C6000 嵌入式应用程序二进制接口

Application Note

Literature Number: ZHCADC4B
SEPTEMBER 2011 – REVISED AUGUST 2025

目录

插图清单

表格清单

章节 1
简介

本文档是德州仪器 (TI) 的 C6000 系列处理器的基于 ELF 的嵌入式应用二进制接口 (EABI) 规范。EABI 是宽泛的标准，定义了程序、程序组件和执行环境 (如果存在操作系统，还包括操作系统) 之间的低级别接口。EABI 的组件包括调用约定、数据布局和寻址约定、目标文件格式，以及动态链接机制。

本规范旨在使 C6000 的工具提供商、软件提供商和用户能够构建彼此可互操作的工具和程序。

1.1 ABI - C6000

在 2009 年 TI 的 C6000 编译器工具 7.0 版本发布之前，C6000 的唯一 ABI 是基于 COFF 的原始 ABI。它严格上来说是一个裸机 ABI；没有执行级别的组件，尽管各种系统实现了动态链接的各个方面，但这类机制没有标准化或工具支持。

TI 编译器工具的 7.0 版本引入了一种名为 C6000 EABI 的新 ABI。它基于 ELF 目标文件格式，并支持动态链接和位置独立性。它源自业界标准模型，包括 *IA-64 C++ ABI* 和*用于 ELF 和动态链接的 System V ABI*。ABI 的处理器特定方面（例如数据布局和调用约定）与 COFF ABI 相比基本没有变化，尽管存在一些差异。毋庸置疑，COFF ABI 和 EABI 是不兼容的；也就是说，给定系统中的所有代码都必须遵循相同的 ABI。TI 的编译器工具支持新的 EABI 和旧的 COFF ABI，但我们鼓励迁移到新的 ABI，因为未来可能会停止支持 COFF ABI。

平台 是程序运行所在的软件环境。ABI 具有特定于平台的方面，尤其是在与执行环境相关的约定领域，例如程序段的数量和使用、寻址约定、可见性约定、抢占、动态链接、程序加载和初始化。目前有两个受支持的平台：裸机和 Linux 。*裸机* 一词表示不存在任何特定环境。这并不是说不能有操作系统，而是说没有特定于操作系统的 ABI 规范。换句话说，裸机 ABI 未涵盖程序的加载和运行方式以及它如何与系统的其他部分进行交互。

裸机 ABI 允许在许多具体方面存在很大的可变性。例如，实现可能提供位置独立性 (PIC)，但如果给定系统不要求位置独立性，则这些约定不适用。由于这种可变性，程序可能仍然符合 ABI，但不兼容；例如，如果一个程序使用 PIC，但另一个程序不使用，则它们无法互操作。工具链应努力强制执行此类不兼容性。

Linux ABI 通过缩小裸机 ABI 的可变性并详细说明其他要求，对裸机 ABI 进行了补充，从而使程序或子程序可以在 C6000 上基于 Linux 的操作系统中运行。

1.2 范围

图 1-1 显示了 ABI 的组成部分及其关系。我们将从下到上简要描述图中的这些组成部分，并提供在此 ABI 规范中参考的相应章节。

底部区域的组成部分与目标级互操作性有关。

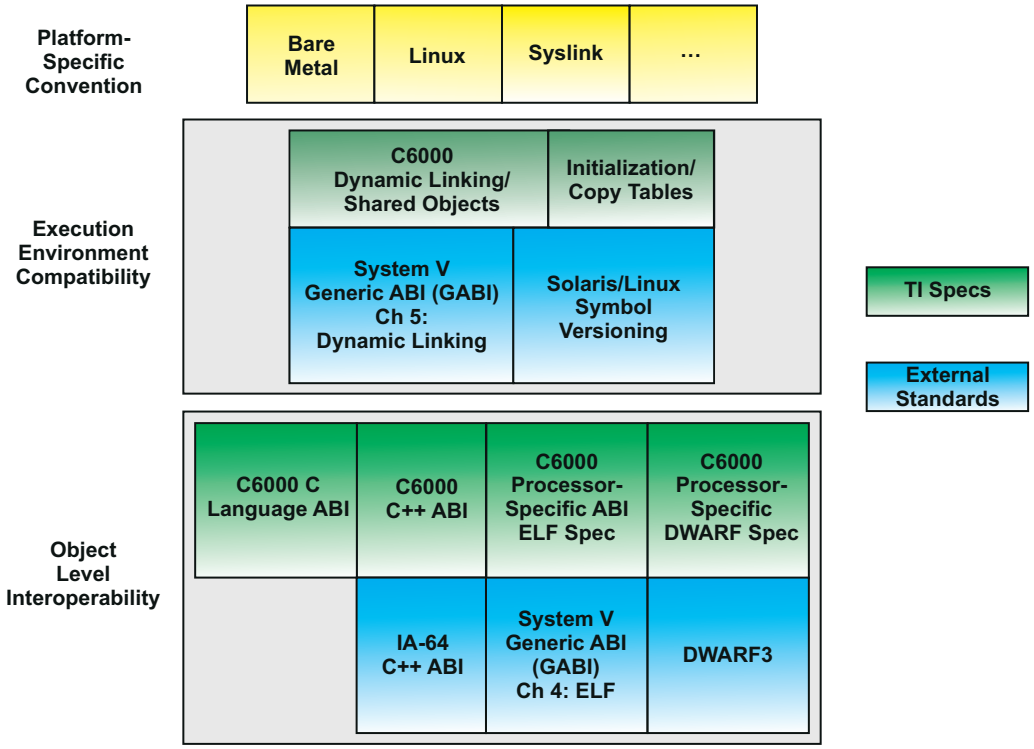


图 1-1. ABI 规范的组成部分

C 语言 ABI（章节 2、章节 3、章节 4、章节 5、章节 8 和章节 9）规定了函数调用约定、数据类型表示、寻址约定和 C 运行时库的接口。

C++ ABI（章节 10）规定了如何实现 C++ 语言；这包括有关虚拟函数表、名称改编、如何调用构造函数以及异常处理机制（章节 11）的详细信息。C6000 C++ ABI 基于流行的 IA-64 (Itanium) C++ ABI。

DWARF 组成部分（章节 12）规定了目标级调试信息的表示。基本标准是 DWARF3 标准。此规范详细说明了处理器特定的扩展。

ELF 组成部分（章节 13）规定了目标文件的表示。该规范为系统 V ABI 规范扩充了处理器特定的信息。

构建属性（章节 17）是指一种将影响对象间兼容性的各种形参（如目标设备假设、内存模型或 ABI 变体）编码到目标文件中的方法。工具链可以使用构建属性来防止组合或加载不兼容的目标文件。

图中间区域的组成部分与执行时互操作性有关。 **动态链接**组成部分（章节 6 和节 14.3）规定了各单独链接的模块可以用来互操作的机制，包括共享它们的代码。数据寻址方法属于动态链接机制的一部分，这样，单独链接的模块便可以在不进行重定位的情况下，对彼此的数据进行寻址。

线程局部存储（章节 7）支持创建静态存储持续时间的线程特定变量。本文档描述了线程局部变量的规范、表示和访问。

符号版本控制（章节 16）是一种机制，通过该机制，符号引用可以包含最低版本，这样就可以使用至少具有该版本的定义来动态解析它们，以便防止运行时不兼容问题。本 ABI 采用标准 GCC/Linux 模型，不做任何更改。

图 1-1 顶部的组成部分为 ABI 扩充了平台特定的约定，后者可以定义使可执行文件与执行环境兼容的要求，如程序段的数量和使用、寻址约定、可见性约定、抢占、程序加载和初始化。**裸机**是指缺失任何具体环境。ABI 目前涵盖的唯一其他环境是 Linux 平台（章节 15）。

最后，有一组规范不是 ABI 的正式组成部分，但本文档进行了介绍以供参考，同时供其他工具链选择实现。

初始化（章节 18）是指初始化变量赖以获取其初始值的机制。名义上，这些变量驻留在 .data 段中，在加载 .data 段时会直接将它们初始化，不需要工具额外参与。然而，TI 工具链支持一种机制，通过该机制，.data 段能够以压缩形式编码到目标文件中，并在启动时解压缩。这是一种通用机制的特殊用法，该机制以编程方式将压缩后的代码或数据从离线存储（例如 ROM）复制到其执行地址。我们将该过程称为*复制表*。虽然不是 ABI 的一部分，但本文档介绍了初始化和复制表机制，以便在需要时通过其他工具链使用。

程序标头属性（章节 19）是由 TI 工具链实现的 ELF 扩展，用于表示 ELF 段在基本 ELF 标准指定属性之外的各种其他属性。TI 工具使用这类属性对内存连接/延迟要求、保护、高速缓存行为和其他系统特定属性进行编码。这类属性具有灵活性和可扩展性。同样，本文档对其进行了介绍，以便在需要时使用其他工具更改这些属性。

1.3 ABI 变体

如前所述，ABI 并未定义所有情况下的具体行为，而是一套允许平台或系统特定变化的原则规范。例如，ABI 并未指定在所有情况下都使用 PIC（与位置无关的代码）寻址，而是针对使用 PIC 的情况标准化其实现。有些变体彼此不兼容。例如，如果任何对象使用 DBST PIC 模型，则所有对象都必须使用。在这种情况下，工具链应使用构建属性来防止组合不兼容的对象。

本节介绍了一些更常见的用例以及它们与 ABI 的关系。这些情况并不相互排斥，也没有完全涵盖所有可能性。

- **裸机 — 独立**：此模型是指一个静态连接的自包含可执行文件。就互操作性而言，它是最简单的形式。ABI 的相关部分是 图 1-1 下部的对象级别组件。由于可执行文件是静态链接和绑定（重定位）的，因此通常不需要位置独立性。由于它是自包含的，因此不需要包含动态链接信息、过程链接表 (PLT) 存根或全局偏移量表 (GOT)。
- **裸机 — 动态链接**：此模型是指这样一个系统，在该系统中，可执行文件可以动态链接到单独链接的模块，但不在操作系统的受控环境中。寻址可能与位置无关，也可能与位置相关，具体取决于环境。环境可能会对寻址或放置方面施加额外的约定。该模型将使用图 1-1 的动态链接组件。有关裸机动态链接模型具体情况的详细信息，请参阅节 14.4。
- **共享对象**：这是指动态链接模型，其中静态链接模块（库）可以在多个单独链接的客户端（可执行文件或其他库）之间共享。根本的问题是每个客户端都必须有自己的库数据副本。ABI 通过使用两个相关结构解决了此问题：位置无关寻址和数据段基表 (DSBT) 机制。
- **位置独立性**：这是指不使用地址常量的寻址方式，使得能够在任何地址加载并运行代码和/或数据而无需重定位。*PIC* 一词通常是指与位置无关的代码，但位置独立性可以指代码、数据或两者。共享库需要与位置无关的数据，以便多个客户端可以拥有私有副本；在共享库的语境中，PIC 一词有时表示这一较狭义的定义。如果在创建 ROM 时未绑定其他对象的地址，则 ROM 中的库可能需要与位置无关的寻址来引用其他对象。与位置无关的数据依赖数据页寄存器 (B14)。当涉及多个模块时（例如，使用动态链接），*DSBT*（数据段基表）模型是一种机制，可用于在从一个模块调用到另一个模块时复位 DP。
- **Linux**：为 Linux 环境构建的可执行文件和共享库必须遵循某些约定。它们具有动态链接信息。它们需要使用 DSBT 模型来实现位置独立性。为 Linux 构建的对象在 ELF 标头的 EI_OSABI 字段中具有 ELFOSABI_C6000_LINUX 标志。有关针对 Linux 平台的 ABI 增强的详细信息，请参阅章节 15。
- **ROM 操作**：可能需要构建一个驻留在 ROM 中的单独链接的模块。链接后，其地址会被永久绑定。它随后可能会静态或动态链接到其他模块。为此，ABI 定义了一类特殊的 ELF 文件，该类文件提供静态和动态链接视图以及几个段标志，以指示其地址永久绑定的段。ROM 模块通常使用 PIC 寻址，使它们独立于其所引用的其他模块的位置。

1.4 工具链和互操作性

此 ABI 不特定于任何特定供应商的工具链。实际上，它的目的是使替代工具链能够存在并可互操作。ABI 描述了如何实现机制；而不是工具链如何在用户层面支持它们。有时会提到 TI 工具，它们仅供说明之用。不过，TI 的

C6000 编译器工具本质上具有独特的地位，因为它们源自器件供应商，并根据 ABI 规范共同开发，在某些情况下构成了后者的基础。

如果 TI 工具的行为与本 ABI 相冲突，则应将其视为工具中的缺陷；如果您发现此类情况，请将缺陷报告提交至 support@tools.ti.com。然而，若本规范不完整或不明确，TI 工具的行为应视为具有决定性。ABI 标准的主要目标是与 TI 工具实现互操作；工具链供应商应努力实现该目标，而无论标准本身是否有遗漏或岐义。在这种情况下请通知我们，我们将努力澄清规范。

1.5 库

通常情况下，工具链包括链接器以及标准运行时库，这些库实现工具链提供的部分语言支持。

C6000 所用库格式是通用 GNU/SVR4 ar 格式。

链接器和库通常具有 ABI 范围之外的相互依赖性。例如，许多链接器使用特殊符号来控制各种库组件的包含或排除；或者，某些库会引用特殊的链接器定义符号。因此，链接器和库应来自同一工具链。如果所用链接器来自一个工具链，而库却来自另一个工具链，则该 ABI 不会支持。这仅适用于属于工具链一部分的内置库；可以链接使用不同工具链构建的应用库。

1.6 目标文件的类型

ELF 定义了以下不同类别的目标文件：

- **可重定位**文件包含一些代码和数据，它们适合与其他目标文件进行静态链接，以创建可执行文件或共享目标文件。
- **可执行**文件包含适合执行的程序。它可能具有或没有动态链接信息。
- **共享目标文件**是程序的组成部分，可在加载时与可执行文件和其他共享对象组合形成进程映像。共享对象始终包含动态链接信息。为避免与可重定位目标文件混淆，我们有时使用术语*共享库* 来指代共享对象。
- **可重定位模块**。可重定位模块是一个共享目标文件，其中也包含静态链接信息：即静态符号表、段表和静态重定位条目。它适用于可以静态或动态链接的可固化库。

此规范使用可互换的术语*静态链接单元* 和*加载模块* 来指代可执行文件和共享库（包括可重定位模块）。

1.7 段

ELF 加载模块（可执行文件或共享对象）以*段* 形式表示程序的存储器映像。在这种语境下，段是指具有共同属性的连续的、不可分割的存储器范围。当段的地址确定后，它便成为联编段，这可能在链接时静态发生，也可能在加载时动态发生。

1.8 C6000 架构概述

TMS320C6000 通常称为 C6000 或 C6x，是德州仪器 (TI) 的 32 位 VLIW 数字信号处理器系列。该系列包括定点（整数）器件和浮点器件。该架构能够在每个周期发出多达 8 个 32 位指令，以实现高水平的并行性。表 1-1 列出了此 ABI 所涵盖的 C6000 产品系列的成员。

表 1-1. C6000 ISA

ISA	数据格式	说明
C62x	定点	原始 ISA
C64x	定点	具有附加指令和寄存器的 C62x
C64x+	定点	附加指令和紧凑指令编码
C67x	浮点	原始浮点 ISA
C67x+	浮点	具有附加指令和寄存器的 C67x
C6740	定点/浮点	C64x+ 和 C67x+ 的联合体以及附加指令
C6600	定点/浮点	具有 128 位数据路径和附加指令的 C6740

大多数系列成员都向后兼容；也就是说，较新的 CPU 可以正确执行为较旧器件构建的目标代码。特定情况在 节 17.2 中的 Tag_ISA build 属性下指定。

C6000 器件可按字节寻址。存储器可以配置为大端字节序或小端字节序。大多数器件没有通用存储器管理单元，因此 CPU 地址指的是实际的物理存储器位置（无虚拟存储器）。

C6000 的流水线不受保护。也就是说，当 CPU 读取先前发布但仍在流水线中且尚未写入的计算的目标时，读取将获取旧值，而不是停止以等待新值。这意味着编程器（或编译器）必须管理流水线延迟并调度操作，才能获得正确的结果。具有多周期延迟的操作包括加载（4 个周期）、跳转（5 个周期）和某些乘法（2 个周期）。

C6000 至少具有 32 个通用寄存器，指定为 A0-A15 和 B0-B15。C64 系列产品的成员将其扩展到 64 个寄存器：A0-A31 和 B0-B31。其中两个寄存器按照约定分配，用于 ABI 下的寻址和链接。B15 被指定为栈指针，通常表示为 **SP**；B14 被指定为数据页指针，表示为 **DP**。DP 用作数据段的基地址，既实现了位置无关性，同时又提供了对 (near) 数据的高效访问。

