Blackfin在线培训课程

课程单元：Blackfin®应用程序开发基础知识
主讲人：Joe Beauchemin

第一章：简介

第1a节：课程安排

大家好，欢迎参加“Blackfin®应用程序开发基础知识”课程。我是模拟器件公司（ADI）的Blackfin应用工程师，我的名字叫Joe Beauchemin。

在今天这个单元，我们讨论的内容均基于Visual DSP++®开发环境。我们将讨论如何将现有的C语言程序代码直接导入Blackfin处理器，并在其上运行。与此同时，我们还要讨论软件开发流程和构建流程，即通过编译器和连接器，生成需要在

Blackfin处理器上运行的可执行代码。我们还将讨论Blackfin编程过程中的一些特殊“陷阱”，并介绍一些在开发流程后期进行系统级优化测试时可以用到的基本代码优化策略。最后，我们将演示一种仅借助优化程序和启用指令高速缓冲存储器的极其简单的两步式优化策略。利用这个优化策略，用户可以轻而易举地实现代码优化。在课程安排方面，我们首先将介绍基于Visual DSP++开发环境的软件构建流程。其中，我们将特别介绍连接器——它可以说是整个开发流程中最为重要的组件。这是因为，只要理解了连接器，你就能在接下来的开发过程中，根据自己的需要，将代码和数据映射到适当的存储空间中，充分实现系统级优化。为此，我们还要介绍连接器的使用说明文件。之后，我们将讨论一些适用于Blackfin处理器的C语言编程技巧，这些是针对Blackfin处理器编程的专门知识。最后，我们将介绍几种创建高效率C代码的方法，并演示一个实例——直接利用从网络下载的代码，在Visual DSP++开发环境下构建应用程序。

第二章：VisualDSP++构建流程

第2a节：C代码导入概述

首先，我们要介绍基于Visual DSP++开发环境的软件构建流程。如果用户导入Blackfin处理器的C代码能够兼容ANSI C语言，那么，就能直接在Blackfin上进行构建并执行这个“现成的”代码程序。由于Blackfin处理器的片上内存空间是有限的，所以，当代码很大时，其运行速度就会很慢。如果代码非常大，超过了内存容量，就会溢出至外接SDRAM存储器，而这个接口显然会降低代码的运行速度。在开始进行编程时，所有优化选项均默认设置为关闭状态。因为在优化状态下，很难对代码进行调试。因此，生成的代码将是未经优化、但功能健全的。此外，处理器将采用默认时钟设置，并且关闭指令高速缓冲存储器。

本幻灯片列出了在软件开发流程中需要使用的文件类型。在最左侧的是源文件（C语言文件或汇编语言文件），这些文件将与C运行头（称为“basiccrt.s”）一同输入编译器。待会，我们将单独讨论这个运行头。编译器和汇编器将生成扩展名为.doj的对象文件。这些对象文件包含了连接器对应用进行解析，并将其映射至相应存储段所必须的所有信息。从根本上讲，对象文件中的所有代码和数据都具备标签信息。连接器将利用这些对象文件、第三方库代码或者用户自己生成的库代码（通称为“DLB文件”）以及连接器使用说明文件（我们称之为“LDF文件”），解析所有这些标签信息，并生成被称为“DXE文件”的可执行文件。实际上，我们将在EZ-KIT Lite™评估板上运行这个可执行文件，所以，今天的课程将以此结束。这个可执行文件可以在目标处理器或模拟器上运行，而在一个“最终应用”中，光有这个可执行文件还不够，用户还需要借助其他组件，将这个可执行文件从外接存储器中映射至适当的内存区。这个文件被称为“引导映像”，即，加载程序文件。这个加载程序文件可能还需要调用一个被称为“引导代码”的组件，即一个单独的DXE文件。因为，如果代码溢出至外接存储器，那么，硬件就必须在向该存储器导入数据之前，对其进行初始化。因此，在最终完成的应用软件中，必须将这个“初始化代码”或“引导

代码”置于可执行代码之前。前面我已经说了，今天的课程将从源文件开始，到可执行文件结束。

第2b章：编译器基础知识

在上一张幻灯片中，我曾提到名为basiccrt.s的C运行头。这个组件有什么用？C运行头负责设置处理器的C运行时间、堆栈指针和启用周期计数器；如果需要使用数据和/或指令高速缓冲存储器，C运行头还提供了一个设置向导，引导用户配置高速缓存；此外，用户还可以利用C运行头，更改芯片的时钟频率和电压设置。用户可以在开发后期，通过“Project Options（项目选项）”窗口，修改C运行头。此外，当用户打开项目向导时，C运行头也将启动，如左侧窗口所示。如果在这个对话框中选择了“Yes（是）”，并单击“Next（下一步）”，那么，这个启动向导将显示一系列选项设置界面，引导用户完成高速缓存配置和所有相关设置。

那么，这个软件构建流程将从哪里开始入手？首先，是位于左侧的C语言文件，即，cFile1.c。其中，我只定义了一个简单的主函数，以及一个被称为func1的函数。我们将这个C语言文件导入C编译器，然后，这个编译器将生成一个扩展名为.s的中间文件。在这个过程中，编译器负责将C源代码转译成Blackfin处理器运行这个程序所需的汇编代码。汇编器利用这个汇编代码，生成对象文件（如右侧方框所示），其中包含标签信息。这个是名为“program（程序）”的对象段，对于所有用户未明确指定其代码段名称的代码，编译器将默认向其分配“program”代码段名称。这些是我们刚才定义的主函数和func1函数的汇编代码。此外，下面是一个堆栈段。这个堆栈段里保存了本地变量和其他帧指针及堆栈指针信息，以及C运行时环境自动备份和恢复的寄存器。

