Analog Devices公司DSP、处理器和开发工具的使用技术指南
联系我们的技术支持中心：dsp.support@analog.com 和 dsptools.support@analog.com
查看我们的在线资源：http://www.analog.com/ee-notes 和 http://www.analog.com/processors

Blackfin®处理器LDF文件指南

Maikel Kokaly-Bannourah 提供 *2004年5月18*

介绍

本应用指南介绍VisualDSP++® 3.5支持的连接器描述文件(.LDF)。适合熟悉.LDF文件，并对通过修改现有文件或用程序片断写一个文件来用户化Blackfin®处理器.LDF文件感兴趣的用户使用。

本应用指南首先介绍VisualDSP++ 3.5内各种.LDF文件的作用，然后讨论Blackfin处理器.LDF文件各部分的概念，并详细介绍ADSP-BF533 Blackfin处理器的缺省的.LDF文件。

本应用指南不介绍用于ADSP-BF561 Blackfin处理器或VDK应用中的.LDF文件。

缺省的.LDF文件

对下面的每款处理器，VisualDSP++提供4个.LDF文件：

- ADSP-BF531 Blackfin处理器
- ADSP-BF532 Blackfin处理器
- ADSP-BF533 Blackfin处理器
- ADSP-BF535 Blackfin处理器
- AD6532处理器

每个.LDF文件的文件名表明此.ldf文件使用的处理器类型（例如ADSP-BF531.ldf），可能还包括可选择的后缀（如ADSP-BF533_C.ldf）。

如果.LDF文件名没有后缀，那它就是“缺省的.LDF文件”。即，如果没有明确指定某个.LDF文件，编译器连接应用程序时会使用缺省的.LDF文件。例如，ADSP-BF531.ldf是ADSP-BF531 Blackfin处理器的缺省的.LDF文件。如果没有通过–T命令行开关明确指定.LDF文件，编译器驱动器为目标处理器选择缺省的.LDF文件。下面的第一个命令使用缺省的.LDF文件，第二个使用用户指定的文件：

```
ccblkfn –proc ADSP-BF531 hello.c                # uses default ADSP-BF531.ldf
ccblkfn –proc ADSP-BF531 hello.c –T ./my.ldf    # uses ./my.ldf
```

对每个处理器，有三个.LDF文件，后缀是_C、_CPP和_ASM (例如ADSP-BF533_C.ldf)。这些.LDF文件是Expert Linker的模板。

如果您使用Expert Linker为您的设计创建一个定制的.LDF文件，Expert Linker将通过询问您希望创建的.LDF文件类型（Asm、C或C++），然后拷贝上述的一个模板来创建定制的.LDF文件。后缀表明他们支持的.LDF文件类型。

CPP模板是C模板的超集，C模板是ASM模板的超集。不同之处如下：

- CPP模板与C++指令库、C++异常库和用来初始化C++构造器的指令头文件链接。它映射数据部分，该数据部分包括控制如何发现异常的信息。
- 目前，C模板与CPP模板一样，这是因为C设计可以链接C++中已经实现的局部或系统库，但是将来的版本可能会有所不同。
- ASM模板不包括指令头部分，不允许命令行与应用发生冲突。ASM模板不适用于编译器的配置文件导引优化。由于模板没有run-time头部分，它不会发出指向复位地址的“开始”符号。它不会将C++异常部分映射到存储器中。

Blackfin处理器.LDF的概念

Blackfin处理器的.LDF文件可以分为5个主要部分：

- 前导段
- 库选择
- run-time头选择
- 存储空间声明
- 代码/数据到存储器映射定义

每个.LDF文件处理一系列命令，允许通过提供少量命令行选项来把多种配置植入应用。通过在.LDF文件中大量使用预处理器宏，可以实现这种灵活性。宏做为一种标记，指示一种选择或另一种选择，并做为.LDF文件中的变量来保持选定文件的名称或其它链接时间参数。这种对预处理器操作的依赖使.LDF文件显得非常重要。

命令行选项

简单地说，通过在链接器命令行上定义预处理器宏，可以选择不同的.LDF文件配置。可以从IDDE的工程选项的Link配置页或直接从命令行指定，例如：

```
ccblkfn –proc ADSP-BF533 prog.c –flags-link –MDUSE_CACHE
```