1.9 参考文档

文档标题	链接或 URL
TMS320C64x/C64x+ DSP CPU 和指令集参考指南	SPRU732
TMS320C6000 优化编译器用户指南	SPRU187
TMS320C6000 汇编语言工具	SPRU186
ELF 规范 - GABI 第 4/5 章	http://www.caldera.com/developers/gabi/2003-12-17/contents.html
IA64 (Itanium) C++ ABI	http://refspecs.linux-foundation.org/cxxabi-1.83.html
IA64 (Itanium) 异常处理 ABI	http://www.codesourcery.com/public/cxx-abi/abi-eh.html
ARM 架构的应用二进制接口	http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.swdev.abi/index.html
适用于 ARM 架构的 C 库 ABI	http://infocenter.arm.com/help/topic/com.arm.doc.ihi0039b/IHI0039B_clibabi.pdf
DWARF 调试格式版本 3	http://dwarfstd.org/Dwarf3.pdf
C 语言标准	http://www.open-std.org/jtc1/sc22/wg14，ISO/IEC 9899:1990
C99 语言标准	http://www.open-std.org/jtc1/sc22/wg14，ISO/IEC 9899
C++ 语言标准	http://www.open-std.org/jtc1/sc22/wg21，ISO/IEC 14882:1998

1.10 代码片段表示法

在本文档中，我们使用代码片段来说明寻址、调用序列等。在片段中，通常使用以下符号约定：

sym	要引用的符号
标签	表示代码地址的符号
func	表示函数的符号
tmp	临时寄存器（还有 tmp1、tmp2 等）
reg, **reg1**, **reg2**	任意寄存器
dest	产生的值或地址的目标寄存器

引入了汇编器内置的几个运算符。它们用于为各种寻址结构生成适当的重定位，并且通常不言而喻。

为了简单起见，代码序列未调度。也就是说，假设每条指令执行完成之后才开始执行下一条指令。

章节 2

数据表示

本节介绍标准 C 数据类型在存储器和寄存器中的表示。可支持其他语言；这些语言使用的类型将定义其自身至这些表示的映射。

在本节的描述和图表中，位 0 始终指最低有效位。

2.1 基本类型

整数值使用二进制补码表示法。浮点值使用 IEEE 754.1 表示法表示。浮点运算在硬件支持的程度上遵循 IEEE 754.1。

表 2-1 提供 C 数据类型的大小和对齐方式（以位为单位）。

表 2-1. 标准类型的数据大小

类型	通用名称	大小	对齐
signed char	char	8	8
unsigned char	uchar	8	8
char	普通字符	8	8
bool (C99)	uchar	8	8
_Bool (C99)	uchar	8	8
bool (C++)	uchar	8	8
short、signed short	int16	16	16
unsigned short	uint16	16	16
int、signed int	int32	32	32
unsigned int	uint32	32	32
long（32 位），signed long	int32	32	32
unsigned long	uint32	32	32
long（40 位）	int40	40	64
long long、signed long long	int64	64	64
unsigned long long	uint64	64	64
enum	--	不尽相同（请参阅节 2.8）	32
float	float32	32	32
double	float64	64	64
long double	float64	64	64
指针	--	32	32

此规范中使用的表中的通用名称以与语言无关的方式标识类型。

列出的对齐方式值是默认值。它们适用于所有情况，但在“紧凑”结构中使用类型时除外。“紧凑”结构内的所有结构成员类型都具有 8 位对齐方式。

char 类型带符号。

整数类型具有互补无符号变体。通用名称以“u”为前缀（例如 uint32）。

bool 类型使用值 0 表示 false，1 表示 true。其他值未定义。

在以前用于 C6000 的 COFF ABI 中，C 类型 *long* 是 40 位整数，对应于原始 62x 硬件的最长原生整数类型。默认情况下，EABI 将 long 更改为 32 位，以与通用不成文约定兼容。但是，工具链可能希望通过支持可选的 40 位类型（指定为 long 或其他类型）来支持与 COFF ABI 下开发的旧代码的兼容性，因此本文将介绍表示法。请注意，该类型的大小为 40 位，但对齐方式为 64 位。如果将 40 位 *long* 类型用于“紧凑”结构的成员，则该类型具有 8 位对齐方式，但容器大小为 64 位。

C、C99 和 C++ 中的其他类型被定义为标准类型的同义词：

```
typedef unsigned int     size_t;
typedef int              ptrdiff_t;
typedef unsigned int     wchar_t;
typedef unsigned int     wint_t;
typedef char *           va_list;
```

2.2 寄存器中的数据

一般来说，实现可自由使用其认为合适的寄存器。本节中指定的标准寄存器表示仅适用于传递给函数或从函数返回的值。

大小为 32 位 或更小的对象可以驻留在单个寄存器中。

寄存器中的数值始终右对齐；也就是说，寄存器的位 0 包含该值的最低有效位。小于 32 位 的有符号整数值将符号扩展到寄存器的高位。小于 32 位 的无符号值将加零扩展。

大小在 32 至 64 位之间的对象使用寄存器对。寄存器对由偶数寄存器和下一个连续奇数寄存器组成，偶数寄存器保存值的最低有效部分，下一个连续奇数寄存器保存最高有效部分。寄存器对表示为 R_o:R_e，其中，R_o 是奇数寄存器，R_e 是偶数寄存器（例如，A5:A4）。寄存器对中的数值右对齐到偶数寄存器中；也就是说，偶数寄存器的位 0 包含该值的最低有效位，奇数寄存器的位 0 包含该值的位 32。有符号整数值符号扩展到奇数寄存器的高位。无符号值进行加零扩展。

大于 64 位的对象没有指定的寄存器表示。

2.3 存储器中的数据

C6000 可配置为在大端或小端字节序模式下运行。字节序是指多字节值的存储器布局。在大端字节序模式下，值的最高有效字节存储在最小地址中。在小端字节序模式下，最低有效字节存储在最小地址中。字节序仅影响对象的存储器表示；无论字节序如何，寄存器中的标量值始终具有相同表示形式。字节序确实会影响结构和位字段的布局，并且会延续影响它们的寄存器表示。

需对齐标量变量，以便可使用适合其类型的本机指令来加载和存储：LDB/STB 表示字节、LDH/STH 表示半字、LDW/STW 表示字，依此类推 。这些指令正确地考虑了进出存储器时的字节序。

40 位整数有 5 个字节，指定为 0 (LSB) 到 4 (MSB)。在存储器中，40 位值填充为 64 位（8 字节）。在存储器中，如果值的地址为 N，则 图 2-1 给出存储布局：

<table>
<tr><th>location</th><th>little-endian</th><th>big-endian</th></tr>
<tr><td>N</td><td>byte 0 (lsb)</td><td rowspan="3">padding</td></tr>
<tr><td>N+1</td><td>byte 1</td></tr>
<tr><td>N+2</td><td>byte 2</td></tr>
<tr><td>N+3</td><td>byte 3</td><td>byte 4 (msb)</td></tr>
<tr><td>N+4</td><td>byte 4 (msb)</td><td>byte 3</td></tr>
<tr><td>N+5</td><td rowspan="3">padding</td><td>byte 2</td></tr>
<tr><td>N+6</td><td>byte 1</td></tr>
<tr><td>N+7</td><td>byte 0 (lsb)</td></tr>
</table>

图 2-1. 存储器中 40 位值的表示

2.4 复数类型

C99 标准规定其 _Complex 类型的布局和对齐等效于相应浮点类型的两元素数组，实部作为第一个元素，虚部作为第二个元素。这使得 ABI 的灵活性较差。相应地，复数类型的 C6000 表示法如表 2-2 所示：

表 2-2. 复数类型

类型	通用名称	大小	对齐	外部对齐
float _Complex	complex32	64	32	64
double _Complex	complex64	128	64	128
long double _Complex	complex64	128	64	128

如果变量具有复数类型或具有外部可见性的复数数组，则变量具有比其类型要求更严格的对齐要求。表格的外部对齐列为此类变量提供了最小对齐方式。

2.5 结构体和联合体

结构体成员会被分配从 0 开始的偏移量。每个成员会被分配满足其对齐要求的最低可用偏移量。成员之间可能需要进行填充，以便满足此对齐约束。

联合体成员全部被分配 0 偏移量。

C++ 类的底层表示是一个结构体。在本文档的其他地方，术语*结构体* 也适用于类。

结构体或联合体的对齐要求等同于其成员中最严格的对齐要求，包括下一节中所述的位字段容器。通过在最后一个成员之后插入填充，存储器中的结构体或联合体大小将向上舍入为其对齐的倍数。如节 3.3 中的规定，在栈上按值传递的结构体和联合体具有特殊的对齐规则。

大小为 64 位或更小的结构体在传递到函数或从函数返回时，可能驻留在寄存器或寄存器对中。

在小端模式下，寄存器中的结构体始终右对齐；也就是说，第一个字节占用寄存器（如果是寄存器对，则为偶数寄存器）的 LSB，结构体的后续字节将填充到寄存器中递增的有效字节中。

在大端模式下，寄存器中结构体的布局遵循以下规则：

- 一个 1 字节结构体占用单个寄存器的 LSB。
- 对于 2 字节结构体，第一个字节占用寄存器的字节 1，第二个字节占用字节 0 (LSB)
- 对于 3 字节或 4 字节结构体，第一个字节占用寄存器的字节 3 (MSB)，其余字节向着 LSB 填充寄存器。3 字节结构体在寄存器的 LSB 中具有一个填充字节。
- 对于 5 字节至 8 字节结构体，第一个字节占用高位（奇数）寄存器的字节 3 (MSB)，其余字节填充递减的有效字节。5 字节至 7 字节结构体在低位（偶数）寄存器的 LSB 中有填充字节。

图 2-2 描绘了大小为 1 字节至 8 字节结构体的大端寄存器表示。

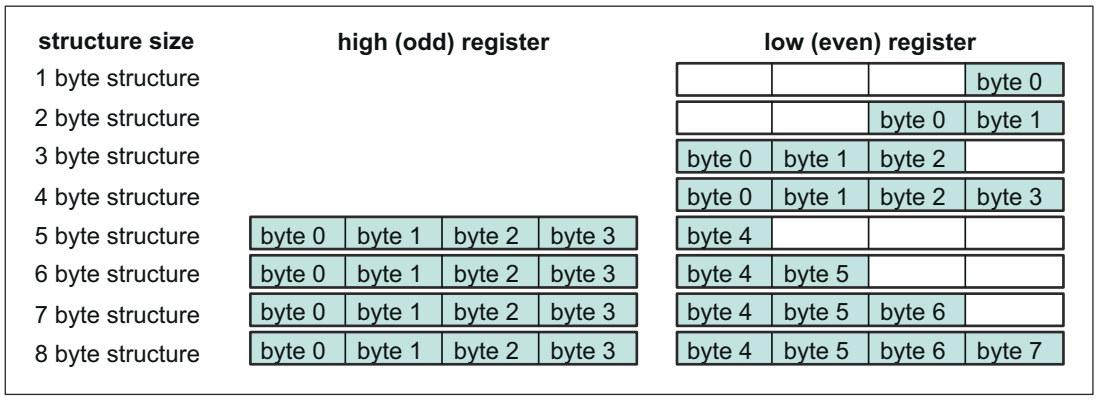


图 2-2. 寄存器中结构体或联合体的大端布局

这种布局的基本原理是，允许使用尽可能最小的存储器引用，在寄存器和内存之间复制结构体；例如，2 字节结构体使用 16 位引用，3 字节结构体使用 32 位引用，5 字节结构体使用 64 位引用，依此类推。结构体在所包含存储器引用的大小范围内左对齐。

2.6 数组

对于具有*数组*类型的对象，其最小对齐方式是由其元素类型指定的。

具有外部可见性的文件作用域数组变量具有更严格的要求。根据目标 ISA，此类变量的对齐方式是其元素对齐方式的最大值，并且适用于：

C62x、C67x	4 字节
所有其他	8 字节

2.7 位字段

C6000 EABI 采用来自 IA64 C++ ABI 的位字段布局。除非明确指出，否则以下说明与该标准一致。

位字段的**声明类型**是出现在源代码中的类型。为了保存位字段的值，C 和 C++ 标准允许实现分配任何大得足以容纳位字段的*可寻址存储单元*，这不需要与声明类型相关。可寻址存储单元通常称为*容器类型*，我们在本文档中也这样称呼它。容器类型是位字段压缩和对齐方式的主要决定因素。

C89、C99 和 C++语言标准对于声明类型有不同的要求：

C89	int、unsigned int、signed int
C99	int、unsigned int、signed int、_Bool 或“其他一些实现定义类型”
C++	任何整型或枚举类型，包括 bool

严格 C++ 中没有 *long long* 类型，但由于 C99 包含该类型，C++ 编译器通常支持该类型作为扩展。C99 标准不要求实现支持位字段的 long 或 long long 声明类型，但由于 C++ 允许这种实现，因此 C 编译器也支持这种类型的情况并不少见。

位字段的值完全包含在其容器中，不包含任何填充位。容器根据其类型正确对齐。包含字段的结构对齐方式受到容器对齐方式的影响，影响方式与该类型的成员对象相同。这也适用于未命名字段，这与 IA64 C++ ABI 不同。容器可以包含其他字段或对象，并且可以与其他容器重叠，但为任何一个字段保留的位，包括用于超大字段的填充位，绝不会与另一个字段的位重叠。

在 C6000 EABI 中，位字段的容器类型是其声明类型，但有一个例外。C++ 允许所谓的超大位字段，这些字段的声明大小大于声明类型。在这种情况下，容器是不大于字段的声明大小的最大的整型。

布局算法保持*下一个可用位*，这是分配位字段的起点。布局算法中的步骤如下：

1. 如前面所述，确定容器类型 T。
2. 假设 C 是包含下一个可用位的 T 类型正确对齐的容器。C 可能与以前分配的容器重叠。
3. 如果可以在 C 中分配该字段，则从下一个可用位开始执行分配。
4. 否则，在下一个正确对齐的地址分配新的容器，并将该字段分配到该容器。
5. 增加位字段的大小，包括超大字段的填充位在内，以确定下一个可用位。

在小端字节序模式下，容器从 LSB 向 MSB 填充。在大端字节序模式下，容器从 MSB 向 LSB 填充。

零长度位字段强制结构的以下成员对齐到与声明类型对应的下一个对齐边界，并影响结构对齐。

C6000 EABI 将 plain *int* 的声明类型视为 *signed int*。

2.7.1 易失性位字段

易失性位字段是用 C *volatile* 关键字声明的字段。当读取易失性位字段时，必须使用其整个容器的适当访问权限仅读取一次它的容器。

当写入自身大小小于其容器的易失性位字段时，必须使用适当的访问，仅读取一次和写入一次它的容器。读取和写入不需要彼此是原子形式。

当写入自身大小完全等于容器大小的易失性位字段时，未规定是否进行读取。由于未规定这类读取，因此对于采用不同实现方式编译的目标文件，如果二者都写入与自身容器宽度完全相同的易失性位字段，则将这些目标文件互连是不安全的。因此，应避免在外部接口中使用易失性位字段。

不能将对同一易失性位字段或同一容器内附加易失性位字段的多次访问合并。例如，增加一个易失性位字段时，必须始终实现为两次读取和一次写入。即使位字段的宽度和对齐允许使用较窄类型进行更高效的访问，这些规则也适用。对于写入操作，即使容器的全部内容将被替换，也必须进行读取。如果两个易失性位字段的容器重叠，则对其中一个位字段的访问也将导致对另一个位字段的访问。

例如，给定以下结构：

```
struct S
{
   volatile int a:8;       // container is 32 bits at offset 0
   volatile char b:2       // container is 8 bits at offset 8
};
```

对“a”的访问也会导致对“b”的访问，但反之不适用。如果非易失性位字段的容器与易失性位字段重叠，则未定义对非易失性字段的访问是否会导致访问易失性字段。

2.8 枚举类型

枚举类型（C 类型 *枚举*）是使用基础整型来表示的。通常情况下，基础类型为 int 或 unsigned int，除非两者都不能表示所有枚举器，在这种情况下，基础类型为 long long 或 unsigned long long。有符号版本和无符号版本都可表示所有值时，ABI 支持实现在两种替代方案中进行选择。（需要不同工具链之间保持一致的应用程序可通过声明负枚举器来确保选择有符号的替代方案。）

C 标准要求枚举常量适合“signed int”类型，因此，在严格 ANSI 模式下，枚举类型只能是 int 或 unsigned int。在 C++ 中可实现更广泛的枚举类型。TI 编译器还允许在宽松模式和 GCC 模式下使用更广泛的枚举类型。