在上一张幻灯片中，我们介绍了名为“program”的编译器生成对象段。这张幻灯片列举了几种编译器自动生成的标签类型。凡是与代码相关的数据，都被命名为“program”，凡是全局公告的数据都被命名为“data1”。在这张幻灯片列出的标签类型中，这两个代码段名称最为重要。因为，连接器要利用这些标签，解析用户输入的所有对象文件，以生成可执行文件。用户也可以自定义代码段名称，这张幻灯片举例说明了如何命名代码段。这里，在左侧的源代码中，我们仅通过扩展名“section（”sdram0”）”，告知编译器，希望将这个整数阵列标签为“sdram0”，而不是将其解析为通常会被映射到外接SDRAM存储器中的“data1”代码段。同样地，用户可以灵活地命名代码。在本例中，我们将被称为“bar”的函数的代码段命名为“L1_code”。然后，这些源代码将经由编译器和汇编器，生成右侧所示对象文件。在这些对象文件中，你会看到“sdram0”标签和一个整数阵列；下面是一个名为“L1_code”的对象段，这些是“bar”函数的汇编代码；再下来，是堆栈段。

在上一张幻灯片中，我们之所以要使用标签“sdram0”和“L1_code”是因为，在连接器使用说明文件中，我们已经指定了用户可用于替代编译器生成的标准代码段名称的代码段名称。也就是说，“sdram0”适用于命名希望将其保存到外接存储器中的任何代码或数据，而“L1_code”则适用于命名明确希望将其映射至片上内存的代码，

相比于从外接SDRAM存储器上执行的代码，这种代码的运行速度更快、效率更高。最后两个标签是“L1_data_a”和“L1_data_b”，这两种代码将分别映射至两个片上数据存储体。

第2c节：连接器基础知识

在理解了编译器生成对象文件的过程之后，就需要了解连接过程。连接过程非常重要，因为在后面的开发阶段，当用户实际将频繁调用的代码段保存到速度更快的存储器中时，需要进行系统级优化。那么，连接器的作用是什么？我在前面已经提过，连接器负责利用已编译或已汇编的对象文件，即.doj文件，以及连接器使用说明文件和所有可用的库代码，创建一个完全解析的Blackfin可执行文件。这个DXE文件可以在模拟器上运行，也可以通过仿真器或调试监视器，在目标处理器上运行。在这里，连接器需要使用的一个重要文件是LDF，其中定义了VisualDSP++的硬件系统。这个文件阐明了用户将在项目中使用的处理器，以及这个连接器在对应用程序执行解析时可以使用的片上内存及外接存储器。如果用户并未定义存储器，那么，连接器就无法使用存储器。因此，用户必须理解存储映射以及连接器如何充分利用各种存储段，实现最优性能。说到存储器，应当指出，Blackfin处理器的存储器采用了分级模式，效率极高。也就是说，我们所说的L1内存，是一级片上内存。L1内存是以核心频率运行的速度最快的内存，应当将最经常调用的代码保存在L1内存中。此外，我们还有一些处理器，如BF535和BF561双核，具备二级片上内存，其核心频率最快超过了2.0 GHz，略低于L1内存。除此之外，离芯片最远的存储器是外接同步存储器——SRAM或SDRAM存储器。可以称之为“片外存储器”或“L3存储器”，你们可能听说过这两个术语。

所有这些存储区域都有自己的地址范围，LDF文件中包含了所有这些信息，稍后我们将详细介绍。另外，必须指出，处理器将自动接入这些存储器。如果你已经将一个指针设置为指向SDRAM，那么，你只需要调用该指针，然后，处理器将自动接入SDRAM存储器。但前提条件是，你必须已经配置了这个硬件。

第三章：控制连接器

第3a节：连接过程概述

现在，我们要来讨论连接器的实质，也就是理解连接器使用说明文件。下面这个亮蓝色的方框就代表LDF文件。LDF文件负责控制连接器，将所有这些输入文件、对象代码、库代码以及标签，解析成输出代码段中的可执行文件，如右侧方框所示。

第3b节：连接器使用说明文件（LDF）

LDF的作用是什么？LDF阐明了将程序映射至存储器的方式。利用LDF，用户可以规定分别将各段程序映射到哪个存储段中。勿庸置疑，每个项目都必须具备一个LDF文件。如果没有，那么，VisualDSP++的安装路径中就会有大量的临时LDF文

件，连接器将根据用户使用的处理器以及编程语言，默认选择这些LDF文件。在LDF文件的顶部，有一些全局命令。你必须首先利用“ARCHITECTURE”命令，设置目标处理器，然后利用“MEMORY”命令，定义存储布局。接下来，我们要详细讨论这个过程。此外，用户还必须阐明所有可用的系统内存。如果你不想使用模板中提供的LDF文件，而要自己编写命令，或者引用数据表中的系统内存映射，那么，你可以定义存储段，但不要跨越任何边界或者使存储段地址范围超出规定。接下来，这些是全局命令。在本例中，这个“ARCHITECTURE”命令的意思是，我们的项目将在BF537处理器上运行。下面是“MEMORY”命令，其中定义了存储段。左侧列出的是存储段的名称，这些都是我们提供的模板中的LDF文件所包含的默认名称。用户可以任意命名，不过，这些名称是默认名称。然后，是定义存储器的类型，通常将其定义为Blackfin处理器的RAM存储器。接下来，要定义开始地址和结束地址。这些都是用户可自定义的，这里举出的例子是BF537的地址范围。最后，还有一个“WIDTH”命令。对于Blackfin处理器而言，这个值总是设为8，因为这是一个一字节宽的可寻址存储空间。