在链接过程中，上面的命令行定义USE_CACHE宏，从而选择.LDF文件配置，配置L1指令和数据存储空间为CACHE。

相反，没有宏定义：

```
ccblkfn –proc ADSP-BF533 prog.c
```

指定缺省配置，假定指令和数据存储区没有被配置为CACHE。

可以用其它几种方法定义宏。例如，在链接器配置页的处理器选项下（工程选项对话框），选项包括：不同库的使用、浮点库的选择、使用什么L1存储器。最后，这些选择在链接过程中定义一个宏，从而选择合适的.LDF文件选项。

L1存储器用法选项提供下面的两种选择：

- Instruction and Data RAM
- Cache (sets USE_CACHE macro)

顾名思义，第二种选择设置了USE_CACHE宏，第一种选择不变，因此可以达到上例中的相同效果。

其它选择则较间接，但操作方式相同。Floating-point选项也有两种选择：

- High performance
- Strict IEEE conformance

第二种选择调用具有-ieee-fp开关的编译器驱动，第一种选择调用没有浮点开关的编译器，它选择-fast-fp的缺省特性。通过在链接时定义IEEEFP宏、或不定义它们，使这些编译器开关生效。

其它编译器开关也通过将宏传递到链接器以选择.LDF文件选项来实现。

构建变量和文件命名

预构建的Blackfin处理器库和run-time头文件，可以有多种配置，并可以在它们之间选择不同的LDF选项。使用对库名添加后缀的命名方法，以便在它们之间选择.LDF文件。

首先，每个文件以特定的处理器或特定的核进行构建。大多数文件是为了特定的核心（即ADSP-BF535 Blackfin处理器或与ADSP-BF532处理器兼容核心）而构建的。这些文件的后缀是535或532。一些文件（如cplbtab531.doj）以特定的处理器构建，（顾名思义）是为ADSP-BF531 Blackfin处理器构建的。

总的来说，由于每个.LDF文件已经代表一个特定的处理器，且代表使用合适处理器/核心后缀的库，处理器/核心的命名不是一个LDF选项。

其它后缀是选项，库和run-time头文件之间的命名有些许不同。库后缀如表1所示。

d	库包含用于调试的符号信息。
x	库具有激活的C++异常（即“-eh”）。
y	库具有所有激活工作区（即“-workaround all”）
mt	库具有目前被激活的任何多线程支持（即“-threads”）。

表1 C库文件后缀

唯一使用具有多线程支持的库(-threads)的.LDF文件是VDK .LDF文件，本应用指南不予介绍。

run-time头文件后缀如表2所示。

c	头文件初始化C++构造器
f	包括文件I/O支持
mt	任何多线程支持都被激活
n	不是run-time头文件，是C++构造器表的表结束标记
p	包括指定代码介绍
s	调用main()时，使处理器处于Supervisor模式
y	编译连接时workaround使能

表2 run-time头文件后缀

LDF文件一同使用这些后缀和命令行宏来决定链接哪个文件。例如：

```
#ifdef __WORKAROUNDS_ENABLED                /* { */
#define LIBSMALL libsmall532y.dlb,
#else
#define LIBSMALL libsmall532.dlb,
#endif                                      /* } */
```

__WORKAROUNDS_ENABLED宏通过编译器的-workaround开关来设置，LIBSMALL宏根据是否使用这个开关来设置一个值。稍后，在.LDF文件中，在库列表中使用LIBSMALL来进行链接。

C++构造器表

在调用main()之前，必须构造位于全局范围内的C++对象，因此，它们的构造器必须从run-time头部分中调用。

对于每一个编译的C++模块，编译器创建一个特殊的数据部分，称为ctor。这个数据部分包括指向全局对象用的构造器函数的指针。

用于对象列表的.LDF文件是：

```
run-time header, command-line objects, ctor-terminator
```

run-time头文件把一个标签放在ctor中。由于链接器服从对象的排序，标签在ctor部分的开始位置。任何指向构造器的ctor指针来自下面的命令行对象，后面有ctor结束符号（一个NULL指针）。这意味着ctor部分被组合到一个指针到构造器的表中，run-time头文件预排并处理它。