章节 3

调用约定

本章介绍函数调用的约定，包括返回值、寄存器和实参传递的行为。

3.1 调用和返回

C6000 具有不同的指令，可使用它们来根据 ISA 变体和调用环境影响函数调用。在任何情况下，通过在寄存器 B3 中保存返回地址并跳转到被调用函数来执行函数调用。被调用函数通过执行间接分支到在它被调用时存在于 B3 中的地址来返回。

3.1.1 返回地址计算

在 64x 目标上，调用是一个双指令序列：ADDKPC 指令使用 PC 相对寻址将返回地址计算到 B3 中，后跟 B（分支）指令。

```
          ADDKPC return_label,B3     ; B3 := return_label
          B      func                ; goto func
   return_label:
```

在非 64x 目标上，可以使用其他方法计算地址，例如绝对寻址、PC 相对寻址或基于 GOT 的寻址（如节 5.1 中所述）。

3.1.2 调用指令

调用本身生成为一个简单的相对于 PC 的分支，用于将控制权转移给被调用者：

```
          B      func                ; goto func
```

位移是一个 21 位带符号字偏移量。如果无法到达调用目标，链接器会生成一个蹦床函数，它是一个存根函数，使用绝对寻址、相对于 PC 寻址或 GOT 间接寻址来寻址目标函数。更多有关蹦床函数的信息，请参阅节 5.3。

对于间接调用，目标是一个寄存器：

```
          B      reg                 ; goto address in reg
```

对于实现调用的分支，TI 工具链使用 **CALL** 伪指令，它编码为分支，但对调试信息进行批注，以便分析器、调试器或其他分析工具可以将该指令识别为函数调用（请参见节 5.3）。因此，先前的直接调用实际上会在汇编源代码中显示为：

```
          CALL   func                ; encodes as B func
```

C64+ ISA 有一个复合指令 CALLP，它单独实现如下的一个调用。

- 将下一个执行数据包的地址加载到 B3 作为返回地址
- 用于填充分支的延迟时隙的 NOP 5

使用 CALLP 的调用只需：

```
          CALLP func,    B3
```

3.1.3 返回指令

通过分支到在 B3 中传递的地址来执行函数返回。被调用函数可以移动此地址并将其存储在其他位置（通常是嵌套调用需要的位置）。如果此地址仍在 B3 中，指令将为：

```
          B      B3                  ; return
```

如果此函数是中断处理程序函数，则必须分支到 IRP：

```
          B      IRP                 ; return from interrupt handler
```

TI 工具链使用 **RET** 伪指令来指定实现函数返回的分支。

3.1.4 流水线约定

在 C6000 上，分支指令（包括调用或返回）的获取与其执行周期之间有五个延迟时隙。可以在延迟时隙中调度指令，但要遵守以下规定：调用者负责确保可能影响被调用者的所有指令的效果在被调用者的第一条指令的 E1 阶段之前完成。类似地，对于返回指令，被调用者负责确保所有可能影响调用者的指令在返回地址处的指令的 E1 阶段之前完成。

3.1.5 弱函数

弱函数是自身符号具有绑定 STB_WEAK 的函数。在不定义弱函数的情况下，程序可以成功链接，使对它的引用保持不解析状态。

链接器以不同方式处理未解析的弱函数调用，具体取决于使用静态链接还是动态链接。

使用静态链接时，链接器会将对未定义弱函数的调用替换为实际上的 NOP。但为了实现优化，比如被调用者不返回调用点的尾调用消除，替换必须保留调用行为的某些方面。因此，不使用 NOP 进行替代，而是使用替代返回指令：

```
        B.S2    B3                  ; replacement for unresolved weak call
```

这种行为对弱函数调用提出了以下额外要求：

- S2 功能单元必须可用。如果在 S2 上对原始调用进行编码，则可以轻松确保这一点。
- 当调用指令到达流水线的 E1 阶段时，返回地址必须在 B3 中可用。换句话说，编译器无法在调用的延迟时隙中调度返回地址计算。

  ABI 支持调用导入的弱函数；也就是可能在其他不同静态链接单元中定义的函数。

使用动态链接时，全局偏移表 (GOT) 用于确定要使未解析的弱函数保留 0 值，还是在 _ _c6xabi_weak_return() 的地址中进行修补，后者会直接返回到其调用者。如果使用 0 值（对于 Linux 平台），则需要保护。如果使用 _ _c6xabi_weak_return() 函数（对于裸机平台），则不需要保护。

3.2 寄存器惯例

C6000 至少有 32 个通用 32 位寄存器。寄存器可以包含整数、浮点值或指针。通用寄存器分为两个寄存器文件，指定为 A 和 B。

B15 指定了栈指针 (SP)。栈指针必须在 2 字（8 字节）边界上保持对齐。SP 指向低于（小于）当前分配栈的第一个对齐地址（请参阅 节 4.3）。

B14 指定为数据页指针 (DP)。它指向当前活动对象的数据段的开头。

ABI 不指定专用的帧指针 (FP) 寄存器。不过在某些情况下，TI 编译器使用 A15 作为帧指针。

GCC 支持词法嵌套函数作为语言扩展。该实现使用一个寄存器（即，静态链寄存器）为子函数提供父函数的激活上下文。寄存器的选择在很大程度上取决于工具链，除非以某种方式（例如由蹦床函数）调解调用。因此，ABI 指定 A2 作为静态链寄存器的推荐选择。调用约定通过将 A2 包含为函数链接所含的一个寄存器来支持此指定，要求在调用点和被调用者的入口点之间保留其值。

ABI 将 A10-A15 和 B10-B15 指定为*被调用者保存*寄存器。也就是说，需要由被调用的函数保留上述寄存器，确保其在从函数返回时具有与调用时相同的值。请注意，该组包括 SP (B15) 和 DP (B14)。

此外，ILC 和 RILC 是被调用者保存的寄存器。这些是 C64+ 的 SPLOOP 机制使用的控制寄存器。

所有其他寄存器都是*调用者保存*寄存器。也就是说，这些寄存器不会在调用中保留，因此如果调用后需要它们的值，调用者负责保存和恢复它们的内容。

地址模式寄存器 (AMR) 是用户可写的控制寄存器，可实现循环寻址。在函数调用边界处，AMR 的位 0-15 必须为 0，以便禁用循环寻址。

表 3-1 列出了寄存器及其在 ABI 中的作用。

表 3-1. C6000 寄存器约定

寄存器	别名	由被调用者保留	在调用惯例中的作用
A0		否	
A1		否	
A2		否	嵌套函数的静态链寄存器
A3		否	通过引用返回结构体的地址
A4		否	第一个实参；返回值 (LSW)
A5		否	第一个实参；返回值 (MSW)
A6		否	第三个实参 (LSW)
A7		否	第三个实参 (MSW)
A8		否	第五个实参 (LSW)
A9		否	第五个实参 (MSW)
A10		是	第七个实参 (LSW)
A11		是	第七个实参 (MSW)
A12		是	第九个实参 (LSW)
A13		是	第九个实参 (MSW)
A14		是	
A15	FP	是	帧指针
A16-A31		否	
B0		否	延迟绑定的动态重定位偏移实参；请参阅节 15.6
B1		否	延迟绑定的动态重定位偏移实参；请参阅节 15.6
B2		否	
B3		否	返回地址
B4		否	第二个实参 (LSW)
B5		否	第二个实参 (MSW)
B6		否	第四个实参 (LSW)
B7		否	第四个实参 (MSW)
B8		否	第六个实参 (LSW)
B9		否	第六个实参 (MSW)
B10		是	第八个实参 (LSW)
B11		是	第八个实参 (MSW)
B12		是	第十个实参 (LSW)
B13		是	第十个实参 (MSW)
B14	DP	是	数据页指针
B15	SP	是	栈指针
B16-B29		否	
B30-B31		否！	蹦床函数暂存寄存器；请参阅节 3.7

3.3 实参传递

函数的前 10 个实参在寄存器中传递。实参按声明的顺序分配给以下序列中的寄存器（在四倍字寄存器中传递的实参除外）：

A4、B4、A6、B6、A8、B8、A10、B10、A12、B12

大小在 32 位到 64 位之间的实参在寄存器对中传递，其中使用先前列表中的偶数寄存器传递其最低有效部分，而使用相应的奇数寄存器传递其最高有效部分。例如，在以下示例中，“a”在 A4 中传递，而“b”在 B5:B4 中传递：

```
func1(int a, double b);
```

float complex 类型的实参在寄存器对中传递。顺序取决于字节序。在小端字节序模式下，实部在偶数寄存器中传递，虚部在奇数寄存器中传递。对于大端字节序模式，此顺序相反。

double complex 类型的实参在四倍字寄存器中传递，并使用以下列表中的第一个可用四倍字：A7:A6:A5:A4、B7:B6:B5:B4、A11:A10:A9:A8、B11:B10:B9:B8。在小端字节序模式下，实部在编号较小的对（例如 A5:A4）中传递，而虚部在编号较大的对 (A7:A6) 中传递。对于大端字节序模式，此顺序相反。四倍字寄存器实参绕过的任何寄存器都可用于后续实参。例如，在以下函数中，“w”在 A4 中传递，“x”在 B4 中传递，“y”在 A11:A10:A9:A8 中传递，而“z”回填到 A6 中：

```
func2(int w, int x, double complex y, int z);
```

所有其余的实参都以递增的地址（从 SP+4 开始）放置在堆栈上。每个实参都放置在下一个与其类型正确对齐的可用地址处。因此，如果第一个堆栈实参需要 64 位对齐，则其地址将为 SP+8。

在 C++ 中，*this* 指针作为隐式第一个实参传递给 A4 中的非静态成员函数。

大小为 64 位或更小的结构体和联合体在寄存器或堆栈上通过值传递，如下表所述。大于 64 位的结构体和联合体通过引用传递，如节 3.5 所述。

任何未在寄存器中传递的实参都会以递增的地址（从 SP+4 开始）放置在堆栈上。每个实参都放置在下一个可用地址处，并根据其类型正确对齐，但需考虑以下其他因素：

- 标量的栈对齐方式是其所声明类型的栈对齐方式。
- 无论成员要求何种对齐，值传递结构体的栈对齐都是大于或等于其大小的 2 最小幂。（此值不能超过 8 个字节，这是通过值传递的结构体的最大允许大小。）
- 每个实参都保留一定数量的栈空间，其大小等于其四舍五入为其栈对齐的下一个倍数。

请注意，SP+4 不是 8 字节对齐，因此如果第一个实参需要 8 字节对齐，它将存储在存储器的 SP+8 处。

对于可变实参的 C 函数（即，用省略号声明，表明它是使用不同数量的实参调用的函数），最后一个显式声明的实参和所有剩余的实参都在栈上传递，以便其栈地址可以作为访问未声明实参的参考。

根据 C 语言，小于整数值的可变实参函数的未声明标量实参被提升为整数值并作为整数值传递。

3.4 返回值

大小为 32 位或更少的标量和结构体在 A4 中返回。32 位与 64 位之间的标量和结构体在 A5:A4 中返回。

float complex 类型的对象在 A5:A4 中返回，其中实部位于奇数寄存器中，虚部位于偶数寄存器中。

返回的 double complex 类型的对象的实部位于 A5:A4 中，虚部位于 A7:A6 中。

大于 64 位的聚合体通过引用返回。

3.5 通过引用传递并返回的结构体和联合体

大于 64 位的结构体（包括类）和联合体通过引用进行传递和返回。

若要通过引用传递结构体或联合体，调用方需将其地址放置在适当的位置：根据其在实参列表中的位置，放在寄存器中或栈中。为了保留值传递语义（C 和 C++ 的要求），被调用者可能不修改指向对象；它必须制作自己的拷贝。

如果被调用的函数返回大于 64 位的结构体或联合体，调用者必须在 A3 中传递一个附加实参，其中包含返回值的目标地址，或者，如果未使用返回值，则返回 NULL。

被调用者通过将对象复制到 A3 中的地址（如果非零）来返回对象。如果需要，调用方负责分配存储器。通常，这涉及在栈上保存空间，但在某些情况下，可以传递已经存在的对象的地址，而无需分配。例如，如果 f 返回一个结构体，则可以通过在 A3 中传递 &s 来编译赋值 s = f()。

3.6 编译器辅助函数的约定

ABI 指定了*辅助函数*，编译器使用这些函数来实现语言功能。通常，这些函数遵循标准调用约定，但为提高性能，对其中的几个函数进行了例外处理。有关使用修改版约定的辅助函数，请参阅节 8.3 。

3.7 段间调用的暂存寄存器

当调用者保存寄存器在调用期间处于活动状态，但被调用者不修改该寄存器时，编译器可以省略调用前后的保存和恢复，从而优化调用者函数代码。当出现该定义或调用辅助函数如节 8.3 中所述具有特殊约定时，就会出现这种情况。

但是，寄存器 B30 和 B31 被指定为可能被任何跨越段边界的调用修改，即使出现了该定义或在调用辅助函数时也是如此。这样一来，如果调用需要 far 调用蹦床函数（节 5.3），则 B30 和 B31 可用作蹦床函数中的暂存寄存器。

此外，Linux 的延迟绑定机制要求调用者保存寄存器可供实现惰性绑定的存根函数使用。当被调用者可能被导入并因此可能通过延迟绑定调用时，编译器不得优化调用点。

同一段内的调用绝不需要蹦床函数；对于此类段内调用，B30 和 B31 的处理方式与其他调用者保存寄存器没有区别。

3.8 设置 DP

在 DSBT 模型下共享对象编译的导出函数可能需要在进入时设置 DP，并在退出时恢复。节 6.7.1 中讨论了这一点。

章节 4

数据分配和寻址

本章介绍数据存储的约定。ABI 定义的数据段如 图 4-1 所示。

4.1 数据段和数据区段

在编译器或汇编器输出的可重定位目标文件中，使用默认规则和编译器指令将变量分配到各个段中。段是可重定位文件中不可分割的分配单元。段通常包含具有类似属性的对象。为数据指定了各种段，具体取决于该段是否已初始化、是可写还是只读的、如何寻址，以及包含的数据类型。

ABI 将静态数据段指定为 *near* 或 *far*。可使用高效的 near DP 相对寻址来寻址 near 段，但其大小和位置受到限制。有关静态变量在段中的位置以及如何寻址这些变量的约定，请参阅 节 4.2.2。

链接器将来自目标文件的段组合起来，形成 ELF 加载模块（可执行文件或共享库）中的区段。区段是分配给加载模块的连续存储器范围，表示程序执行映像的一部分。

加载模块可包含一个或多个数据区段，链接器在其中分配栈、堆和静态变量。项可分组为单个区段或多个区段，仅受以下限制：

- 必须对采用 near DP 相对寻址来访问的所有段进行分组，使其位于静态基址的无符号 15 位寻址范围内，该范围由 _ _C6000_DSBT_BASE 定义。
- 给定区段内的所有数据都具有相同的区段属性（请参阅 章节 19）。
- 在区段内，初始化数据必须位于未初始化数据之前。这是 ELF 的结构约束。
- 平台特定约定施加的任何其他限制。

如果使用 DP 相对寻址来访问区段，则将该区段指定为 DP 相对。单个 DP 相对区段可包含 near 寻址和 far 寻址混合形式，只要满足上文所列限制即可。

运行时环境可动态分配或调整未初始化的数据区段，以便为栈和堆等项分配空间。

图 4-1 展示了 ABI 定义的数据段，以及段到区段的抽象映射。该映射仅是代表性的；具体配置可能因平台或系统而异。初始化段为蓝色阴影；未初始化段为灰色阴影。

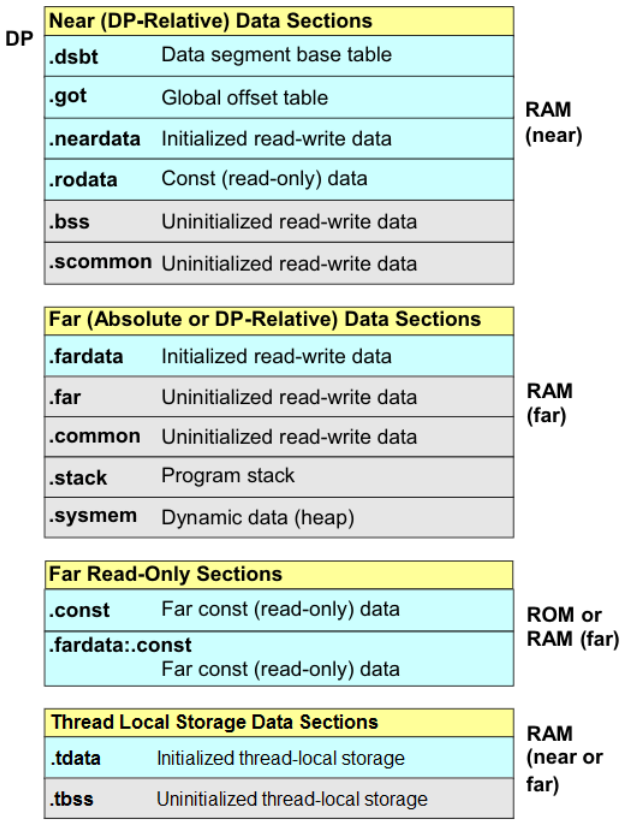


图 4-1. 数据段和数据区段（典型值）

.const 和 .fardata:.const 段包含只读常量。.const 段包含与位置无关的常量。根据平台的不同，.const 段可位于只读存储器中，并且可使用绝对寻址来寻址，或者在位置无关模型中通过 PC 相对寻址相对于代码来寻址。在一些共享只读段的平台（例如 Linux）上，无法共享初始化值包含地址常量的 const 对象。因此，将其置于名为 .fardata:.const 的不同段中，这样命名是因为可视其为数据区段中 .fardata 的一部分。