在存储段定义完毕之后，接下来，连接器使用说明文件将通过“SECTIONS”命令，阐明存放输入代码段的存储段。这里，在“INPUT_SECTIONS”命令的上面还有两级命令，我们将在下一张幻灯片中举例说明这两级命令。“INPUT_SECTIONS”命令阐明了这些输入文件中的哪些内容会映射到我们当前所在的输出代码段中。因此，“INPUT_SECTIONS”命令的使用方法是，在这个命令中输入*file_source（源文件）*和*input_labels（输入标签）*。*file_source（源文件）*就是一系列文件名，如，.doj文件或者能够扩展为一系列.doj文件的LDF宏指令。这个命令的含义是，对这些.doj文件进行解析，并搜索其中的输入标签，而*input_labels（输入标签）*就是一系列代码段名称。例如：“INPUT_SECTIONS {main.doj(program)}”。在这个例子中，这个命令告知连接器，对main.doj对象文件进行解析，找出所有标签为“program”的代码段，并尽其所能将这些代码段映射至这个输出代码段。这张幻灯片显示的是默认模板中提供的LDF文件。这里，我们已经为处理器设置了我们上面讨论的“SECTIONS”命令。现在，需要定义DXE代码段名称。这些名称只表示这些代码属于.map文件——VisualDSP++附带的调试工具之一。用户可以打开一个.map文件，其中将阐明输出代码段的名称并列出其中的所有内容。在这里，绿色显示的内容至关重要。这些是经汇编的代码，或者是.S文件的内容，即，编译器生成的汇编代码。你可以看出，“data1”是默认的全局数据段，编译器将所有的全局数据都命名为“data1”。连接器将在所有对象文件中搜索“data1”数据段，并将其映射到右侧定义的存储段中。这些存储段实际就是我们在上一张幻灯片中定义的存储段。这里，值得注意的是，将输入代码段映射至多个不同存储段的能力，我们将在介绍什么是LDF文件时，讨论这一点。“data1”和“program”代码段拥有专用的片上内存段——MEM_L1_DATA_A和MEM_L1_CODE，不过，这两种代码段也可能被映射至外接存储器。

接下来，我们就可以看到如何实现这一点。不过，我首先要讨论宏指令。

第3c节：连接器的工作原理

在前面的幻灯片中，你已经看到了“$OBJECTS”命令。“$OBJECTS”就是LDF文件中定义的一个宏指令，其中包含了CRT，即我们在开始时介绍的通过basiccrt.s生成的C运行时对象文件。然后是“$COMMAND_LINE_OBJECTS”，这些通常都是利用用户项目中的输入源代码生成的对象文件。受连接器的工作方式的影响，这些对象文件的排列顺序极为重要，并且项目窗口中显示的源文件顺序就是这个顺序。这里仅简单举例说明，这是VisualDSP++，会话已经发起，这是面向BF537-EZ-KIT的项目，在左侧，我打开了“ModuleFileProject”项目，其中有“function 1”、“function 2”、3、4、5，和“ModuleFileProject.c”六个源文件，在$COMMAND_LINE_OBJECTS宏指令中，它们也按这个顺序排列。这个顺序为什么如此重要？用户可以编写宏指令，并将其映射到特定的输出代码段，我们将在后面详细介绍这方面内容。

连接器的工作原理是什么？前面我说过，可以将对象段映射至多个存储段。连接器的工作方式是从左至右、从上到下，对LDF文件进行解析。也就是说，连接器将彻底搜索这些.doj文件，查找用户为其定义的标签信息。然后，它会考虑，“这个对象段能不能放到当前处理的存储段里。可以的话，就放进去；如果不行，就得暂时搁在一边。然后要从头到尾彻底检查是不是所有的代码段都已妥善地映射到相应的存储段中。”如有不妥之处，就会导致连接器错误。此外，如果片外存储器不能为从内存溢出的对象提供容身之所，并且用户并未为其定义其他输出存储段，那么，这些未能映射的源代码将导致连接器错误，最终应用将无法使用这些源代码。

下面，举例说明宏指令的工作方式。如果输入文件是“main.c”和“file2.c”，并且，使用了$OBJECTS宏指令（即，前面的幻灯片中所举默认模板LDF提供的宏指令），那么，对连接器而言，它收到的命令就是分别对“main.doj”和“file2.doj”文件进行解析，并查找这些标签。连接器将解析“main.doj”文件，找出所有标签为“data1”的代码段，并试图将其映射到MEM_L1_DATA_A内存段。如果一切顺利，那就万事大吉；如果不行，就得暂时将这些代码段放在一边，然后对“file2.doj”文件进行解析，并找出所有标签为“data1”的代码段。在完成对MEM_L1_DATA_A内存段的映射任务后，连接器将继续在MEM_L1_CODE内存段上，对对象文件中标签为“program”的代码段执行同样的处理。这个任务也完成之后，它将进入这个SDRAM存储区。在本例中，我们已经定义，将所有从内存溢出的“data1”和“program”代码，保存到SDRAM存储器提供的广阔的外接存储空间中。因此，在这里，连接器会将前面两个任务中未能映射的所有代码保存到这个外接存储器中。所以，如果用户打算导入Blackfin处理器的C文件非常庞大，那么，这个巨大的文件可能只会生成一个包含所有代码的“program”代码段，而片上内存将无法容纳这个代码段。这样一来，连接器就会将整个代码段映射到外接存储器中，使得最终应用必须从这个SDRAM存储器上执行一切代码。只要理解了连接器使用说明文件，用户就能将代码拆分为多个模块，并且灵活地将代码存放到不同的存储段，从而充分利用存储架构，实现最优效率。