配置文件导引优化支持

配置文件导引优化(PGO)是收集在具有多种输入数据的大量可执行请求上运行应用的信息，然后使用收集到的信息再优化。该过程依赖于在不同的数据设置下运行的同一个应用，这通常意味着应用使用存储在文件中的实例数据来运行。更明确地说，这意味着应用是通过传递到main()的命令行选项处理每个文件的。

.LDF文件和IDDE一起对命令行变量提供支持。通常情况下，典型的嵌入式程序对命令行变量不感兴趣，且什么也不接收。这些情况下，运行部分调用一个函数来对全局字符串__argv_string[]进行语法分析，并查明它是空的。

为了支持PGO，可以用LDF选项IDDE_ARGS来定义一个叫做MEM_ARGV的存储部分，__argv_string[]被直接映射到这部分的开头部分。IDDE也遵循该惯例，即命令行变量可以通过在MEM_ARGV开始处将变量字符串写入存储器，来将命令行变量传递到一个应用。

使用存储器

LDF文件为处理器1上所有定义的空间指定存储区域，在.LDF文件中，并不是所有这些存储区域都使用了，相反地，.LDF文件提供两种基本的存储器配置：

- 缺省配置规定只有内部存储器可用、不激活CACHE。因此，没有指令或数据被映射到SDRAM，除非明确说明放在那里，所有可用的L1空间被代码或数据使用。
- USE_CACHE选项选择替代配置，可使指令和数据CACHE被激活，并使用外部SDRAM。代码和数据被映射到L1上可用的空间，但CACHE/SRAM区域是空的，任何溢出指令段都放置在SDRAM的低32 MB区域。

如果使用USE_CACHE，可以安全地打开CACHE，因为这样做不会损坏代码和数据。选择这个选项实际上不会真正使能CACHE－这必须单独进行（例如，通过___cplb_ctrl配置变量）。相反地，这个选项确保存储器规划允许稍后使能CACHE。请注意，当USE_CACHE激活时，SDRAM不是自动激活。

常见的用户错误是，尽管没有规定USE_CACHE，还是激活CACHE，这样，由于CACHE覆盖了SRAM的内容，导致代码或数据破坏。因此，.LDF文件使用下面的“防护符号”。

- ___l1_code_cache
- ___l1_data_cache_a
- ___l1_data_cache_b

这些符号被.LDF文件定义，并根据是否定义了USE_CACHE来赋值（即规定其地址为0或1）。当需要CACHE配置时，run-time库检查这些符号，如果相应的防护符号是0，表明有效信息已经占据了这个空间，就拒绝诗能CACHE。

输入部分

这部分介绍编译器和库生成的，由.LDF文件映射到存储器区域的各种代码和数据部分的内容和目的。

LDF文件使用如下代码部分：

[1] 除核心MMR以外，链接器认为它“超越限度”。

- L1_code: 缺省时，编译器不生成代码。.LDF文件映射这部分，以便通过使用这部分的指令使必须进入L1指令SRAM的代码被明确放在那里。这部分通常是映射到L1 指令SRAM的第一部分。
- cplb_code: 在这里是run-time库的CPLB管理程序。通常映射到L1指令SRAM中。特别地，如果可能代替CPLB，这部分必须映射到保证一直可用的存储器中，这意味着必须使用锁定的CPLB对其进行编址。
- program: 是编译器生成的缺省代码部分。如果可能，这部分的代码将映射到L1指令存储器，不过，如果没有足够的空间，代码将溢出到较慢的存储器中。

