking section. Again, remember that we were talking about 2 and three-quarter billion cycles previously. You see the same number of bytes decoded, and now you're looking at 817 million cycles. So, you're seeing a 67% improvement over the last iteration with the cache off, and that was a 50% improvement over the first iteration of just coming fresh off the web and running it un-optimized. That is what this slide will show you. On the left hand side here, the percentage that you see is the number of cycles consumed by the core to run the application. So, it was at 100%. When we turned on the optimizer, we got down to 50% of the cycles just by setting a couple of switches. And then, by turning on the instruction cache, which is just writing a couple of registers to take advantage of the cache memory, we got down to 1/3 of that, which was ½ of that. So, a simple two-step process to go from fully consumed, full-bore usage of the core, to 16% of that - pretty powerful given the amount of effort that was required to do so.

Chapter 6: Conclusion

Subchapter 5a: Summary

So that pretty much concludes this module. We walked through the software development process, hopefully gave an understanding of how the linker works so that you can use your knowledge of the linker to move memory around when you get to system optimization later on. We described the linker to help you with the optimization strategies for that process, and we demonstrated the simple two-step process for significant performance increase with almost zero effort. That concludes the demo. For additional information, you can look at the links here on the last slide. Thank you for taking the time to view this demo and good luck with the development process.