第3d节：优化连接器

如何利用LDF文件实现优化？用户可以利用一个名为“eliminate unused objects（消除未使用对象）”的选项，优化对象文件的大小。步骤如下：首先，发起Visual DSP++会话；鼠标右键单击项目名称；屏幕将出现“Project Options…（项目选项……）”对话框；在连接选项卡下，有一个名为“Elimination（消除）”的子选项卡；在其中，单击“eliminate unused objects（消除未使用对象）”选项。这样，连接器就会将随用户调用的部分库代码而输入的所有其他库代码或诸如此类的代码，全部删除，从而避免将其映射到有限的存储空间中。这就是“eliminate unused objects（消除未使用对象）”特性的工作方式。在性能优化方面，用户可以利用连接器的LDF文件，如我先前所说，根据自己的需要，将代码灵活地保存到不同存储段中。如果想将一个函数保存到L1内存中，则可以使用代码行：“section (“L1_code”) void my function”。如果想明确地将一个阵列映射到Data A片上内存段中，则可以使用下面这个代码行。此外，在以后的系统优化技巧或Blackfin优化技巧课程单元也将用到LDF文件。例如，如果用户使用了一个向量乘法，需要从两个不同的存储体中调用数据，以避免调用冲突，在这种情况下，用户就可以利用LDF文件中的这些代码行，明确地将输入向量“mult1”保存到Data A片上内存段中，而将输入向量“mult2”保存到Data B片上内存段中。

除此之外，用户还可以灵活地创建一个特殊的代码段名称；在本例中，我将其命名为“Map1st”。基本上，我的目的是通过“Map1st”这个标签，告知连接器，无论如何，必须优先将这个代码段保存到片上内存中。待会，我将举例说明。最后，利用下面这个代码行，用户还可以创建特殊的宏指令列表，使连接器按列表的顺序，依次对输入文件进行解析。接下来，我将举几个例子。现在，我启动的这个项目名称是“SingleFileProject”；这里，我已经打开了源代码文件。从本质上讲，这是一个不断调用这五个函数的无限循环，所有这些函数都定义在这一个代码模块中。这个头文件提供的只是所有这些函数的原型。如果用户构建并运行这个代码模块，那么，它会将所有代码映射到片上内存中。这个代码模块并不大，不会溢出至外接存储器或类似的存储空间。这个代码模块构建完成后，用户可以单击反汇编窗口上的浏览按钮，查看反汇编窗口；其中包含了用户调试代码中的所有符号。在这个窗口里，用户可以任意单击，并键入需要查看的内容。这里，我键入了“fun”，于是，窗口转至“function 1（函数1）”标签，我们看见的函数名称是“function 1”、“function 2”、3、4和5。连接器将函数1、2、3、4和5依次映射到片上内存中。0xFFA0____（开始地址）是片上一级代码。连接器在生成这个可执行文件时，是按函数1、2、3、4和5的顺序进行解析，因此也按此顺序进行映射。如果用户的输入文件很大，那也没关系，只不过，需要将这个输入文件设置为当前项目；这也为利用连接器进行下一步优化做好了准备。采用代码模块化方法，我将在调用同样的函数时产生同样的无限循环延迟的主函数文件，按函数拆分为不同的文件。于是，“function1.c”的作用没有变，只不过它有了自己的专用文件。然后，单击上面这个“build all（全部构建）”图标，进行构建，那么，应用运行的结果应当是一样的，因为我们并未更改向连接器传达的命令。它仍然对同样的信息进行解析，只不过，它是按照左侧

所示的顺序，不断地对这些文件进行解析。此外，函数1、2、3、4和5仍然顺序保存在前缀为0xFFA0____的片上内存段中。

理解了代码模块的工作方式后，再来看看下面这个例子。本例所用项目名称为“AlternateSectionCodeProject”。我现在将其设置为我的当前项目。这里，我要使用这个主函数。其实就是第一个例子中的主函数，其中所有的函数都是在同一个源代码中定义的。不过，在这个项目中，没有将所有函数都解析映射至L1片上内存。现在，我打开了头文件，并且在函数原型前面添加了这个“section(“sdram0”)”，从而告知连接器，我希望将函数4和函数5保存到外接存储器中。接着，单击构建图标，构建这个项目。同时，我退出这些源文件窗口。现在，再单击浏览按钮，同样键入“fun”，查找函数1、2、3、4和5，可是，现在只有函数1、2和3依然保存在内存中，而函数4和5则已映射至外接SDRAM存储器中，其地址前缀为0x00_____。也就是说，函数4和5已顺序保存在外接存储器中，而函数1、2和3则保存在内存中。通过这种方法，可以查看连接器生成的文件，并且在进行系统级优化时，可以轻松改变存储代码的位置。

再举一个例子。在这个部分，我们实际上是修改LDF文件而不是源代码。在这里，我打开LDF文件，并向下滚动查看，这些都是VisualDSP++工具使用的预处理程序。现在，我找到了“Map1st”代码段名称。这个名称是我自定义的，意思是我希望连接器无论如何必须将这个代码段放到片上内存中，而且只能放到片上内存中。现在，我更改了INPUT_SECTIONS，因此，连接器会对输入行中列出的所有对象文件进行解析，搜索“Map1st”标签，并将这些代码段映射至MEM_L1_CODE片上内存区。那么，这个修改对源代码有何影响？打开头文件，找到源代码；现在，除了明白无误地将函数4和5保存到外接SDRAM存储器中，我还指示连接器找到“Map1st”标签，并将其分配给函数2。这个代码构建完毕后，我们应当看到函数4和5位于外接SDRAM存储器中，而函数2则应在L1内存中排在第一位。再单击浏览按钮，查看为所有这些函数定义的标签。你看，函数2在内存中的位置低于函数1。连接器首先径直将函数2放到L1内存中，在函数2映射*完毕后*，再依次映射函数1和函数3。和上一个项目一样，函数4和5保存在片外存储器中。