LDF文件使用以下数据部分：

- L1_data_a: 缺省情况下，编译器在此不生成数据。这部分与L1_code类似，因为它使用这部分指令，允许将数据明确地映射到L1 Data A SRAM中。
- L1_data_b: 与L1_data_a类似，不过用它来将数据映射到L1 Data B SRAM中。
- cplb_data: run-time库的CPLB管理程序使用CPLB配置表（放在这里）。特别地，被.LDF 文件映射的cplbtabx.doj文件(*x*指目标)被放在这部分里。它必须映射到一直可用的数据区域中。如果可能进行CPLB替代，这部分必须被映射到被锁定的CPLB所覆盖的存储空间中。
- data1: 用于编译器生成的全局数据的缺省部分。
- voldata: 编译器不为这部分生成数据。它用于放置由于外部影响（如DMA）可能变化且不能被放在CACHE的数据。
- constdata: 用于声明为常数的全局数据，和文字常量如字符串和阵列初始化。编译器的缺省特性是假设const数据是常量，因此，可以映射到只读存储器中（尽管缺省的.LDF文件不做此工作）。在这方面，编译器的特性与ANSI C Standard不同：在ANSI C下，诸如const数据，可以通过其它手段改变。详情请参照“*VisualDSP++ 3.5 C/C++ Compiler and Library Manual for Blackfin Processors* [1]. ”中的-const-read-write开关。
- bsz: 用于未初始化全局数据，当指定-bss开关时，其被隐含地初始化为零。这部分实际上不包括数据，通过IDDE载入或被载入程序处理时，被填充为零。当使用-no-bss开关时，不生成这部分。
- bsz_init: 保存一个指向由存储器初始化实用程序生成的run-time初始化数据的指针。预计这部分将被映射到只读存储器中。当.DXE文件被存储器初始化实用程序处理、程序开始运行时，通过将这部分的数据拷贝到其它的数据部分（如data1和constdata），对其它数据部分进行初始化。如果不使用存储器初始化实用程序，这部分可以被映射到RAM中。

LDF文件使用下面的类部分：

- cplb: 这部分是历史文件，不再使用。已经被cplb_code 和cplb_data取代。
- sdram0: 这部分通过使用部分指令，允许代码或数据明确映射到外部存储器中。可以用来将大量

的、不常用的数据或功能放入外部存储器中，以释放宝贵的内部存储器。

LDF文件使用下“表”部分。在所有的情况下，这些部分都可以被映射到只读存储器中。对给定的部分*s*及其结束部分*sl*，必须在*s*部分后立即映射结束部分*sl*。

- ctor: 这部分包括指向C++对象的构造器的指针。在调用main()之前，其用来调用构造器。在存储器内其必须是紧邻的。
- ctorl:这部分包括ctor表部分的结束符号。在ctor部分之后，必须被立即映射。
- gdt:全局遣送表。C++异常库用它来决定哪个特定的地址属于哪些代码段。在存储器内其必须是紧邻的。
- gdtl:这部分包括.gdt表部分的结束符号。在.gdt部分之后，必须被立即映射。
- edt: 异常发送表。C++ 异常用它来从try块映射到catch块。
- cht: 捕捉处理程序类型表。用来映射RTTI类型的信息。C++ Exception Library用它来决定于try块的捕捉输入等同的类型。
- frt:功能范围表。在异常处理过程中，C++ Exception Library用它来释放起作用的堆栈。
- frtl:这部分包括.frt表部分的结束符号。在.frt部分后，必须被立即映射。

详细介绍ADSP-BF533.LDF

前导段

.LDF文件的开头是简单的注释，介绍.LDF文件中可以使用的选项（由链接器预处理器宏决定）。在.LDF文件中的相关部分，对这些选项有详细的解释。

ARCHITECTURE指令规定这个.LDF文件用于ADSP-BF533 Blackfin处理器。所有的Blackfin处理器.LDF文件都面向单个处理器。

SEARCH_DIR指令识别标准run-time库的位置，像VisualDSP++安装目录的Blackfin\lib子目录一样。链接器将$ADI_DSP置为VisualDSP++安装目录。

如果选择__NO_STD_LIB选项，不包括SEARCH_DIR指令，这意味着链接器没有搜索run-time库的缺省空间。这个选项由-no-std-lib编译器开关选择，确保应用仅被用户提供的库链接。

库选择

这部分.LDF文件构建不同的宏和变量，目的是产生$LIBRARIES表，按需要的顺序搜索库和目标文件，解决引用问题。一些选项指定选择一个库对另一个库（如工作区激活版本对普通版本），其它选项指定选择一个库对另一个库顺序（如选择完全兼容的IEEE浮点支持版本对较高性能版）。

USE_FILEIO选项是强制定义的。这对在大部分开发周期内使用的printf()和其它与stdio-相关的函数，是必要的。禁用USE_FILEIO选项就禁用了所有与stdio-相关的I/O操作。