.rodata 段包含可通过 near DP 相对寻址来寻址的只读常量。

.neardata 和 .fardata 段包含初始化的读写变量。这些段对应于其他架构中常见的 .data 段。

.bss 和 .far 段包含未初始化变量。

.common 和 .scommon 段包含链接器分配的通用块符号。这些并不是目标文件中的实际段。相反，段名称是链接器命令文件中用于放置变量的约定。这些段不应用于其他目的。

.got 和 .dsbt 段包含与动态链接相关的数据结构。请参阅章节 6。

节 13.3.5 中列出了可由链接器命令文件放置的其他特殊段。

4.2 静态数据的分配和寻址

所有非自动或动态的变量都被视为静态数据；也就是说，具有 C 存储类 *extern* 或 *static* 的变量，其地址在（静态或动态）链接时确定。这些变量根据其属性被分配到各个段中，然后组合成一个或多个静态数据段。

指定为 *DP 相对段* 的数据段使用 DP 相对寻址进行寻址。在进入加载模块中的任何代码时，DP 会被初始化以指向加载模块中具有最低地址的 DP 相关段的进程私有副本。链接器定义符号 _ _C6000_DSBT_BASE 以指向该地址。

DP 相对寻址有两种形式。当 DP 相对偏移量可以作为 15 位无符号常量编码到单个指令中时，适用 Near DP 相对寻址。当无法编码到单个指令中，而需要额外的指令时，则适用 Far DP 相对寻址。使用 near 形式对变量进行寻址时，变量的放置限制在 DP 的 32KB 范围内。

DP 相对段由程序头文件中的 PF_C6000_DPREL 标志标识（请参阅节 14.1）。

某些平台（尤其是 Linux）可能会将加载模块限制为具有不超过一个 DP 相关段。

包含静态变量的其他数据段称为*绝对数据段*，并使用绝对寻址或基于 GOT 的寻址进行寻址。它们的数量、大小或放置位置没有限制。

如果程序是动态链接的并且具有共享库，则每个加载模块的数据段都独立于其他加载模块的数据段。具体来说，每个加载模块都有自己的数据段，包括 DP 相对段，因此也有自己的 DP。如果多个可执行文件共享一个库，则每个可执行文件都将获得该库中数据段的私有副本。在没有虚拟地址转换的情况下用于管理多个数据段的模型称为 DSBT 模型，如 节 6.7 中所述。

4.2.1 静态数据的寻址方法

ABI 支持以下用于静态数据寻址的基本方案：DP 相对、绝对、GOT 间接和 PC 相对。在给定情况下具体使用哪一个方案取决于多种因素，包括变量的声明、执行平台、 模块是构建为可执行库还是共享库、可见性约定等。由于编译器生成寻址，因此它必须了解此上下文，通常是通过源代码中的命令行选项和/或可见性指令来实现。此 ABI 的其他部分提供了有关每种寻址形式*何时* 适用的详细信息；本部分说明了*如何* 执行寻址。

4.2.1.1 near DP 相对寻址

这是 near DP 段中静态变量的默认寻址。DP 偏移量是一个 15 位无符号值，它将这种寻址形式限制到 32KB DP 内的对象。

```
        LDW     *+DP(sym),dest              ;reloc R_C6000_SBR_U15_W (also _B and _H)
```

4.2.1.2 Far DP 相对寻址

这是一种与位置无关的寻址 *far* 数据（即 near DP 区段以外的区段中的数据）的方法。使用 2 条 MVK 指令将 32 位 DP 相对偏移量加载到寄存器中，然后使用索引寻址将其添加至 DP。必须根据访问大小适当缩放偏移量。TI 工具链使用特殊汇编语言运算符来指示比例因子。

```
        MVKL    $DPR_word(sym),tmp        ;reloc R_C6000_SBR_L16_W
        MVKH    $DPR_word(sym),tmp        ;reloc R_C6000_SBR_H16_W
        LDW     *+DP(tmp),dest
        MVKL    $DPR_hword(sym),tmp       ;reloc R_C6000_SBR_L16_H
        MVKH    $DPR_hword(sym),tmp       ;reloc R_C6000_SBR_H16_H
        LDH     *+DP(tmp),dest
        MVKL    $DPR_byte(sym),tmp        ;reloc R_C6000_SBR_L16_B
        MVKH    $DPR_byte(sym),tmp        ;reloc R_C6000_SBR_H16_B
        LDB     *+DP(tmp),dest
```

4.2.1.3 绝对寻址

以下指令使用绝对寻址。

```
        MVKL    sym,tmp                   ;reloc R_C6000_ABS_L16
        MVKH    sym,tmp                   ;reloc R_C6000_ABS_H16
        LDW     *tmp,dest
```

由于这种寻址模式会对地址进行编码，因此它与位置相关。它可用于访问 far 数据。与前文所述的 Far DP 相关方案相比，实际访问的成本相同，但计算变量 (&sym) 的地址不需要添加 DP，从而节省了一条指令。这种情况下没有调节，因为加载的常量是实际地址，而不是偏移量。

4.2.1.4 GOT 间接寻址

全局偏移量表 (GOT) 是一种与位置无关的机制，用于动态解析在静态链接时无法知道的地址。GOT 中的地址由动态加载器解析。节 6.6 中讨论了基于 GOT 的寻址。

4.2.1.5 PC 相对寻址

这是对代码段中的 *far* 数据进行寻址的方法，此方法与位置无关。假定数据位于距访问数据的代码的（链接时）恒定偏移处。例如 switch 语句的标签表，以及可放入代码段的只读常量变量 (.const)。无论是寻址代码还是数据，寻址机制均相同；节 5.1 对此进行了说明。

4.2.2 静态数据的放置约定

工具链之间的互操作性要求一个工具链生成的寻址与另一工具链生成的放置一致，特别是对于 near DP 相对寻址。使用 near DP 相对寻址进行寻址的任何变量都必须分配在位于 32KB DP 以内的段中。

这就需要 ABI 建立一些约定。其中一些约定取决于特定于工具链的行为，例如支持的代码生成模型，甚至是用户行为，例如选择的命令行选项或应用的语言扩展。为此，ABI 采取了双管齐下的方法：

- 为了实现一致性，ABI 定义了一些有关放置和寻址的抽象约定，这些约定以特定于工具链的方式映射到工具链行为。通过这些约定，可以使用不同的工具链构建兼容的目标文件，但无法确切指明如何做到这一点。
- 为了强制一致性，ABI 要求链接器要么以满足寻址约束的方式链接程序，要么拒绝链接程序。

生成寻址的工具链仅对变量的声明可见，而对其定义不可见。因此，约定必须仅基于这两点的可用信息。这不包括例如阵列维度的使用。

4.2.2.1 放置的抽象约定

抽象约定将变量指定为 near 或 far，如下所示：

- 无论任何变量，只要使用特定于工具链的关键字、属性或 pragma 声明并指定为 near 或 far，都会使用该指定。
- 无论任何变量，只要是使用特定于工具链的关键字、属性或 pragma 声明位于 .bss、.rodata 或 .neardata 以外的段中，都会被指定为 far。
- 任何其余变量都会根据三个模型之一指定，通常由命令行选项控制。

- Near 模型 — 未以其他方式指定的所有变量均被指定为 near
- Far 模型 — 未以其他方式指定的所有变量均被指定为 far
- Far 聚合模型 — 具有标量类型的变量被指定为 near；具有聚合类型的变量（即数组、类、结构体和联合体）被指定为 far。这应该是工具链的默认模型。

工具链可以支持其他模型，但必须至少支持这三种模型。如果使用其他模型，则可能会也可能无法实现与其他工具链的互操作性。

如果指定依赖于工具链特定方面（如命令行选项或语言扩展），程序员有责任在声明变量的地方始终使用这些构造，但需要由链接器来捕获错误（请参阅 节 4.2.2.3）。

ABI 建立了变量到段的传统赋值。变量的赋值是其 near/far 指定及其初始化类别的函数，由以下列表中的第一个匹配条件确定。

- 如果变量没有初始化值，则该变量未初始化，或在启动时通过构造函数调用进行初始化。
- 如果变量的类型符合常量条件，则变量为常量。
- 如果变量具有初始化值，则该变量已进行初始化。

表 4-1 列出了传统的段赋值：

表 4-1. 变量到段的传统赋值

	初始化类别		
名称	**未初始化**	**已初始化**	**常量**
	.bss	.neardata	.rodata
far	.far	.fardata	.const

传统赋值可以通过特定于工具链的方式被覆盖。例如，变量可以分配给用户定义的段。但是，工具链不得允许用户将指定为 far 的变量放置到三个 near 段中的任何一个中。

4.2.2.2 寻址的抽象约定

变量的寻址方式取决于其指定为 near 还是 far、其可见性以及代码生成模型（例如，与位置无关还是与位置相关）。

只有指定为 near 的对象才能使用 near DP 相对寻址进行寻址。near 对象也可以通过其他方式寻址，例如使用绝对寻址（与位置相关）或通过 GOT（与位置无关），但这些方式都与 near 放置不一致。

指定为 far 的变量不能使用 near DP 相对寻址进行寻址。

4.2.2.3 链接器要求

链接器负责确保使用 near DP 相对寻址来寻址的变量放置在所需 15 位 DP 范围内，如建立的 _ _C6000_DSBT_BASE 符号。链接器可检测由 R_C6000_SBR_* 重定位条目标记的此类访问。如果链接器不能满足该约束（可能是由于用户指令冲突），则必定无法链接程序。

4.2.3 静态数据的初始化

具有初始非零值的静态变量应分配到初始化数据段中。该段的内容应为对应于该段中所有变量初始值的存储器内容的映像。因此，当该段加载到存储器中时，变量会直接获取自己的初始值。这就是大多数基于 ELF 的工具链使用的所谓*直接初始化模型*。

可将预期将初始化为零的变量分配至未初始化段。加载器负责归零数据区段末尾的未初始化空间。

虽然编译器需要直接对初始化变量进行编码，但链接器不需要。链接器可将目标文件中直接编码的初始化段转换为可执行文件的编码格式，依靠库函数来解码信息并在程序启动时执行初始化。（由前文可知，链接器可能假设该库来自同一工具链。）编码初始化数据有助于节省可执行文件中的空间，还为不依赖加载器的基于 ROM 的自引导系统提供了初始化机制。TI 工具链实现了此类机制，如 章节 18 中所述。其他工具链可采用兼容机制、不同机制，或者不采用任何机制。

4.3 自动变量

过程的局部变量，即具有 C 存储类 *auto* 的变量，由编译器自行分配到堆栈上或寄存器中。堆栈上的变量通过栈指针 (B15) 或者在偏移量过大的情况下通过指向激活帧并支持更大偏移量的临时帧指针寄存器 (A15) 进行寻址。

堆栈从 .stack 段分配，并且是程序数据段的一部分。

堆栈从高地址向低地址增长。栈指针必须始终在 2 字（8 字节）边界上保持对齐。SP 指向低于（小于）当前分配栈的第一个对齐地址。

节 4.4 更详细地介绍了堆栈约定和本地帧结构。

4.4 帧布局

至少有两种情况需要标准化布局局部帧并且对由被调用方保存的寄存器进行排序，它们是异常处理和调试。

本节介绍用于以下情形的约定：管理栈、帧的一般布局、由被调用方保存区域的布局。

栈向零增长。SP 指向最上面分配字上方的字；也就是说，分配了 *(SP+4) 处的字，但未分配 *SP 处的字。

使用具有正偏移量的 SP 相对寻址来访问帧中的对象。

编译器可自由分配一个或多个“帧指针”寄存器来访问帧。TI 编译器 使用 A15 作为帧指针 (FP)。如已分配 FP，则其值是创建函数帧之前的 SP 值。也就是说，FP 指向当前帧的底部和调用方帧的顶部。帧中的对象是通过具有负偏移量的 FP 来访问的。通过具有正偏移量的 FP 来访问传入实参。

如果帧指针不是函数之间链接的一部分，则由工具链自行决定选择是否使用帧指针、使用哪个寄存器，以及指向何处。然而，一些用于栈展开的虚拟指令假设 A15 指向帧，如上一段所述。如果函数没有帧指针，或者对于使用哪个寄存器或其指向的位置采用不同的约定，则无法使用这些展开指令，并且可能需要效率较低的序列。

函数的栈帧包含以下区域：

- 在栈上传递的**传入实参**是调用方帧的一部分。
- **由被调用方保存的区域**存储由函数修改且必须保留的寄存器。如果启用了异常或调试，则必须遵循特定布局。如未启用，则编译器可自由使用替代方案来保存寄存器。
- **局部变量和溢出临时变量**区域由函数使用的临时存储器组成。
- **传出实参**段用于将非寄存器实参传递给调用函数，详见 节 3.3。该段的大小是任何单个调用所需的最大值。

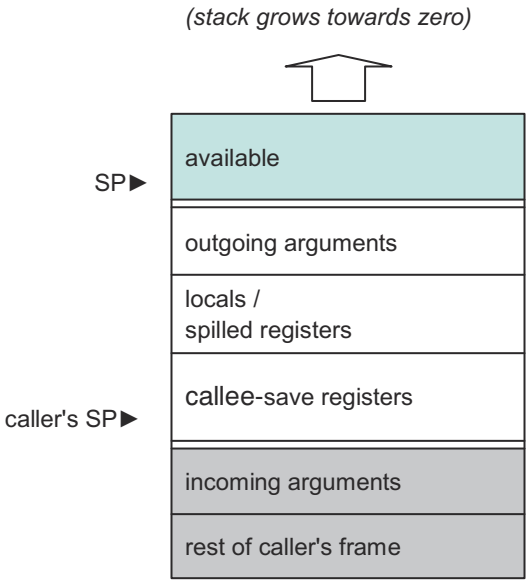


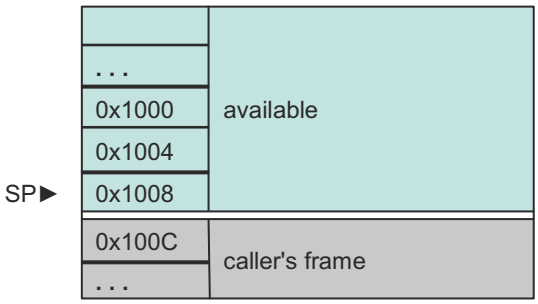
图 4-2. 局部帧布局

4.4.1 栈对齐

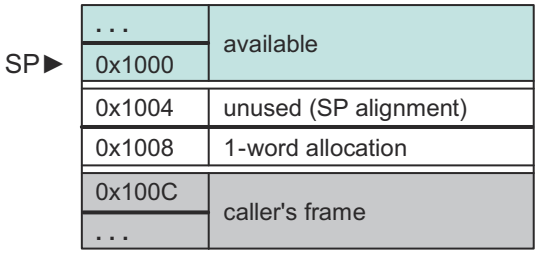
SP 为 8 字节对齐，并且必须始终保持 8 字节对齐，以防在帧分配或释放期间发生中断。这意味着对 SP 的每次原子调整都必须是 8 字节的倍数。

帧底部的双字（8 字节）跨越帧边界。也就是说，第一个字位于被调用者的帧中，但第二个字位于调用者的帧中，因此二者都不能使用它来存储双字。从使用双字加载和存储来保存和恢复寄存器的角度看，这让人遗憾，但这是以前不支持双字的架构的历史遗留问题。在下图中，双字边界用较粗的线表示。

在函数中的第一条指令之前，栈如下所示：



如果此函数需要栈上的一个字来存储某些内容，则需要分配 2 个字的帧（因为 SP 必须始终保持 8 字节对齐）。执行分配的方法是将 SP 递减 8。现在栈如下所示：