最后，用户还希望能够利用宏命令，优先处理部分输入文件。在这个例子里，我也要采用代码模块化方法。我定义了一个主函数，调用不同输入文件中定义的所有函数。在定义宏命令时，我和默认LDF选择了相同的位置。这个$OBJECTS命令包含了C运行时间和C运行时间正确运行所必需的其他一些初始化对象文件。不过现在，我们定义了两个宏命令：$MY_L1_OBJECTS和$MY_SDRAM_OBJECTS。通过这些命令，我告诉连接器：我明确希望将函数2、1和3，以及我的主函数，映射到片上内存里。并且希望将函数4和5，映射到片外SDRAM存储器中。这是一个功能强大的工具，因为，如果要将一两个极其重要的文件保存到片上内存中，那么，用户只需要在这里进行更改就可以一劳永逸了，而如果选择其他方式，用户就必须为需要映射的每一行代码或每一段数据贴上明确的标签。除了定义宏命令，用户还必须在负责执行映射的SECTIONS命令中使用宏命令；例如，使用

$MY_L1_OBJECTS宏命令，要求连接器搜索“program”标签，并将其映射到MEM_L1_CODE内存段中。同样地，在下面的SDRAM部分，你可以看到，我已经做出修改，要求连接器将所有的$MY_SDRAM_OBJECTS代码段，即，“program”、“data1”和“const data”，映射到片外存储器中。如果我继续构建这个项目，我们将看到这个宏命令对整个系统的影响。现在退出这些源文件窗口，单击浏览按钮，然后找到函数。然后，你会看到，宏命令定义已经完成，它要求首先将函数2映射到内存中，也就是说，在这些函数中，0x_____174是最低位的内存地址，函数1和函数3顺序排在其后，而函数4和5则保存在片外存储器中。

上面的几个例子演示了如何利用连接器使用说明文件将代码保存到不同的存储段中，在以后的课程单元里，将介绍如何进行系统优化，到时就可以充分利用这个工具了。

第3e节：Expert Linker（连接器专家）

现在，你已经了解了连接器使用说明文件，接下来，我们将介绍VisualDSP++中的另一个工具：Expert Linker。Expert Linker具备一个图形化用户界面，允许用户控制LDF文件。实际上，用户可以利用抓取和延伸等操作，更改我们在LDF文件的MEMORY命令中定义的存储空间大小。也就是说，用户可以抓取两个存储段之间的边界，通过拖动边界，以此消彼长的方式调节这两个存储段的容量。此外，用户还可以通过鼠标拖放操作，将代码段放到不同的输出存储段中。在这个Expert Linker界面中，左侧所示为可供使用的对象文件，右侧所示则为存储映像。通过鼠标拖放操作，可以实现与手动编辑LDF文件同样的效果。Expert Linker的另一个绝妙之处就是它提供了关于堆栈和堆阵的水印信息，而C运行时环境就需要利用堆栈和堆阵。用户可以运行刚刚编写完成的代码，并通过Expert Linker查看这些代码实际使用了多少堆栈空间和/或堆阵空间。这样，用户就能知道代码在运行时动态需要的最高存储空间，并修改堆栈段和堆阵段的深度，为其他数据和代码提供更多存储空间。Expert Linker的另一个重要特性就是，用户更改的只是它底层的一个文本文件。用户在设置界面上以直观的方式做出更改，而实际被修改的是与之相关联的文本。当用户更改或重新调节存储配置时，实际上就是更改LDF文件的MEMORY命令中的开始地址和结束地址。如果用户通过拖放操作移动代码段，那么，用户其实就是在修改SECTIONS命令，这个操作的含义是：将这个对象代码映射到这个存储段。Expert Linker不会毫无选择地更改以前存在的所有内容，它只是按照用户在Expert Linker会话期间做出的修改，对LDF文件进行相应的更改。用户每一次启动Expert Linker时，它都会反映出用户通过文本编辑做出的更改。用户利用Expert Linker做出的更改只是修改其底层文本，刚才我们已经解释过了。

利用Expert Linker，我们除了可以更改LDF文件，还可以创建LDF文件。这张幻灯片将演示创建方法。在下拉菜单中，依次单击“Tools（工具）->Expert Linker（连接器专家）->Create LDF（创建LDF文件）…”，屏幕将出现一个对话框，提示用户输入一个LDF文件名称和编程所用的语言。用户必须知道，C++、C和汇编语言这三种语言的级别高低。如果要混合使用C++和C源文件，那没问题，不过对于LDF

文件，用户必须阐明使用的是C++语言，因为C++包含了诸如构造器和破坏器等细微差别，而C语言则无法实现这些功能。同样地，如果要混合使用C语言和汇编语言，那么，用户应当选择C语言模板，因为C语言模板可以定义运行C语言应用程序所必需的C运行时间。模板LDF在安装路径中的这个目录下：Blackfin子树下的“\ldf”目录，模板LDF的后缀包括_ASM、_C和_CPP（适用于C++）。例如，如果用户要使用面向BF537的C语言LDF文件，那么，就要选择这个选项。像这样，依次单击“Tools（工具）->Expert Linker（连接器专家）->Create LDF（创建LDF文件）…”，不错，出现了一条错误信息，提示说已经存在一个LDF文件。在任何一个项目中，都只能有一个LDF文件。如果我要生成一个新的LDF文件，那么，屏幕将显示这个LDF设置向导。现在，我要取消并退出这个向导，我不打算实际演示一遍设置过程。好了，关于Expert Linker的介绍到此为止。