一些选项与指定代码介绍有关。编译器有三个开关用于profiler: -p, -p1和-p2。对这三种开关，编译器用相同的方法编码，使它在每个指定函数调用的开始和结束处调用profiling库，以获得统计数据。这三种开关的不同之处是随后的应用如何报告收集到的数据。-p1开关将数据写到mon.out文件，-p2开关将它写到标准的输出流，-p开关将数据写到这两个位置。通过链接由.LDF文件选项控制的不同的目标文件，就可以实现这些不同。

开关选择下面的LDF选项：

- p selects USE_PROFILER
- p1 selects USE_PROFILER1
- p2 selects USE_PROFILER2

当检测到输入目标文件被指定给profiling时，由预链接器设置一个附加的USE_PROFILER0选项。这意味着，即使链接时没有打开profiling开关，也必须链接profiling库。USE_PROFILER用来设置USE_PROFILER0，解释链接时任何profiling开关的缺少，缺少时表明两种输出方法都要使用。

通过链接包括两个全局变量的小目标文件，决定输出方法的选择。这些变量表明是否选择了每个输出方法。几个不同的目标文件定义两个变量的置换（见表2）。像其它的目标文件一样，这些文件存在几个变量，如“普通”和“激活工作区”变量。当对象构建了激活工作区时，会产生不同的代码以处理已知的芯片异常。当对象只包括数据时（没有代码），在这个特例中，这个“激活工作区”开关不起作用，“变量”是一样的。仍旧构建这些冗余的多个变量，是为了保持命名习惯。

如果需要profiling的话，PROFFLAG被定义为一个目标文件的名称。如果不需要，就不定义PROFFLAG。

OMEGA是包括idle程序的文件名称。如果应用自动终止，如从main()返回或通过调用exit()，就使用这个程序。

只要链接时需要工作区，__WORKAROUNDS_ENABLED选项就被编译器驱动器定义；尽管编译时可以单独选择每个可用的工作区，不能对所有的工作区组合提供预构建库。因此，每个库和目标都有单个激活工作区版本，选择了所有的工作区。

MEMINIT保持目标文件的名称，包括指向由存储器初始化实用程序产生的任何初始化数据的指针。除非随后的应用被存储器初始化程序处理，否则，这个指针是NULL指针。这个文件没有变量。

规定LIBSMALL是监视模式库的名称。

M3_RESERVED选项用于模拟器。缺省时，模拟器将Blackfin处理器的堆栈用作工作区，不过，可以使用M3寄存器来替代它。构建的run-time库具有-reserve m3开关，允许使用寄存器替代堆栈用作工作区。但有少量程序需要额外的处理。如，如果保留了M3，中断处理程序可能不存储和恢复M3，但如果没保留，就必须进行这样存储和恢复。因此，用一些库的两个变量来处理这种情况。LIBM3包括

异常处理程序setjmp()和longjmp()，而LIBDSP包括与DSP有关的优化程序，当M3被模拟器保留时，会损失一些性能。

通过排序选择浮点库。完全兼容的库是独立的库，高性能库是LIBDSP的一部分。这两个库的排序根据需要的浮点库而改变。由-ieee-fp编译器开关设置IEEEFP选项，选择替代排序方案。

这样选择的库被汇编到LIBS表，如果设置了__ADI_LIBEH__(链接时由-eh设置)，可以选择与C++异常（-eh和-rtti开关）一起构建的C++库。合适的文件I/O库附到这个表上，形成完整的$LIBRARIES表。

CRT的选择

这部分选择需要的C run-time部分(CRT)的预构建版本，调用在main()之前执行的启动代码。根据下面的四个选项的值（定义的或未定义的），选择CRT：

- USE_PROFILER
- USE_FILEIO
- __cplusplus
- __WORKAROUNDS_ENABLED

选中的目标文件命名时，使用本应用指南前面介绍的后缀。对后三个选项，选择单个目标文件，包括在$OBJECTS表中。如果选择USE_PROFILER，也设置CRT，以包括profiling库和前面决定的PROFFLAG文件。

C++构建(__cplusplus选项)时，把ENDCRT定义成包括C++构造器表的表结束符号的对象。对其它构建，不定义ENDCRT。

然后，CRT和ENDCRT、以及$COMMAND_LINE_OBJECTS，被包括在$OBJECTS表中，链接器定义它是链接器命令行上命名的所有对象和库。CPLB定义表对象，用来配置CACHE设置，完成$OBJECTS表。