4.4.2 寄存器保存顺序

如节 3.2 所述，函数负责保留指定为*被调用者保存*的寄存器内容，这通常是通过在进入函数时将修改后的寄存器保存在本地帧并在退出前将其恢复来完成的。通常，被调用者保存的寄存器在栈上的顺序和位置并不重要，只要它们从保存时的相同位置恢复即可。在大多数情况下，编译器以任意顺序保存寄存器。但是，有一些功能需要已知的顺序：

- **安全调试**。当启用符号调试（通常由 -g 选项指示）时，将应用安全调试约定。在此模式下，编译器以固定顺序在栈上保存和恢复寄存器。
- **异常处理**。用于异常处理的栈回溯过程需要确切知道每个寄存器的位置，以便模拟函数收尾程序。为了使用位矢量高效地对此信息进行编码，我们定义了一个固定顺序。异常处理重复使用*被调用者保存的寄存器安全调试顺序*对位矢量进行编码，因此顺序通常相同，但有一些例外，如下所示。

被调用者保存的寄存器安全调试顺序为 A15、B15、B14、B13、B12、B11、B10、B3、A14、A13、A12、A11、A10。

当使用安全调试并且没有特殊栈布局时（请参阅节 4.4.3 和节 4.4.4），编译器将始终按照该相对顺序保存寄存器，从帧的底部（最高地址）开始。如果未保存任何寄存器，则寄存器将打包，使栈中没有空洞，但相对顺序保持不变。

4.4.2.1 大端字节序对交换

对于具有双字（64 位）LDDW 和 STDW 指令的目标，更有效的方法是保存属于排列在堆栈上的偶数对的寄存器，以便可以使用一个 LDDW 读取该对。请注意，小端字节序的安全调试排序通常会放置寄存器，使得这样做是

正确的；这并非完全出于巧合。但是，对于大端字节序，每个对的顺序都需要反向。在为大端字节序编译时，编译器会查找在栈上占用相同对齐的双字的寄存器对，并交换顺序。尽管排序方式与小端字节序不同，大端字节序的顺序可能因保存不同寄存器的函数而异，但这仍然被视为安全的调试顺序。即使在不支持 LDDW 或 STDW 的 C6x 目标上也会发生这种交换。

请记住，将寄存器放置在堆栈时首先参考安全调试排序；仅当偏移量可被 8 均匀整除时，给定偶数对的排序才会被交换。如果保存偏移量未对齐，那么寄存器将按原始顺序单独保存。

4.4.2.2 示例

如果所有 13 个由被调用方保存的寄存器都是由为 C62x 编译的函数来保存的，则保存区域如下图所示。大端字节序一列中的粗体条目表示交换对。

SP►	*little-endian*	*big-endian*
. . .	rest of callee's frame	
0x1004		
0x1008	A10	**A11**
0x100C	A11	**A10**
0x1010	A12	**A13**
0x1014	A13	**A12**
0x1018	A14	A14
0x101C	B3	B3
0x1020	B10	**B11**
0x1024	B11	**B10**
0x1028	B12	**B13**
0x102C	B13	**B12**
0x1030	B14	**B15**
0x1034	B15	**B14**
0x1038	A15	A15
0x103C	caller's frame	
. . .		

图 4-3. 函数保存所有由被调用方保存的寄存器时的 C62x 保存区域

如果仅保存寄存器 B13、B12、A12、A11 和 A10：

SP►	*little-endian*	*big-endian*
. . .	rest of callee's frame	
0x1004		
0x1008	A10	**A11**
0x100C	A11	**A10**
0x1010	A12	A12
0x1014	B12	B12
0x1018	B13	B13
0x103C	caller's frame	
. . .		

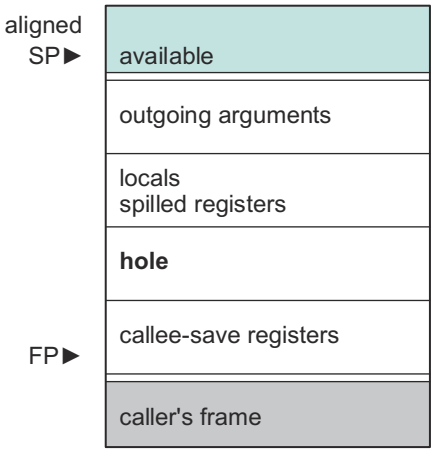
图 4-4. 仅保存寄存器 B13、B12、A12、A11 和 A10 时的 C62x 保存区域

请注意，B13:B12 未交换，因为其偏移量未正确对齐。

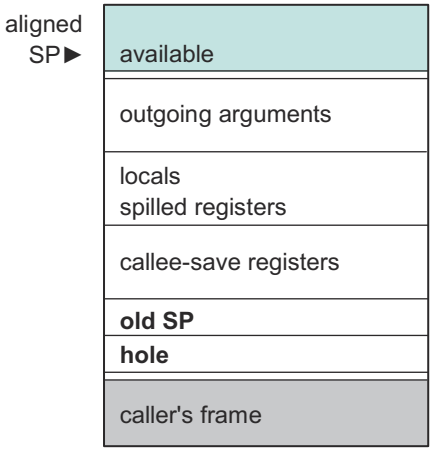
4.4.3 DATA_MEM_BANK

pragma DATA_MEM_BANK 在栈上创建孔洞，以保证局部变量的特定对齐。为实现该目的，它将旧 SP 存储在栈中并清除 SP 的低位。

如已分配 FP，则只需从 FP 中恢复 SP，因而无需在栈中保存旧 SP：



若尚未分配 FP，则必须在栈中保存 SP 的旧值：



4.4.4 C64x+ 特定的堆栈布局

为了以激进的方式减小代码量，C64x+ 和 C674x 上的某些函数使用了特殊的堆栈布局。

4.4.4.1 _ _C6000_push_rts 布局

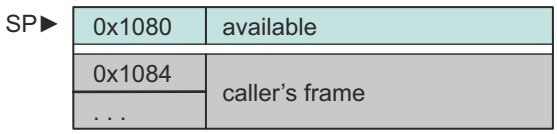
许多函数会保存和恢复所有被调用者保存的寄存器，而执行此操作的代码相当大。不必在每个函数的逻辑程序和收尾程序中都包含执行此操作的代码，而是可以调用运行时库中的函数。这些函数使用特殊的调用约定来避免损坏将要保存的寄存器。保存所有被调用者保存的寄存器的调用如下所示：

```
        CALLP __C6000_push_rts, A3      ; CALLP puts the return address in A3
```

恢复它们的代码是：

```
        CALLP __C6000_pop_rts, A3       ; A3 is unused
                    ; Returns to the location saved from B3 by __C6000_push_rts
```

在调用 _ _C6000_push_rts 之前，堆栈如下所示：



_ _C6000_push_rts 会存储所有被调用者保存的寄存器，从而得到：

SP▶

	little-endian	*big-endian*
. . . 0x1048	available	
0x104C	unused (SP alignment)	
0x1050	X (pushed with B3)	B3
0x1054	B3	X (pushed with B3)
0x1058	A10	A11
0x105C	A11	A10
0x1060	B10	B11
0x1064	B11	B10
0x1068	A12	A13
0x106C	A13	A12
0x1070	B12	B13
0x1074	B13	B12
0x1078	A14	A15
0x107C	A15	A14
0x1080	B14	B14
0x1084 . . .	caller's frame	

4.4.4.2 紧凑帧布局

为了鼓励使用可压缩指令，某些 C64x+ 函数的寄存器保存/恢复代码略有不同。

通常，编译器使用一个 SP 递减分配整个帧，然后使用相对于 SP 的写入保存被调用者保存的寄存器。

```
    STW   B14,     *SP--[12]
    STDW  A15:A14, *SP[5]
    STDW  A13:A12, *SP[4]
    STDW  A11:A10, *SP[3]
    STDW  B13:B12, *SP[2]
    STDW  B11:B10, *SP[1]
```

在*紧凑帧*模式下，编译器改成为每个寄存器对生成一系列 SP 自动递减存储。

```
    STW   B14,     *SP--[2]
    STDW  A15:A14, *SP--
    STDW  A13:A12, *SP--
    STDW  A11:A10, *SP--
    STDW  B13:B12, *SP--
    STDW  B11:B10, *SP--
```

对于这种情况，堆栈布局是相同的。但是，对于不同的已保存的寄存器子集，堆栈布局可能不同。例如，假设我们需要保存 A10、A11、B10、B11 和 B3，我们选择使用紧凑帧布局以减小代码量。传统布局将最有效地利用堆栈空间：

```
    STW   A11, *SP--[6]
    STW   A10, *+SP[5]
    STW   B3,  *+SP[4]
    STW   B11, *+SP[3]
    STW   B10, *+SP[2]
```

SP▶	0x1050	available
	0x1054	unused (SP alignment)
	0x1058	B10
	0x105C	B11
	0x1060	B3
	0x1064	A10
	0x1068	A11
	0x106C	caller's frame

但是，为了使用更多可压缩 SP 递减指令，紧凑帧布局将在寄存器保存区域中留下多个空洞。由于每次压入必须将 SP 递减 8（以确保中断安全），编译器会尝试将寄存器对的成员压在一起；如果无法做到，则必须将单个寄存器压入一个双字中，因而在保存区域中形成空洞。

```
    STW   A11, *SP--[2]
    STW   A10, *SP--[2]
    STDW  B11:B10, *SP--
    STW   B3, *SP--[2]
```

SP▶	... 0x1048	available
	0x104C	unused (SP alignment)
	0x1050	B3
	0x1054	unused
	0x1058	B11
	0x105C	B10
	0x1060	A10
	0x1064	unused
	0x1068	A11
	0x106C	caller's frame

4.5 堆分配对象

动态分配对象由运行时库分配，比如通过 C 的 malloc () 或 C++ 运算符“new”。执行环境可提供自己对这些函数的实现，只要这些函数符合语言标准指定的 API。该 ABI 不会对动态分配机制指定任何附加要求。

章节 5

代码分配和寻址

编译器和汇编器将代码生成到一个或多个节中。默认代码节称为 .text，但程序员可以将代码定向到其他命名节中。链接器将代码节组合成一个或多个段。尽管可能存在平台特定的限制，但基本 ABI 对代码节的数量、大小或放置没有限制。

除了 C64x+ 的紧凑指令编码格式之外，C6000 上的所有指令都是 32 位宽。表示函数地址的标签以及大多数其他标签始终在 32 位边界上对齐。有关紧凑指令的注意事项，请参阅节 5.4。

可以通过三种方式引用代码对象：计算其地址、作为分支目标或将其作为函数调用。

5.1 计算代码标签的地址

汇编代码节需要计算代码地址以便：

- 执行调用或分支
- 创建函数指针
- 构成调用的返回地址
- 填充切换表
- 在链接器生成的蹦床函数或 PLT 条目中使用

有三种方法可以构成代码对象的地址：绝对寻址、相对于 PC 寻址和基于 GOT 寻址。绝对寻址与位置相关；相对于 PC 和 GOT 形式与位置无关。

5.1.1 代码的绝对寻址

基本方法是将目标简单地编码为绝对常量：

```
          MVKL label, B5       ; B5 := lower 16 bits of label
          MVKH label, B5       ; B5 := upper 16 bits
```

对这类常量进行编码的任何代码都会直接变得与位置相关，从而具有不需要的特性，如果重定位，比如在加载时，就需要进行修补。

5.1.2 PC 相对寻址

这是对代码（或代码段中的常量数据）进行寻址的方法，此方法与位置无关。地址计算为当前取指数据包的地址与常量之和。

```
   base:  MVC   PCE1,tmp1                      ; address of current fetch packet
          MVK   $PCR_OFFSET(label,base),tmp2   ; label-base, reloc R_C6000_PCR_L16
          MVKH  $PCR_OFFSET(label,base),tmp2   ; label-base, reloc R_C6000_PCR_H16
          ADD   tmp1,tmp2,tmp2                 ; &label
```

$PCR_OFFSET 汇编运算符用于计算在包含指令的取指数据包（由基址 (MVC) 标记）与目标符号之间的偏移量。

5.1.3 同一段内的 PC 相对寻址

当引用的标签与引用位于同一段时，偏移量是汇编时常量。如果偏移量可以用 15 位或更少位数进行编码，则可以使用 ADDK 将其直接添加到基址：

```
   base:  MVC PCE1,tmp                       ; address of current fetch packet
          ADDK  label-(base & ~0x1F),tmp     ; no reloc; tmp == &label
          ...
   label: ; must be in same section
```

这里不需要重定位；汇编器直接对偏移量进行编码。表达式“(base &~0x1F)”代表包含基址的取指数据包的地址。

（如果偏移量太大而无法使用 ADDK 进行编码，则必须按照节 5.1.2 所述使用 MVK/MVKH/ADD。）

5.1.4 短偏移 PC 相对寻址 (C64x)

C64x 和较新的 ISA 具有优化的指令，用于对附近的标签实现 PC 相对寻址：

```
          ADDKPC label, B5             ; B5 := label, position independent
```

此形式将标签限制在取指数据包（包含 ADDKPC 指令）的有符号 7 位常量偏移量（+/- 64 个字）以内。

ADDKPC 对计算返回地址很有用，返回地址通常只差几条指令。

5.1.5 基于 GOT 的代码寻址

用于计算代码地址的另一种与位置无关的形式是从全局偏移量表中加载代码地址，如 节 6.5 中所述。

5.2 分支

分支始终假定为在同一个函数内，因此始终可以使用相对于 PC 寻址，且解析时间不超过静态链接时间。

编码使用放大 2 倍的 21 位带符号偏移量，产生的范围为 $\pm 2^{22}$ 字节 (4MB)。这有效地将任何给定函数的大小限制为 4MB。

5.3 调用

C6x 没有特定的调用指令。[1] 通过在寄存器 (B3) 中生成返回地址并执行分支来生成调用。节 3.1 中介绍了返回地址计算。对指定函数的直接调用会生成相对于 PC 的分支，因此受限于 4MB 限制。

5.3.1 直接 PC 相对调用

如果直接调用的目标函数放置在直接 CALL 指令中偏移量不可到达的位置，则静态链接器重写 CALL 指令，以便改为调用名为*蹦床函数*的辅助存根函数。蹦床函数只调用目标函数。链接器负责将蹦床函数放置在 CALL 指令的范围内。

```
        CALL   sym                      ; reloc R_C6000_PCR_S21
```

注意：CALL 是伪操作；该指令编码为 B。

5.3.2 Far Call Trampoline

如果调用目标是在同一静态链接单元中定义的，但使用 21 位字偏移量是无法到达，静态链接器就会生成蹦床函数，它是一个使用备用寻址形式来到达目标的存根函数。该示例说明了使用绝对寻址的蹦床函数：[2]

```
   $Tramp$$sym2:
        MVKL sym,tmp                ;reloc R_C6000_ABS_L16
        MVKH sym,tmp                ;reloc R_C6000_ABS_H16
        B tmp
```

或者，蹦床函数可使用 节 5.1 中所述的其他寻址形式来计算目标函数的地址。

如果未在同一静态链接单元中定义调用目标，静态链接器就会生成一个类似于蹦床函数的 PLT 条目。节 6.5 介绍了这种情况。

根据所使用的寻址，蹦床函数通常需要一两个临时寄存器（例如前一个序列中的 tmp）。

对于 C64x 和较新的目标，B30 和 B31 可供任何蹦床函数使用。防止调用方假设 B30 和 B31 由调用保留，即使已知所调用函数不能修改它们（请参阅 节 3.7）。

对于较旧的 C62 和 C67 目标，蹦床函数必须保存和恢复它们使用的任何寄存器。

5.3.3 间接调用

通过函数指针进行的间接调用会生成带寄存器操作数的分支。例如：

5.4 寻址紧凑指令

C64+ ISA 以及一些更高版本均具有称为紧凑指令的功能，这是一种将 16 位指令对打包到程序存储器的 32 位字中的编码格式。在大端字节序模式下，指令以与其在源程序中的词法顺序相反的顺序存储在存储器中，并且与它们的执行顺序相反。本节阐明了针对这种差异表示地址的约定。

一条 16 位指令可以被视为具有两个地址：

[1] C64x+ 具有 CALLP，其结合了 ADDKPC、B 和 NOP 5。
[2] 蹦床函数标签的名称可能因约定而异。

- 其**物理地址**是它在程序存储器中物理驻留的地址。
- 其**逻辑地址**是从程序控制流的角度来看它所驻留的地址。逻辑地址可以被视为是对应于指令的程序计数器值。程序按照逻辑地址顺序执行指令，该顺序与程序的词法顺序相对应。分支目标和位移都根据逻辑地址来计算。

在小端字节序配置中，逻辑地址与物理地址相同。在大端字节序中，16 位指令对会在程序存储器中交换，因此，如果地址 A 表示包含该对的 32 位字的物理地址，则第一个逻辑指令存储在 A+2 中，而第二个指令存储在 A 中。