第4章：面向Blackfin处理器的C语言编程

第4a节：概述

现在，我们要来谈谈工具开发流程，接下来我们将专门集中讨论适用于Blackfin处理器的C语言编程。Blackfin处理器拥有十分丰富的外设组合，没有任何其他架构、微处理器或PC可以与之媲美。要利用外设，用户必须理解什么是存储映像寄存器（MMR）。每个外设都有一个专用MMR，以便用户配置和使用该外设，例如串行端口、UART（通用异步收发器）和CAN（控制器局域网）等等。只要理解了MMR，用户就可以根据自己的需要，轻松自如地使用MMR。必须指出，如果使用了系统服务程序或设备驱动程序，那么，下面几张幻灯片中介绍的处理将自动完成。如果用户使用了这些特性，就不必担心MMR的编程问题。

第4b节：利用头文件

如果用户选择使用自己编写的程序，则可以利用我们提供的这些头文件，轻松完成。defBFxxx.h头文件的基本作用是将所有存储映像寄存器映射至存储器中的相应地址。我的意思是，处理器可以通过这个明确的指针，找到EMAC_STAADD寄存器。这个功能非常重要，因为，用户需要调用这些寄存器，并且要使编写的代码清晰明了。对EMAC_STAADD寄存器编程可比对0xFFC03014存储段编程容易得多，因为后者会使调试过程相当痛苦。除了定义存储空间，头文件还定义了《硬件参考手册》中定义的那些寄存器中的所有位。用户不一定将寄存器设置为特定的16位值，也可以将其设置为一些不同位单元的“/=（或等于）”值，并且用户编写的代码是采用硬编码数字，以明示而不是默示的方式完成设置。

除了定义寄存器中的位以及寄存器本身，我们还提供了一些面向常见任务的非常有用的宏命令。如果用户想在这个寄存器中设置物理层地址，用户不必编写数字，对其进行掩码以符合字段要求，然后将之移动至寄存器中的适当位置——用户只需使用这个宏命令就能完成这一切。这些头文件都定义在“\include”目录下，就在这里。这些是低级头文件。我们还提供了更高级的C头文件，它包含了刚才介绍的那

种低级头文件的所有内容，存储映像、地址和位。此外，它还创建了一个特别适用于C语言编程的宏命令。利用经典的*p前缀符号，用户可以调用整个存储映像，只要在*p之后以大写字母输入希望调用的存储器的名称。这一点非常重要，因为在Blackfin处理器上，用户接入的寄存器必须符合要求。对于一个16位的MMR，用户必须在程序中注明是16位存储器。对于一个32位MMR，用户必须在程序中注明是32位存储器。如果用户不能严格遵守这些规则，就会产生一个硬件错误，导致读或写操作出错。这里，我要提醒大家，系统服务程序和设备驱动程序将自动完成这个处理。但是，如果用户要自己进行编程，那么，就应当知道，这是硬件的工作方式。也就是说，这是一个C语言编程中的特殊“陷阱”，一个串行端口自身存在的“陷阱”。串行端口硬件是Blackfin处理器上唯一既可以支持16位数据又可以支持32位数据的硬件，这取决于用户对硬件的配置。如果用户将其配置为32位，那么，当用户以16位数据方式接入时，将会无效，反之亦然。如果用户使用了错误的宏命令，那么，代码依然可以构建并运行，但是，其运行结果却会令用户大失所望。因此，如果用户要写或者读SPORTx_TX或SPORTx_RX数据寄存器，不论是通过SPORT0还是通过SPORT1，用户必须清楚，我们为此提供了一个后缀。由于Blackfin处理器的串行端口最高可以支持32位数据，所以，我们将32位数据默认设置为“long（长数据）”。如果用户希望将串行端口编程为支持16位数据，那么，用户必须使用“16”后缀，将其注明为16位数据接入或者说“short（短数据）接入”。如果用户要使用SPORT（串行端口）设备驱动程序，那么，聪明的SPORT设备驱动程序可以自动完成这个设置。如果用户已经将串行端口配置为支持16位数据，那么，SPORT设备驱动程序将选择适当的宏命令来进行读和写。如果用户将串行端口设置为支持高于16位的数据，那么，它将使用32位数据。这里举出的是一个特殊情况代码：在这个子程序中，我们对SPORT进行初始化。如你所见，我们使用的是一个经常使用的宏命令——SLEN(15)，也就是说，我们将这个寄存器的串行端口长度设置为15，而这个宏命令会将其配置为支持16位数据。然后，我们向发射寄存器，即TX16——注意后缀“16”，传送一种已知数据模式，通过示波器可以清楚地看到这种数据模式。最后，我们将通过“/=（或等于）”命令和设置发射串行端口的启动位（TSPEN），启动这个串行端口发射器。在这个子程序中，我们实现了一个无限循环，等待发射缓冲区为空时，向其传送新的数据，然后在程序循环到这段代码时，串行端口发射器将再次发送。对于接入16位端口，这两行红色的代码是错误的，如果使用了“32”后缀或未加后缀，那么，这段代码的意思就是接入32位端口。这些例子旨在说明如何使用后缀及其适用情况。

第4c节：编写高效率的C代码

上面介绍了针对Blackfin处理器的C语言编程，现在，我们要讨论几种创建面向Blackfin处理器的高效率C代码的编程技巧。可以通过多种方式优化C代码。可以缩小代码长度，或者提高运行速度。用户可以在“Project Options（项目选项）”对话框中，设置优化选项，实现全局优化。如果没有选中任何选项，则表示禁用优化特性，在调试环境下，这是所有新项目的默认设置。这些选项提供了不同的优化功能，例如，优化运行速度——实现尽可能最快的代码；或者，优化代码长度——实