$LIBRARIES和$OBJECTS为链接器使用的源命名，以分解符号。$OBJECTS中的项被链接到应用中，$LIBRARIES从$OBJECTS得到分解符号参考。

存储空间声明

Blackfin处理器.LDF文件定义与硬件存储器映射很接近的存储器。.LDF文件有附加的划分方法，可以根据run-time库如堆和堆栈的不同需要，对存储器进行合适的分配。

核与系统的存储器映射的寄存器(MMR)在存储器的顶部。链接器不允许为核MMR定义存储器部分，这个存储器声明被通过注释去掉，不过还保留下来，以保持透明度。

定义了L1中间结果暂存器，不过，.LDF文件不使用它。注意，静态初始化数据可能不会映射到中间结果暂存器。

L1指令存储器被分为代码和代码CACHE。代码CACHE可以被选作CACHE或代码空间。

L1数据存储器Bank B被分为数据和数据CACHE，前者进一步分为常数数据和堆栈使用的空间。总的来说，.LDF文件试图将堆栈放置到快速L1存储器中，因为编译器常常将堆栈用作函数参数和临时工作空间。

L1数据存储器Bank A也有CACHE/非CACHE的划分。非CACHE区域有可选择的256字节用于命令行变量。这个IDDE_ARGS选项支持导配置文件导引优化（PGO），如前面所述。

定义了4个异步存储器Bank和一个SDRAM Bank。后者是存储器的最低Bank，是一个32 MB的空间，最低16 KB被声明为堆使用。注意堆不包括地址0x0000 0000，这个地址为C中的NULL指针保留，不能用来指向有效的数据。

代码/数据到存储器映射定义

PROCESSOR指令将随后得到的.DXE文件写入$COMMAND_LINE_OUTPUT_FILE，这是链接器定义的输出文件名称，由编译器驱动器传递。

使用RESOLVE指令，start符号（CRT的第一个可执行部分）被直接映射到处理器的复位地址。如果使用配置文件导引优化（PGO），通过同样的方法，_argv_string符号被直接映射到变量部分的开头。

_main和start符号被明确命名，以免激活选项时，被链接器屏蔽。

.LDF文件的其余部分将代码或数据通过存储区域从输入部分映射到输出部分。.LDF文件是通用的，目的是确保适用所有的情况。因此，像program和data1这样的部分，被映射到不同存储区域的大量输出部分中。当一个存储部分添满后，余下的可能会进入其它的部分。

一些项可以随意映射。C++对象仅能用于C++链接映射；代码和数据被映射到L1 SRAM（当指定USE_CACHE选项时，被映射到外部存储器)的CACHE区域中。

BSZ部分被标记为ZERO_INIT，它指示存储器初始化时，将相应部分初始化为空并且在运行时应该填充零。bsz_init部分包括指向存储器初始化程序产生的表的开始处的指针，通常被映射到只读存储器。为简单起见，在开发的初期，它被映射到常规数据部分。

.meminit部分被链接器区别对待。在链接时，.meminit中没有代码和数据。不过，一旦链接完成，.meminit部分会扩展，使存储区域中所有未使用的空间用来容纳被映射的内容。特殊的.LDF文件中，.meminit被映射到MEM_L1_DATA_B，因此，链接后，.meminit占据.DXE文件未使用的MEM_L1_DATA_B的其余部分。这定义了一个空白空间，存储器初始化程序可以用它来存储配置表。像bsz_init一样，.meminit部分通常映射到只读存储器空间。

堆栈和堆空间没有.LDF文件映射的数据。而是各自定义全局符号，包括它们的开始、结束地址和长度（对堆而言）。这允许Run-time Bank来决定运行时堆栈和堆的大小和位置，因此，在.LDF文件中，可以很容易地修改它们。注意CRT要求这些符号。

参考文献

[1] VisualDSP++ 3.5 C/C++ Compiler and Library Manual for Blackfin Processors. Rev 2.2. October 2003. Analog Devices, Inc.

文件历史

版本	描述
第一版－2004年5月18日，Steve K提供	第一版