图 5-1 中的代码片段展示了逻辑地址和物理地址之间的区别。程序按其词法顺序显示。第一列显示小端字节序物理地址，这也是小端字节序和大端字节序的逻辑地址。第二列显示大端字节序物理地址。虚线表示程序存储器中的 32 位边界。

Physical Address		Opcode	Source Code		
Little-Endian	Big-Endian				
			code:		
0000	0002	40CE		MV.S1	A1,A2
0002	0000	41B0	||	ADD.L1	A2,A3,A3
0004	0006	58E7		NEG.L2	B1,B1
0006	0004	614F	||	MV.S2	B2,B3
0008	000A	814F		MV.S2	B2,B4
			local:		
000A	0008	A14F		MV.S2	B2,B5
000C	000E	C14F		MV.S2	B2,B6
000E	000A	40CE		MV.S1	A1,A2
0010	0012	41B0	||	ADD.L1	A2,A3,A3
0012	0010	58E7	||	NEG.L2	B1,B1

图 5-1. 寻址紧凑指令

ABI 指定目标文件中的所有程序地址均表示为逻辑地址。这包括分支位移、符号值、展开表中的地址和调试信息中的地址。

参考图 5-1，尽管所标记的指令（操作码 0x40CE）存储在 0x0002，但是符号表中标号 code 的值为 0x0000。同样，尽管所标记的指令（操作码 0xA14F）存储在 0x0008，但是标号 local 的值为 0x000A。

在大多数情况下，逻辑地址和物理地址之间的区别对于程序员和工具链都是透明的。为了保持这种透明度，规定了以下条件：

1. 分支必须跳转至有标号指令。可以构造指向无标号指令的分支，但结果是未定义的。
2. 除一些例外情况之外，标号必须在 32 位边界上对齐。例外包括：
   a. 可确定不是分支目标的标号（例如 DWARF 标号）
   b. 作为从一个获取数据包到另一个获取数据包的段内分支目标的标号（BNOP 的紧凑形式对半字偏移进行编码）。
3. 可重定位字段不能出现在 16 位指令中。

条件 (1) 和 (2) 加在一起，将间接分支排除到非 32 位对齐的地址。请注意，(2b) 的实例不需要重定位，因为偏移量是汇编时常量。这反过来实现了条件 (3)，因而不需要在逻辑地址和物理地址之间进行转换来访问可重定位字段。

章节 6

动态链接的寻址模型

在为裸机环境构建独立程序的最基本场景中，程序会被静态链接并绑定到特定地址运行。链接器只需使用最终解析的地址修补所有引用，程序便可以运行。这种场景简单而高效。

甚至嵌入式系统也越来越多地由多个单独链接的组件组成。这自然导致了通用系统上普遍采用动态链接模型：Windows 上的动态链接库 (DLL) 或基于 Unix 平台（包括 Linux）上的动态共享对象 (DSO)。本节介绍了一组 C6000 的基本级别动态链接和共享对象机制约定。有关与动态链接相关的目标文件机制，请参阅节 14.3。特定执行平台（如 Linux）可能会指定其他约定；请参阅章节 15。

6.1 术语和概念

静态链接 是将可重定位目标文件与静态库组合为*静态链接单元* (ELF *可执行文件* (.exe) 或 ELF *共享对象* (.so)) 的传统过程。在本文档中，我们使用术语 *加载模块* (或简称模块) 来指代静态链接单元，并使用 *共享库* (或库) 来指代共享对象。

确切来说，*程序* 包含一个可执行文件，以及它赖以满足任何未定义引用的其他共享库。如果多个可执行文件依赖同一个库，它们可以共享其代码的单个拷贝 (因而是共享对象中共享) ，从而显著降低系统的存储器要求。

当加载此类包含多个对象的程序时，必须解析其组件模块 (其中一些可能已作为其他应用的一部分进行加载) 之间的引用。此过程称为*动态链接*，由一个称为*动态链接器* 的运行时组件处理。由于系统状态会随着在给定时间加载哪些对象而异，因此动态链接器可能希望动态控制其存储器分配和放置。分配程序位置，然后在加载时将其重定位，这一能力有时称为*动态加载*。尽管动态链接和动态加载在某种程度上是两种独立的功能，因为其中任何一个都可以离开对方使用，但用于实现它们的机制却密切相关。在本文档中，我们使用术语“动态链接”来指代复合功能，同时使用可互换的术语“动态链接器”和“动态加载器”，指代执行这些操作的组件。

对象*自身的* 函数和变量 (统称*符号*) 是在其内部定义的函数和变量。当模块 (可执行文件或库) 引用在该模块内未定义但在另一个模块中定义的符号时，称为*导入* 该符号。定义模块称为*导出* 符号。

一般来说，动态链接模块的代码和数据的地址在静态链接时是未知的。此外，任何导入符号的地址也是未知的，直到它们被动态链接器解析为止。因此，当动态链接器加载模块时，它可能需要根据其分配的地址以及它导入的任何符号的地址，修补它的代码和/或数据。动态链接时执行的重定位称为*动态重定位*。大多数动态链接机制的设计目标是，尽可能减少动态重定位的数量和复杂性。动态重定位以及关联的符号信息包含在 ELF 目标文件的特殊段中。

共享库的一个基本问题是，共享同一库的每个可执行文件仍必须有自己的 ***专用*** (非共享) 库数据拷贝。这意味着共享代码不能使用绝对寻址来访问数据。术语*位置无关数据 (PID)* 适用于以可共享方式访问数据的代码，这一访问通常通过相对寻址或基于 GOT 的寻址来实现。

广义的术语*位置无关代码 (PIC)* 是指不以任何方式使用绝对寻址的代码，因此与自身的放置和其他负载模块的放置无关。位置无关代码不需要对代码段进行加载时修补，从而可以加快加载时间和/或允许其位于 ROM 中。典型的 PIC 方法依赖 PC 相对寻址、虚拟存储器、间接寻址和/或来自基址指针寄存器的相对寻址，如 C60 的 DP (B14)。

对于将位于 ROM 中的模块，需要额外考虑一点。显然，ROM 中的代码无法在加载时修补，因此它具有许多相似的位置无关性要求。

6.2 动态链接机制概述

ABI 通过若干相关机制解决这些问题：

通用 ***ELF 动态链接*** 机制定义目标文件表示，以支持加载时符号解析和重定位。其中大部分与目标无关，由 GABI 指定。节 14.3 说明了特定于目标的方面。

过程链接表 (PLT) 条目 是链接器生成的存根，用于解析对导入函数的调用。

全局偏移表 (GOT) 是引用导入对象的寻址方法，它支持位置无关性和私有性，将地址常量置于数据段的表中，而不是将其编码到代码中。这些好处的代价是，基于 GOT 的引用需要额外的间接寻址，表也需要额外的数据空间。

数据段基表 (DSBT) 模型 是一种软件约定，允许每个组件拥有自己的专用数据段，因此可以静态解析对其自身数据的引用，而无需考虑其他组件。DSBT 机制无需虚拟存储器即可实现与位置无关的代码，从而使共享代码的单个实例能够寻址动态绑定私有数据的多个副本。

6.3 DSO 和 DLL

系统在其支持的动态链接模型方面各不相同。在 UNIX 系统 (包括 Linux) 中，动态链接旨在从应用程序的角度来看是透明的。也就是说，可编写和编译程序或库，而不考虑是否静态或动态地解析任何未解析的引用。例如，如果程序声明“extern int f()”，然后调用 f，则编译器会生成能够静态或动态解析 f 的代码。这种方法的主要优点是

灵活性：可以编写和编译程序，而无需考虑其将如何链接。主要缺点是效率可能比较低，因为编译器必须假设可能无法静态解析任何外部引用，并且生成适合的寻址代码来支持动态链接。

UNIX 将动态链接的库称为动态共享对象或 *DSO*。

在 Windows 和各种嵌入式系统（例如 Symbian 和 PalmOS）中，动态链接是通过语言扩展（通常是 _ _declspec(import)）在符号声明的源代码中显式指定的。这种方法的优点是编译器明确知道何时生成所需特殊寻址。这些系统通常具有链接后阶段，该阶段用符号索引方案来替代通过符号引用的动态链接。这些系统将共享库称为动态链接库或 *DLL*。

6.4 抢占

当一个对象引用在另一对象中定义的全局符号时，此对象为导入符号，定义的对象为导出符号。假设两个不同对象定义和导出同一符号。其中一个定义优先于并*抢占* 另一定义。抢占使动态链接的行为与静态链接相同：可执行文件的定义抢占库的定义，因此不会链接库的实例。在动态链接情况下，库可能已加载，并且只有一个客户端需要共享实例中的定义。

抢占意味着即使一个符号在静态链接时似乎在模块内定义，但实际上它可能在动态链接时替换为不同定义。这会对编译器产生影响，编译器必须生成代码，就像导入符号一样。因此，抢占的成本很高（即使它实际并未发生）。节 6.8 讨论了性能影响。Linux 使用称为“作为自有导入”的技术（如节 15.9 所述）来减轻可执行文件的损失。

ELF 符号表中的符号可见性字段指示符号的可抢占性。标记为 STV_HIDDEN 或 STV_INTERNAL 的符号不会导出（因此不可抢占）。标记为 STV_PROTECTED 的符号将导出，但无法被抢占。标记为 STV_DEFAULT 的符号可以被抢占。

特定于平台和工具链的不同约定适用于哪些符号可以被抢占以及编程器如何指定可见性。

6.5 PLT 条目

通常，当编译器看到对 extern 函数的调用时，它仅生成一条 CALL 指令，而不考虑被调用函数的位置。在静态链接期间，如果函数是在另一源文件或在静态链接库中定义的，则链接器只需重定位 CALL 指令中的位移字段即可解析引用。

如果函数是从共享库导入的，则其地址在静态链接时未知，最终在动态链接时解析。可能需要额外指令来寻址和调用导入的函数。出于这种可能性，以及为了避免必须在动态链接时修补调用，静态链接器会生成与位置无关的存根来调用此函数，并修补初始调用以执行此存根。此存根称为 *PLT 条目*。PLT 表示*过程链接表*。（将 PLT 指定为表是历史做法；它的条目是独立生成的代码片段，不会收集到任何内聚实体中。）PLT 条目在概念上类似于 far 调用蹦床函数（请参阅节 3.1）。蹦床函数旨在调用 *far-away* 函数，而 PLT 条目调用的是*导入*函数。

6.5.1 直接调用导入函数

在发生调用的代码区段中生成 PLT 存根。PLT 根据 节 5.1 中的注意事项来编码目标函数的地址。

6.5.2 通过绝对地址寻址的 PLT 条目

```
$sym$plt:
        MVKL   sym,tmp                  ;reloc R_C6000_ABS_L16
        MVKH   sym,tmp                  ;reloc R_C6000_ABS_H16
        B      tmp
```

除了在下一节中讨论的一个细微差别（涉及 tmp 选择），此代码序列与 far 调用蹦床函数相同。

6.5.3 通过 GOT 寻址的 PLT 条目

如果函数可以被抢占，则函数的地址无法在 PLT 条目中编码，即使以与位置无关的方式编码也是如此。此地址必须通过 GOT 间接寻址。

```
$sym$plt:
        LDW    *+DP($GOT(sym)),tmp      ;reloc R_C6000_SBR_GOT_U15
        B      tmp
```

某些编译器辅助函数遵循非标准寄存器保留约定（节 8.3），这会影响选择将哪个寄存器用于 tmp。此外，除 PLT 条目直接提及的寄存器外，延迟绑定（节 15.6）可能会影响其他寄存器。因此，ABI 规定不能导入遵循非标准约定的函数；也就是说，这些函数不能通过 PLT 条目调用。根据此规定，链接器可以随时修改 PLT 条目中的函数调用接口未涉及的任何调用者保存寄存器。

编译器可以选择内联 PLT 条目，以调用它已知或怀疑已导入的函数。这样做的好处是可以减少附加分支的延迟，但会增加代码量。

如果动态加载器使用 节 15.6 所述的延迟绑定，则内联 PLT 条目必须遵循其中所述的约定。或者，内联 PLT 可以生成动态重定位表的 DT_JMPREL 部分中未包括的 GOT 重定位（请参阅 System V ABI 第 5 章的*动态段*），以免其受到延迟绑定的影响。

6.6 全局偏移表

完全位置无关意味着代码与自身位置无关，与自身数据的位置以及任何导入代码或数据的位置无关，无需在加载时进行重定位修补。在此上下文中，“自身”一词表示与引用同属于相同的静态链接单元。我们来看一看每种情况意味着什么：

- 引用自身代码（节 5.1）：必须使用 PC 相对寻址或基于 GOT 的寻址。不能使用绝对地址。这种情况会影响蹦床函数、switch 表和返回地址的计算。
- 引用自身数据（节 4.2）：必须使用 DP 相对寻址、PC 相对寻址或基于 GOT 的寻址。不能使用绝对地址。通常，必须在编译时做出选择。这种情况会影响对 near 和 far 数据的引用。
- 引用导入代码：不能使用绝对地址或 PC 相对地址。此情况适用于 PLT 条目中生成的调用。
- 引用导入数据：不能使用绝对地址或 DP 相对地址。此情况适用于对导入数据的任何引用。

为避免将位置相关绝对地址编码到代码段中，它们会生成到称为全局偏移表 (GOT) 的表中，该表是每个静态链接单元的数据段的一部分。程序不直接访问对象，而是从 GOT 读取符号的地址并间接对其寻址。GOT 是数据段的一部分，始终使用静态链接时固定的偏移量来进行 DP 相对寻址。它由链接器生成，以响应编译器发出的特殊 GOT 生成重定位。当地址已知时，GOT 中的地址会在动态链接时得到修补。

基于 GOT 的访问涉及两个存储器引用：一个从 GOT 加载地址，另一个引用变量本身。第一个引用将访问 GOT 自身，本质上与正常的 DP 相对数据访问相同（请参阅节 4.2.1）。绝大多数时候，我们期望 GOT 位于 near DP 段中，因此可以使用 near DP 相对寻址进行访问。

6.6.1 使用 Near DP 相对寻址的基于 GOT 的引用

使用 near DP 相对寻址形式的完整的基于 GOT 的引用如下所示：

```
        LDW   *+DP($GOT(sym)),tmp     ;reloc R_C6000_SBR_GOT_U15
        LDW   *tmp,dest
```

此处指示的重定位使得静态链接器分配 GOT 条目并计算其 DP 相对偏移量。表条目本身标有动态重定位，其计算结果为符号地址。

6.6.2 使用 Far DP 相对寻址的基于 GOT 的引用

为确保完整性，当 GOT 本身较远（即超出 DP 的 15 位偏移量范围）时，ABI 还支持基于 GOT 的寻址。在这种情况下，使用 far DP 相对寻址来到达 GOT：

```
        MVKL  $DPR_GOT(sym),tmp       ;reloc R_C6000_SBR_GOT_L16
        MVKH  $DPR_GOT(sym),tmp       ;reloc R_C6000_SBR_GOT_H16
        LDW   *+DP[tmp],tmp2
        LDW   *tmp2,dest
```

6.7 DSBT 模型

每个共享同一库代码的可执行文件必须分配自己的库数据专用拷贝。此外，每个静态链接单元的自有数据（包括 GOT）使用 DP 相对寻址进行寻址，同时使用静态链接时固定的偏移量。（在具有 MMU 的系统上，为完成这一点，通常会使用 PC 相对寻址来实现位置无关虚拟偏移，并使用地址转换在同一（虚拟）地址，将数据段的多个物理拷贝实例化。）对于没有 MMU 的系统，如 C6000，通常依赖某种静态基址指针 (DP) 和偏移寻址。

给定静态链接单元的所有寻址均与它的数据段相对，因此与任何其他静态链接单元无关。结果会得到一个模型，其中由可执行文件和一个或多个（可能是共享的）库组成的给定程序具有多个数据段，每个数据段具有不同的地址，而 DP 相对偏移量便基于这些地址。当控制从一个模块转移到另一个模块时，DP 必须更改为新模块数据段的基础地址。

此模型的一般问题，同时也是大多数静态基址寻址方案所常见的问题是：

- 谁更改 DP：调用者还是被调用者
- 新 DP 值如何确定
- 如何处理间接调用

其他架构采用了各种解决方案，如 FDPIC、XFLAT 和 DSBT。我们选择了采用 DSBT 模型，作为效率、兼容性和灵活性之间的最佳折衷。

当调用导入函数时，被调用者负责将 DP 设置为指向其数据段（更准确地说，是包含被调用者的模块的底层可执行文件的专用数据段拷贝），并负责在返回后立即将其恢复。

在我们解释被调用者如何实现这一点之前，请考虑两个观察结果。首先，每个模块都有自己的数据段，以及自己的基地址，如果在多个可执行文件之间共享，则每个模块还有该段的专用拷贝，以及不同的基地址。此外，这些地址是动态确定的。显然，与存储在 GOT 中的地址非常相似的是，基地址不能是绝对地址，因此必须存储在数据段中。