现尽可能最小的代码。在下面这个特殊的选项“-Ov num”中，用户可以通过滑动块，选择运行速度和代码长度的折衷优化。例如在Visual DSP++中，选择一个打开的项目，单击“Project Options（项目选项）”，打开对话框，再单击“Compile（编译）”选项卡，打开这个图形化界面，然后用户就可以直观地方式设置各种优化选项，如刚才那张幻灯片所示。选中“Enable optimization 100（完全优化）”，优化器将全力以赴优化运行速度，所以我们刚才选中了“-O”选项。如果用户转而希望优化器竭尽全力优化代码长度，那么，就选中“-Os”选项，如此等等。用户不仅可以在“Project Options（项目选项）”对话框中，通过这些设置实现优化；用户还可以在C源代码中，利用“#pragmas”代码，动态实现代码优化。这里列出的四种“#pragmas”代码是最流行也最常用的，处理器将在编译时间动态解析这些代码。用户可以在命令行中设置一组优化设置，并通过上面这些全局选项控制这些优化设置。在整个C代码中，用户可以为某段代码编写一行“#pragma optimize_for_space”代码，告知优化器：“好吧，对于这段代码，我只在乎它占用了多少存储空间。所以，对它进行代码长度优化。”回过头来，用户又可以为刚才那段代码后面的代码编写“#pragma optimize_as_command_line”，将优化器恢复为上面的全局选项的设置。利用“#pragmas”代码，用户可以迅速地动态更改优化设置。

此外，用户还可以逐一对源文件进行优化设置。不使用全局选项，而是对每个源文件单独进行优化设置。举例说明，在Visual DSP++中，打开“MacroListProject”项目。鼠标右键单击任意一个源文件，打开“File Options（文件选项）”对话框，其中，已默认设置为采用适用于整个项目的设置，也就是“Project Options（项目选项）”对话框中的设置。如果要将其更改为采用针对源文件的设置，那么，单击此处，然后，在“Compile（编译）”选项下，将出现同样的子树。在第一个界面，即“General（一般设置）”选项卡，用户可以同样灵活地控制每个源文件的优化设置。

现在，介绍代码优化的一般原则。在今后的课程单元，还将进一步讨论这个问题，必须老记，最好使用本机数据类型，因为Blackfin处理器业已针对本机数据类型完成了性能优化。也就是说，固定点应用优于浮点应用，因为相比于浮点运算，Blackfin处理器执行固定点运算的效率高得多。

第二点必须牢记的是，Blackfin处理器核心包含一个乘加器、一个ALU（运算器）和一个桶形移位器。Blackfin处理器不能本机支持直接进行除法运算，因此，要进行除法运算，就需要调用仿真库，而这会耗用大量周期。如果用户可以将除法运算转换成除以2或者2的因素的运算，那么，这个除法运算就会看起来像是一个移位运算，即一个单循环指令。因此，用户需要尽力避免使用除法运算。

第三点必须牢记的是，有效利用存储架构可以产生长期效益。如果用户综合利用了片上内存和片外存储器，就需要分析使用外接存储器与片上内存在运行时延方面的不同影响。如果可以在适当情况下使用高速缓存，就能将保存在外接存储器中的代码和数据临时存入片上内存，而处理器也会将这些代码和数据完全当作片上内存中保存的内容来进行处理，从而提高运行速度。此外，如果懂得在什么情况下利用

DMA（直接内存存取）来灵活接入内存，而无需处理器核心的干预，也有助于提高Blackfin处理器的整体性能。

必须牢记的最后一点是，VisualDSP++附带的C库已经针对Blackfin处理器完成了性能优化，因此，例如，使用 “memcpy”命令，比从缓冲区A线性拷贝至缓冲区B更高效。

第五章：实例说明

第5a节：OggVorbis概述

现在，我们已经完全理解了工具开发流程的工作方式，以及如何对Blackfin处理器进行C语言编程。接下来，我将举一个实际应用的例子。这个例子的名称是“OggVorbis Tremor”，那么，“OggVorbis Tremor”是什么？

OggVorbis是一个基于闪存的解码器，具备一个音频DAC接口。OggVorbis本身只是一种音频压缩格式，是一种完全开放的免专利费和版税的代码。用户可以直接从网络免费下载这个代码，并且随意使用。Tremor实现是一个免费的OggVorbis解码器，是一个在浮点解码器的基础上构建而成的固定点整数运算解码器。我们在本例中使用的就是这种解码器。这张幻灯片显示的是OggVorbis的工作流程图，这个红色点线框出的区域就是解码过程，即，解析原始音乐数据的过程。接下来的例子将重点演示这个过程。

第5b节：优化策略

就编译器优化策略而言，这个参考代码——“OggVorbis Tremor”代码，是利用“现成”的代码完成编译的。原码基于主文件I/O，但是，Blackfin处理器不具备文件结构，因此，我们首先要将这个原码修改为支持闪存I/O。也就是说，它从闪存而不是PC文件系统，获取流式文件。然后，用户可以利用模拟器或者BF537 EZ-KIT Lite评估板，对这个代码进行基准测试。我们今天将要使用的是BF537 EZ-KIT Lite评估板。根据基准测试的结果，可以对不同的代码存储映射方案，分析其运算周期和数据调用周期。理论上，如果可以通过优化代码减少运算周期，那么，就可以在进行系统级优化时，优化存储配置。如果对代码进行分析，就会发现80/20定律，即，80％的代码仅完成了20％的工作，而20％的代码，也就是最重要的代码，完成了80％的工作。你当然希望尽一切可能，将这20％的代码保存到片上内存中，以充分利用片上内存空间以及在片上内存执行代码所能实现高速率。最后是全局编译器设置，我们将对其进行优化，并启用自动内联。这是一个两步式优化策略的第一步。我在课程开始时已经说过，这是一个轻而易举的优化过程，只有两个步骤，分别启用优化器和指令高速缓存。通过启用优化器，可以使编译器生成更加高效的代码。优化器会执行一个基本的C代码优化分析，然后利用多发布指令和硬件环路，同时考虑Blackfin处理器使用的处理管线的情况，生成可以在Blackfin处理器上更快速地运行的更高效的代码。只要进行这两步优化，就可以大幅提高代码效率。现

在，进入VisualDSP++，这里我已经设置了一个利用这个Tremor代码的项目，库的名称是“libtremor537”，这些是所有的源文件。现在，我单击鼠标右键，打开“Project Options（项目选项）”对话框，在“Compile（编译）”下面的“General（一般设置）”选项卡中，你可以看到，尚未启用任何优化选项。现在，我构建这个库，这需要几秒钟的时间，因为必须解析所有这些源文件，创建对象文件，并将其解析、映射至内存。之后，下面的这些名为“TremorBF537”的应用代码将使用这个过程生成的代码。现在，我的库已经构建完毕，接下来，我将下面这个“TremorBF537”应用代码设置为当前项目。再次打开“Project Options（项目选项）”对话框，在“Compile（编译）”下面的“General（一般设置）”选项卡中，同样是尚未启用任何优化选项。现在，构建我的用户代码。这个代码基本上只是大量的“printfs”代码，将执行tremor解码过程，并且计算出解码过程耗用了多少个周期以及对多少个字节进行了解码。这个就是我的代码，如果我运行这个代码，这个OggVorbis解码器将开始执行解码。现在，这个解码器正在读取输入数据流——它将这个被称为“chopin.ogg”的数据流从我的闪存加载到片上内存，再进行读取。这些是调试信息，现在正在进行调试，这些信息表明了正在进行的解码过程；实际上，这个过程就是对代码进行基准测试。这也需要几秒钟的时间，因为这是一个完全未经优化的代码。我只是将这个从网络下载的代码直接导入VisualDSP++，然后进行构建和运行。这是解码的字节数量，最后这个是周期数量，你可以看见，以百万为单位的周期数是5,489，利用完全未经优化的代码执行整个解码过程要耗用将近55亿个周期。

现在，我重新将我的代码库设置为当前项目，并且打开“Project Options（项目选项）”对话框，然后，在“General（一般设置）”选项卡中，启用“enable optimization（启用优化特性）”选项和自动内联。这个100表示全力以赴优化运行速度。然后，我单击“OK（完成）”，并按我在项目选项对话框中选定的新的优化标准，重新构建我的项目。由于我采用了另一种策略来构建代码，所以要再次解析所有这些输入代码，并根据我刚刚设置的优化级别，生成新的汇编代码。好了，我的库构建完毕。接下来，再将下面的这个应用代码设置为当前项目，然后，在“General（一般设置）”选项卡中，启用这些优化选项，再构建这个用户代码。VisualDSP++将利用我的应用，基于我刚刚构建的优化库，构建我的用户代码。现在，构建和加载已经完成；我把这个窗口放大一点以便大家可以看得更清楚。如果我运行这个代码，那么，这些“printfs”代码将执行解码过程，并且显示它正在Blackfin处理器的片上内存中解析这些输入文件。又是一些状态信息，现在其实也是在对代码进行基准测试，这和在评估板上进行基准测试是一样的。

记得吗？在第一次运行时，消耗了55亿个周期。而这一次，解码相同数量的字节只用了约一半的时间。这里显示的是27.5亿个周期，而不是55亿个周期。我们只是更改了优化选项，就实现了如此显著的提升。

现在，执行我刚才介绍的优化策略第二步——在头文件中启用指令高速缓存。在头文件中，关于指令高速缓存的设置只是一段预处理注释，因此，我要做的就是取消

关于这个选项的注释，在代码中添加两三个寄存器，启用指令高速缓存。现在，利用业经优化的库，重新构建我的用户代码。这将在刚才实现的27.5亿个周期的基础上进一步提高代码运行效率。好了，构建和加载都已完成，开始运行了。现在，这个处理器可以利用高速缓存，它会想“好吧，这些代码都已保存在外接存储器中，我要把反复使用的那部分放到片上内存中，并从片上内存上执行这些代码，即使它实际上是解析到外接存储器中的。”这就是高速缓存的作用。

接下来，进行基准测试。记住，上一次运行使用了约27.5亿个周期。现在，还是解码相同数量的字节，却只用了8.17亿个周期。相比于上一次未启用高速缓存时的基准测试，速度提高了67%，而那一次又比第一次直接采用未经优化的从网络下载的代码执行基准测试时，速度提高了50%。这张幻灯片显示了这一系列性能提升。左侧的这些百分比数代表处理器运行应用所耗用的周期数量。初始值是100%。在启用优化器后，周期数减少了50%，而我们所做的只不过是启用了一两个优化选项。接下来，启用指令高速缓存，也就是说编写了两三个寄存器，以充分利用高速缓冲存储器；结果，周期数量将至1/3，而第二次又是第一次的1/2。因此，通过这两个简单的步骤，就可以将这个需要消耗大量处理周期的代码的运行速度提高84%，相当惊人，而我们只需进行简单的操作。

第6章：结束语

第6a节：总结

好了，今天的课程单元到此结束了。我们演示了整个软件开发流程，希望大家能够理解连接器的工作方式，以便在以后进行系统优化时，借助连接器，随需利用存储器。我们还详细讲解了连接器，以帮助大家优化连接器。此外，我们演示了一个可以轻而易举地大幅提升代码性能的简单的两步式优化策略。最后，我们举了一个实例。如需了解更多信息，请登录下一张幻灯片上列出的链接。感谢收看本课程，祝你好运！