其次，虽然*被调用者*负责更改 DP，但进入*被调用者*后，DP 仍然指向调用者的数据段。因此，被调用者只具有调用者的上下文，进而以某种方式设置*自己的* DP。

解决方案是，每个数据段的前几个字包含一个表拷贝，称为数据段基表 (DSBT)，并列出构成程序的其他模块中所有其他段的基地址。每个共享库都会分配到一个唯一索引，该索引从 1 开始。索引 0 为可执行文件而保留。被调用者使用其分配到的索引，在调用者的表拷贝中查找自己的基地址，然后将该值分配给 DP。在给定可执行文件及其共享库的专用数据段内，每个 DSBT 拷贝都相同，从而使任何被调用者都可以使用任何调用者的表来查找自己的基址。

DSBT 方法具有所需的特性，即动态链接的惩罚仅局限于导出函数。对于不使用动态链接的裸机程序或不使用导出函数的可执行文件，ABI 不受影响。通过明智地使用工具链特定声明结构，明确标识外部可访问的函数（请参阅节 6.7.2），程序员可以尽可能减少开销。在确实需要调整 DP 的函数中，开销通常仅为 3 条指令。

DSBT 模型的缺点是，需要协调库索引的分配并在最大模块数量上强制达成一致，这将决定每个数据段中的表大小。

DSBT 在 .dsbt 段中由静态链接器分配，并且必须位于每个模块第一个 DP 相对段的基地址，以便 DP 指向它。加载模块时，动态链接器会将表条目初始化。

可执行文件始终使用索引 0 来访问表；库索引从 1、2 或为特定平台指定的某个其他索引开始。可以通过以下两种方式之一分配库的索引：

- 它可以在静态链接时（或等效于通过静态后链接工具）通过命令行选项或其他指令，进行静态分配。当库驻留在 ROM 中并且无法在动态加载时重定位时，必须使用此方法。
- 它可以由动态链接器动态分配。这需要在加载库的代码段时对其重定位（修补），以便更新索引，这样具有动态分配索引的库就不会被视为与位置无关。

关于每个对象的 DSBT，必须至少与分配给作为程序一部分动态加载的任何模块的最大索引一样大。动态链接器负责确保所有模块都具有足够大的 DSBT；如果不是这样，必定无法加载程序。DSBT 的大小在静态链接时（或

静态链接后工具）通过命令行选项或环境变量指定。嵌入式系统通常需要少量动态库；因此 DSBT 的典型大小为 5 或更小。

模块的动态段包含 C6000 特定标记，这些标记指定其 DSBT 表的大小及其索引（如果已分配）。节 14.3.2 中详细介绍了这些信息。

6.7.1 导出函数的进入/退出序列

以下代码序列说明了导出函数如何通过编制索引到其调用方的 DSBT 来更改 DP 以指向其数据区段。任何更改 DP 的函数都有责任在返回时恢复 DP（DP 是由被调用方保存的）。

设置 DP 的进入序列

```
func:
        MV      DP,somewhere                ;typically the stack
        LD      *+DP[$DSBT_index(func)],DP ;reloc R_C6000_DSBT_INDEX
        ; body of function
```

表达式 $DSBT_index(func) 计算当前对象的唯一库索引，并生成特殊重定位来指示这一点。将在静态链接时或动态绑定索引。

退出序列

```
        MV      somewhere, DP
        RET
```

退出序列仅恢复调用方的 DP。

如果导出函数不使用任何 DP 相对寻址，*并且*不调用任何使用 DP 相对寻址的函数，则可选择不更改 DP。

6.7.2 避免在内部函数中使用 DP 负载

只有可从其他链接单元调用的函数才需要调整 DP。只能从其静态链接单元内调用的函数不需要调整 DP，因为它们可以依靠其调用方来执行此操作。函数被外部调用的能力称为其*可见性*。（请注意，可见性也适用于对象被抢占的能力；请参阅 节 6.4。）

来自另一个链接单元的外部调用可以是直接调用（这种情况下通过名称调用函数），也可以是间接调用（这种情况下会获取函数的地址并传递给外部调用方，然后调用方通过此地址调用函数）。ELF 提供了四个可见性级别，涵盖了从其他模块进行直接和间接调用的各种可能性，如表 6-1 中所述：

表 6-1. ELF 可见性属性解读

名称	可直接调用	可间接调用	可抢占
STV_DEFAULT	是	是	是
STV_PROTECTED	是	是	否
STV_HIDDEN	否	是	否
STV_INTERNAL	否	否	否

函数的可见性由其声明以及一组编译器和平台特定约定共同决定。例如，在 Linux 模型中，除非在外部函数的声明中添加 __attribute__((visibility)) 修饰符来指示，否则外部函数具有 STV_DEFAULT 可见性；但对于裸机平台，STV_HIDDEN 或 STV_INTERNAL 的默认可见性可能更合适。

6.7.3 函数指针

一般而言，由于被调用方负责设置自己的 DP，因此无需对函数指针进行特殊处理。导出函数可安全地从定义它们的模块内部或外部间接调用。（这是 DSBT 模型相对于其他一些无 MMU 方法的主要优势。）

但是，使用函数指针时会遇到潜在陷阱。如果具有内部可见性的函数获取其地址并传递给另一个静态链接单元，然后间接调用，则很可能无法正确设置 DP，并且程序会失败。

严格来说，获取*内部*函数的地址并将其提供给另一个模块属于编程错误，因为这种操作违反了可见性声明暗示的假设。为帮助检测此类违规行为，工具链可能会选择让编译器在获取非导出函数的地址时发出警告。操作合规的用户可禁用该警告。

获取*外部*函数的地址并将其传递到另一个模块始终是合规的。为了能够按预期对在不同模块中计算的函数指针进行比较，ABI 要求表示函数地址的表达式在所有模块中的求值是唯一值。其他架构的一些 ABI 采用约定：在可执行文件中，对函数地址的引用可解析为 PLT 条目，从而允许静态解析这些引用。（由于占先，必须动态解析来自共享对象的引用）。

C6000 ABI 未采用该约定，因为它会导致通过函数指针进行跨模块调用时出现问题。如果此类指针可解析为 PLT 条目，则当间接调用到达该 PLT 条目时，DP 值可能是不同静态链接单元的 DP 值，从而使 PLT 条目不能访问 GOT。实际上，PLT 条目是一个内部函数，因此不应从模块外部间接调用。

因此，C6000 ABI 的约定是：对函数地址的引用必须解析为该函数的实际地址。这意味着对于导入对象，此类引用是无法静态解析的；它们必须在加载时由动态链接器解析。

6.7.4 中断

在没有共享库的独立应用中，DP 从不改变。假设该约定适用于整个系统，则中断服务例程能够可靠地假设 DP 指向仅有的一个 RW 区段。

如果存在动态链接，中断例程就无法假设有关 DP 的任何信息。它必须像任何其他导出函数一样为自己保存、设置和恢复 DP。

6.7.5 与非 DSBT 代码的兼容性

DSBT 模型作为 ABI 的变体提供，以支持位置独立性和共享库。许多嵌入式系统不需要这些功能，因此可以避免增加复杂性和性能开销。使用 DSBT 模型的代码与不使用该模型的代码不存在二进制兼容。目标文件中的构建属性表示它是使用 DSBT 模型构建的；链接器和加载器应防止 DSBT 代码与非 DSBT 代码混合。

6.8 动态链接的性能影响

动态链接会造成性能损失。通过 PLT 调用的导入函数会产生额外调用的开销，类似于蹦床函数。如果通过 GOT 访问函数的地址，则还存在间接访问以载入其地址的开销。

通过 DP 寻址的 near 数据不会有任何损失。对于 far 数据，DP 相对寻址需要三条指令，而与位置有关的绝对寻址需要两条指令。对于通过 GOT 寻址的对象，存在额外引用 GOT 以载入地址的开销。

符号抢占显著加剧了 GOT 损失。编译器和静态链接器必须将任何可能被抢占的符号（即共享库中定义的任何全局符号）视为导入符号。即使是本地定义的函数也必须通过 PLT 调用，从而排除内联或专门化。本地定义的变量必须通过 GOT 间接访问。这些限制适用于编译器生成的代码，因此即使符号最终未被抢占，损失通常也无法恢复。

由于抢占而造成的损失仅适用于共享库。可执行文件（不是库）中定义的符号不能被抢占。

系统采用多种技术来减轻这些影响。在某些遵循 DLL 模型的系统（Windows、Palm、Symbian）中，除非明确声明，定义的符号不被视为导出符号。

在 UNIX 系统（包括 Linux）中，所有外部符号都可能为动态链接，这意味着编译器必须为所有此类符号生成低效的 GOT 间接寻址。为了减轻这种影响，UNIX 模型采用“作为自有导入”模型，如节 15.9 所述。

工具链可以采用其他特定于供应商的方法来减轻抢占损失，例如通过选项或声明说明符来改变外部符号的默认可见性。

DSBT 模型引入了开销（因为导出函数必须保存和恢复 DP），即 3 条指令和 2 个存储器引用的成本。表本身也具有数据大小开销，它向每个可执行文件和库的数据段添加 N+1 个字，其中 N 是应用使用的任何库的最大索引。

章节 7

线程局部存储分配和寻址

多线程编程在许多使用 C6000 系列处理器的嵌入式系统中很常见。考虑到基于 C6000 CPU 的多核器件数量增加，多线程编程的应用预计将更加广泛，从而利用多个内核。此外，OpenCL 等多核编程模型也依赖底层多线程支持。

7.1 关于多线程和线程局部存储

如果线程可以使用具有静态存储持续时间且特定于线程的变量，则复杂的多线程程序在结构上可以更加清晰，开发也更加容易。也就是说，其他线程无法看到或访问此类静态存储持续时间且特定于线程的变量。考虑以下 C 代码：

```
int global_x;
foo()  {
    int local_x;
    static int static_x = 0;
    ...
}
```

global_x 和 static_x 变量为每个进程分配一次，并且所有线程均共享同一个实例。相比之下，local_x 是从堆栈分配的。由于每个线程都具有自己的堆栈，因此变量 local_x 特定于线程，而 static_x 则不是。但是，目前尚无简单的方法能够根据每个线程来定义全局/静态变量。POSIX 线程接口允许使用 pthread getspecific 和 pthread setspecific 创建特定于线程的静态存储变量。但该接口使用不方便。

为了解决此问题，我们引入了线程局部存储 (TLS)，这是一个存储类，允许程序定义具有静态存储持续时间且特定于线程的变量。TLS 变量或“线程局部变量”是一个每线程实例化一次的全局/静态变量。

用于 TLS 的存储器在整个程序运行期间都是静态分配的。每个线程都有自己的*所有*线程局部变量（甚至那些它没有声明或使用的变量）实例，这些变量由创建线程时加载的所有动态模块定义。创建线程后，其 TLS 块由底层操作系统 (OS) 线程支持库分配和初始化。如果线程完成，然后在同一程序运行中再次运行，则线程的 TLS 块将重新初始化。如果线程暂停或被其他线程阻止，然后在解除阻止后恢复执行，则 TLS 变量不会重新初始化。

访问 TLS 变量的方式取决于 OS 或 RTOS 如何为每个线程创建和管理线程局部存储。Linux 系统需要支持对多个动态库和运行时使用 dlopen() 加载的库进行 TLS 分配。此外，Linux 系统可能要求仅在访问线程本地时才缓慢地分配 TLS 存储。这需要复杂的 TLS 存储管理，并影响线程本地的访问方式。另一方面，包含 RTOS 的静态可执行文件只需管理单个 TLS 块，访问可以很简单。

在概述线程本地概念之后，本文档介绍了如何在源代码中指定线程本地，以及如何在 ELF 目标文件（节 7.4）中表示线程本地。然后，本文档介绍了如何针对 C6x Linux、静态可执行文件和裸机动态链接 TLS 模型（节 7.5）访问线程本地，以及如何解析对线程局部变量的弱引用。(节 7.6)。

C6000 TLS 机制基于行业通用约定，例如 Ulrich Drepper 编写的线程局部存储的 ELF 处理白皮书中所述的机制。

7.2 术语和概念

线程局部变量是特定于线程的变量，并具有静态存储持续时间。它们必须与全局及静态变量以类似的方式进行分配，如果它们已初始化，则在 .neardata 或 .fardata 段中分配，如果未初始化，则在 .bss 中分配。全局变量和静态变量每个进程只有一个拷贝，而线程局部变量每个线程需要一个单独的实例。

当程序为创建线程而进行调用时，线程由线程管理器创建。例如，多线程应用中的并行区域会调用操作系统线程库，以创建工作线程；这些工作线程将加入/合并到并行区域的底部。

在线程创建期间，必须分配线程局部变量的存储并将其初始化。这意味着，每个线程 TLS 存储需要有一个初始化映像以用于初始化。如果使用线程局部存储，静态链接器（静态链接单元）的输出必须包含 TLS 初始化映像。静态链接单元称为*模块*。

单个模块的 TLS 初始化映像称为一个 *TLS 映像*。在创建线程的过程中，会为每个线程分配 TLS，并使用来自 TLS 映像的数据将其初始化。对于来自单个模块的线程局部变量，每个线程分配的存储器被称为 *TLS 块*。

在静态可执行文件模型中，静态链接器会生成从起始地址加载并执行的可执行文件。RTOS 和/或线程库作为可执行文件的一部分链接进来。在本例中，只有一个模块，因此只有一个 TLS 映像和一个 TLS 块。这可以简化 TLS 访问。主线程通常在程序初始化时创建；其他线程则由线程库创建。主线程的 TLS 块应由程序加载器分配并初始化（请参阅节 14.2）。线程库负责为其创建的线程分配 TLS 并将其初始化。

在 C6x Linux 系统中，程序（进程）是通过加载多个模块来创建的：一个可执行文件和零个或多个动态库。每个模块可以有一个 TLS 映像。程序的 TLS 映像包含所有模块的 TLS 映像。这称为 *TLS 模板*。通常，可执行文件和所有依赖模块都在进程启动时加载。它们称为*初始加载的模块*。Linux 程序也可以在启动后，通过调用 dlopen() 系统函数来加载动态库。启动后加载的模块称为*经过 dlopen 处理的模块*。在创建线程过程中，会根据 TLS 模板创建 TLS 块。由所有模块的 TLS 块组成的运行时结构称为 *TLS*。

如果是裸机动态链接，默认情况下只有初始加载的模块，它们可以连续放置以形成 TLS 模板。

有关 C6x 程序加载和动态链接的更多信息，请参阅章节 14。

7.3 用户界面

多年来，C 和 C++ 经过扩展，已允许定义线程局部变量：

- Linux 系统编译器（GCC、Sun、IBM 和 Intel）支持 _thread 存储限定符，将其作为一种 C/C++ 语言扩展。但是，这不是官方语言扩展。
- Windows 编译器（MS VC++、Intel、Borland）支持 __declspec(thread) 存储属性扩展。
- 最新的 C++ 标准 C++11 (ISO/IEC 14882:2011) 引入了 thread_local 存储类说明符。
- 最新的 C 标准 C11 引入了 _Thread_local 存储类说明符。

用于支持线程局部存储的语言扩展特定于工具链，不在 ABI 的范围内。

线程局部变量可以初始化或未初始化。与未初始化的全局变量和静态变量一样，未初始化的线程局部变量也会初始化为零。线程局部变量的分配和初始化将在创建线程时发生，无论以静态方式还是动态方式。

7.4 ELF 目标文件表示

ELF 规范 (www.sco.com/developers/gabi/) 提供了有关如何在 ELF 可重定位目标文件和 ELF 模块中表示线程局部存储的详细信息。

为总结 ELF 规范的相关部分，线程局部变量在目标文件和 ELF 模块中的表示方式与静态数据类似。不同之处在于，ELF 要求在段中分配线程局部变量，这些段具有在可重定位文件中设置的 SHF_TLS 标志。此外，ELF 规范要求将段名称 .tdata 和 .tbss 分别用于初始化的和未初始化的线程局部存储。这些段具有读写权限。

在模块中，ELF 要求 TLS 区段具有 PT_TLS 区段类型。该段为只读。PT_TLS 段是 TLS 映像。

线程局部符号的符号类型为 STT_TLS。

7.5 TLS 访问模型

每个线程都有自己的*所有*线程局部变量（甚至包括它未声明或使用的变量）的实例。访问线程局部变量时，应该会访问当前线程的该线程局部变量实例。这意味着线程局部访问需要查找当前线程的 TLS，然后对定义变量的 TLS 块使用偏移以访问变量。

访问 TLS 数据有六种模式，具体取决于是否支持 DLL/DSO（独立链接）、是否支持 dlopen() 以及访问自有数据还是导入数据等因素。表 7-1 总结了 TLS 访问模型的各种特征。

表 7-1. 线程局部存储寻址模型

型号	通用动态	局部动态	初始可执行文件	局部可执行文件	静态可执行文件	裸机动态
系统有 DLL 或 DSO	是	是	是	是	否	是
可以访问另一个模块的 TLS 数据	是	否	是	否	否	是
从 DLL 或 DSO 进行 TLS 访问	是	是	是	否	否	是
从经过 dlopen 处理的模块进行 TLS 访问	是	是	否	否	否	否
TLS 初始化的执行者	加载器	加载器	加载器	加载器	链接器	加载器
支持弱引用	是	是	否	否	否	否

表 7-1. 线程局部存储寻址模型（续）

型号	通用动态	局部动态	初始可执行文件	局部可执行文件	静态可执行文件	裸机动态
用例	访问另一模块的 TLS 数据	从 DLL 或 DSO 访问自有的 TLS 数据	访问加载时加载的模块的 TLS 数据	从可执行文件访问自己的 TLS 数据	无 DLL 或 DSO	访问加载时加载的模块的 TLS 数据
部分	节 7.5.1.1	节 7.5.1.2	节 7.5.1.3	节 7.5.1.4	节 7.5.2	节 7.5.3

C6x Linux TLS 访问模型需要满足更多约束，而且可能很复杂。它需要符合已经确立的约定。另一方面，静态可执行文件访问模型很简单，只有一个 TLS 块，并且可以使用线程指针相对 (TPR) 寻址来访问任何线程局部变量。有益的做法是，先描述比较复杂的 C6x Linux TLS 模型（节 7.5.1），然后描述静态可执行文件 TLS 模型，作为一种较为简单的情况（节 7.5.2）。最后，描述裸机动态链接情况（节 7.5.3）。

文献中讨论了四种广泛使用的 TLS 访问模型。[3] [4] 它们是：

- 通用动态 TLS 访问模型（节 7.5.1.1）
- 局部动态 TLS 访问模型（节 7.5.1.2）
- 初始可执行文件 TLS 访问模型（节 7.5.1.3）
- 局部可执行文件 TLS 访问模型（节 7.5.1.4）

表 13-5 和表 13-6 中列出了用于 TLS 的重定位完整列表。后面的章节展示了这些重定位的用法。

7.5.1 C6x Linux TLS 模型

在某些动态链接模型中（包括 Linux），可以在运行时使用 dlopen() 来加载模块。dlopened 模块中的 TLS 块是动态分配的，因此不能以从 TP 的固定偏移量的方式为所有线程分配。因此，访问线程局部变量的方法是通过使用模块标识符和模块 TLS 块中线程局部变量的偏移量进行引用。

[3] Ulrich Drepper，*ELF Handling for Thread-Local Storage*（线程局部存储的 ELF 处理），http://www.uclibc.org/docs/tls.pdf，2005 年，版本 0.20

[4] Alexandre Olivia 和 Glauber de Oliveira Costa，*Speeding Up Thread-Local Storage Access in Dynamic Libraries in the ARM Platform*（加速 ARM 平台动态库中的线程局部存储访问），http://www.fsfla.org/~lxoliva/writeups/TLS/paper-lk2006.pdf，2006 年。

图 7-1 展示了 C6x Linux TLS 运行时表示法。每个线程都有一个此运行时 TLS 结构的实例。

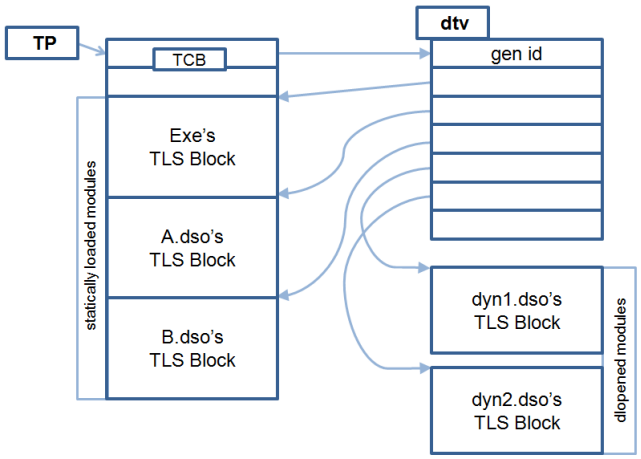


图 7-1. C6x Linux TLS 运行时表示法

对于每个线程，线程指针 (TP) 都指向线程控制块 (TCB)。可执行文件的 TLS 块如果存在，则在调整对齐后放置在 TCB 后面。来自其他非动态模块的 TLS 块随后将按照其对齐要求进行放置。静态模块的随后 TCB 和 TLS 块构成程序的静态 TLS。在创建线程的过程中，将会创建线程的静态 TLS。

TCB 为 64 位宽度。前 32 位指向动态线程矢量 (DTV)。剩下的 32 位保留。

通过 TCB 指向的 dtv 是一个 32 位元素的向量。dtv[0] 元素是生成 ID，用于在装载经过 dlopen 处理的模块时管理 dtv 的动态增长。dtv[n] 元素（其中 n != 0）是指向模块 n 的 TLS 块的 32 位指针。当加载含 TLS 数据的模块时，会为该模块分配一个模块 ID。此模块 ID 特定于进程。由多个进程共享的动态共享库在每个进程中可以具有不同的模块 ID。模块 ID 1 始终分配给可执行文件。

主线程由动态加载器创建，后续线程由线程库创建。创建主线程时，dtv 阵列只需要包含指向最初加载的模块的指针。

当线程对新模块进行 dlopen 处理时，应该为进程中的所有线程分配模块的 TLS 块。如果其他线程访问这个新模块的线程本地数据，则需要执行此操作。但是，可以推迟分配经过 dlopen 处理的模块的 TLS 块，推迟到首次访问该模块的存储器后。这可以通过将相应的 dtv[module-id] 初始化为 TLS_DTV_UNALLOCATE 来实现。__tls_get_addr() 函数可以检查 dtv[module_id] 是否为 TLS_DTV_UNALLOCATED；如果是，则它会为当前线程分配并初始化 TLS 块。

7.5.1.1 通用动态 TLS 访问模型

这是最通用的 TLS 访问模型。使用该访问模型的对象可用于构建任何 Linux 模块：可执行文件、初始加载模块和 dlopened 模块。在静态链接期间，为该模型生成的代码不能假设模块 id 或偏移是已知的。

采用这种访问模型，可在运行时加载动态模块。为实现这种可能性，线程库的线程管理架构必须提供一种方法，以便在运行时加载和卸载动态模块时，添加和移除 TLS 块。

编译器生成对 __tls_get_addr() 的调用来获取线程局部变量的地址。模块 TLS 块中的模块 id 和线程局部变量的偏移量都是作为形参来传递的。该代码从全局偏移量表 (GOT) 条目中获得模块 id 和偏移量，以确保位置独立性 (PIC) 和符号占先。

__tls_get_addr() 函数传递模块 id 和偏移量的最简单方法如下所示：

```
void * __tls_get_addr(unsigned int module_id, ptrdiff_t offset);
```

请注意，两者都是 32 位实参，GOT 条目也是 32 位条目。作为优化方案，如果 ISA 能够支持，我们可将这两个 GOT 条目加载为 64 位双字。这两个 GOT 条目必须连续分配并与 64 位边界对齐。可将该 GOT 实体视为如下结构：

```
struct TLS_descriptor
{
    unsigned int module_id;
    ptrditt_t offset;
} __attribute__ ((aligned (8)));
```

那么，__tls_get_addr() 接口变为：

```
void * __tls_get_addr(struct TLS_descriptor);
```

在该 EABI 中，大小为 64 位或更小的结构由值传递，从而在 A5:A4 寄存器对中传递 TLS 描述符。在小端字节序模式下，模块 id 在 A4 中传递，偏移量在 A5 中传递。在大端字节序模式下，按照 C6x EABI 调用约定交换寄存器。本节中的示例使用小端字节序模式。

使用该接口，线程局部访问变为以下内容（适用于 C64 及更高版本）：

```
    LDDW   *+DP($GOT_TLS(X)), A5:A4  ;reloc R_C6000_SBR_GOT_U15_D_TLS
|| CALLP  __tls_get_addr,B3          ; A4 has the address of X at return
    LDW    *A4, A4                   ; A4 has the value of X
```

重定位 R_C6000_SBR_GOT_U15_D_TLS 使得链接器为 x 的模块 id 和偏移量创建 GOT 条目，如下所示：

```
64-bit aligned address:
        GOT[n]            ;reloc R_C6000_TLSMOD  (symbol X)
        GOT[n+1]          ;reloc R_C6000_TBR_U32 (symbol X)
```

然后，链接器使用 GOT 实体的 DP 相对偏移量来解析 R_C6000_SBR_GOT_U15_D_TLS 重定位。动态加载器将 R_C6000_TLSMOD 解析为在其中定义 x 的模块的模块 id。它将 R_C6000_TBR_U32 解析为模块 TLS 块中 x 的偏移量。

C6x ISA 当前无直接加载 64 位 TLS 描述符的指令。但是，我们使用 64 位描述符来定义 __tls_get_addr() 接口，以期未来 ISA 具有此类支持功能。

```
void * __tls_get_addr(struct TLS_descriptor);
```

链接器须连续分配线程局部变量的模块 id 和偏移量的 GOT 条目，并在发现 R_C6000_SBR_GOT_U15_D_TLS 重定位时将第一个条目与 64 位边界对齐。

由于不支持 DP 相对 64 位加载，因此可在当前 ISA 上使用以下序列：

```
    LDW    *+DP($GOT_TLSMOD(X)), A5   ;reloc R_C6000_SBR_GOT_U15_W_TLSMOD
    LDW    *+DP($GOT_TBR(X)), A4      ;reloc R_C6000_SBR_GOT_U15_W_TBR
|| CALLP  __tls_get_addr,B3          ; A4 has the address of X at return
    LDW    *A4, A4                    ; A4 has the value of X
```

重定位 R_C6000_SBR_GOT_U15_W_TLSMOD 和 R_C6000_SBR_GOT_U15_W_TBR 使得链接器为 x 的模块 id 和偏移量分别创建 GOT 条目。该访问模式不要求这些 GOT 条目是连续且 64 位对齐的。如果链接器也未发现同一符号的 DW_TLS 重定位，则可自由分别定义模块 id 和偏移量 GOT 条目，无需 64 位对齐。但是，如果除同一符号的 TLSMOD/TBR 重定位之外还发现了 DW_TLS，则必须定义 64 位对齐的连续 GOT 条目，并将其重新用于 TLSMOD/TBR 重定位。

如果必须使用 far DP 寻址来寻址 GOT，则通用动态寻址变为：

```
    MVKL $DPR_GOT_TLSMOD(X),  A5      ;reloc R_C6000_SBR_GOT_L16_W_TLSMOD
    MVKH $DPR_GOT_TLSMOD(X),  A5      ;reloc R_C6000_SBR_GOT_H16_W_TLSMOD
```

```
        ADD  DP, A5, A5
        LDW  *A5, A5
        MVKL $DPR_GOT_TPR(X),  A4          ;reloc R_C6000_SBR_GOT_L16_W_TBR
        MVKH $DPR_GOT_TPR(X),  A4          ;reloc R_C6000_SBR_GOT_H16_W_TBR
        ADD  DP, A4, A4
        LDW  *A4, A4
     || CALLP  __tls_get_addr,B3           ; A4 has the address of X at return
        LDW    *A4, A4                     ; A4 has the value of X
```

__tls_get_addr() 可计算线程局部地址，如下所示：

```
    void * __tls_get_addr(struct TLS_descriptor desc)
    {
        void *TP  = __c6xabi_get_tp();
        int  *dtv = (int*)(((int*) TP)[0]);
        char *tls = (char *)dtv[desc.module_id];
        return tls + desc.offset;
    }
```

7.5.1.2 局部动态 TLS 访问模型

该访问模型是通用动态模型的优化，用于访问模块自带数据。如果编译器知道自己正在访问模块自带的线程局部存储，则可使用该访问模型。如果线程局部变量是在访问自己的同一模块中定义的，则 TLS 偏移量在静态链接时是已知的。但是，模块 id 在静态链接时是未知的。

使用为零的偏移量实参来调用 __tls_get_addr()，将返回该模块 TLS 块的基址。该基址可用于访问属于该模块的所有线程局部数据。

在编译时，使用符号绑定和可见性来标识线程自带数据。具有静态范围或隐藏/受保护可见性的符号是自带数据。在该模型中，可按如下方式访问线程局部 x：

```
        LDW  *+DP($GOT_TLSMOD(x)), A4      ; reloc R_C6000_SBR_GOT_U15_W_TLSMOD
        MVK  $TBR_word(x), A5              ; reloc R_C6000_TBR_U15_W
     || CALLP    __tls_get_addr,B3    ; A4 has the address of x at return
```

如前文所述，可以一次性获取自带 TLS 基址，然后重复用于访问其他自带线程局部变量，如下所示：

```
        LDW  *+DP($GOT_TLSMOD()), A4     ; reloc R_C6000_SBR_GOT_U15_W_TLSMOD w/ Symbol=0
        MVK  0x0, A5                     ;
     || CALLP    __tls_get_addr,B3 ; A4 has the module's own TLS base
        MVK  $TBR_byte(x), A5            ; reloc R_C6000_TBR_U15_B; Get x's scaled TLS offset
        LDB  *A4[A5], A6                 ; A6 has the value of thread-local char x
        MVK  $TBR_hword(y), A5           ; reloc R_C6000_TBR_U15_H; Get y's scaled TLS offset
        LDH  *A4[A5], A6                 ; A6 has the value of thread-local short y

        MVK  $TBR_word(z), A5            ; reloc R_C6000_TBR_U15_W; Get z's scaled TLS offset
        LDW  *A4[A5], A6                 ; A6 has the value of thread-local int z
        MVK  $TBR_dword(l), A5           ; reloc R_C6000_TBR_U15_D; Get l's scaled TLS offset
        LDDW *A4[A5], A7:A6              ; A7:A6 has the value of thread-local long long l
```

符号为零时，重定位 R_C6000_SBR_GOT_U15_W_TLSMOD 解析为模块的自带模块 id。TBR_U15 重定位对来自模块 TLS 基址的 15 位无符号偏移量进行编码，用于 near TB (TLS 块基址) 寻址。重定位根据访问宽度进行缩放。前面的寻址可访问大小为 32KB 的 TLS 块。该规范将每个模块 TLS 块的大小限制为 32KB，预计这一限制足以满足大多数用例。因此，未定义 far TB 相对地址。可定义 far TBR 寻址，但这最多会使用 8 个新的重定位，最好是保留 ELF 允许的有限重定位数量 (256)。

静态链接器使用纯静态重定位来解析所有 TBR 重定位。也就是说，这些重定位不能位于动态重定位表中。

7.5.1.3 初始可执行文件 TLS 访问模型

用于构建初始加载模块的对象可使用该访问模型。使用该访问模型的模块不能是 dlopened 模块。

由于模块始终是初始加载的，并且动态加载器能够在可执行文件的 TLS 块之后连续分配来自初始模块的 TLS 块，因此，在动态链接时，线程指针的偏移量是已知的。可使用 *(TP + offset) 来访问线程局部变量，其中，从 GOT

加载偏移量，以确保 PIC 和符号占先。使用该类寻址构建的模块不能是 dlopened 模块。此类模块标有动态标志 DF_STATIC_TLS，动态加载器会拒绝 dlopen 标有 DF_STATIC_TLS 的模块。

7.5.1.3.1 线程指针

用于初始可执行文件模型的寻址需要一种方法来获取当前线程的线程指针。新的 c6xabi 函数 __c6xabi_get_tp() 会返回当前线程的线程指针值。此函数不修改返回寄存器 A4 以外的任何寄存器。此函数可通过 PLT 调用，因此调用者应假定 B30 和 B31 寄存器会被对此函数的调用修改。此函数具有以下签名：

```
void * __c6xabi_get_tp(void);
```

线程库负责提供此函数的定义。

7.5.1.3.2 初始可执行文件 TLS 寻址

在初始可执行文件模型中，线程局部变量的访问方式如下：

```
callp  __c6xabi_get_tp()          ;Returns TP in A4; Can be CSEed
LDW  *+DP($GOT_TPR_byte(x)), A5   ;reloc R_C6000_SBR_GOT_U15_W_TPR_B
LDB    *A4[A5], B4                ;
LDW  *+DP($GOT_TPR_hword(x)), A5  ;reloc R_C6000_SBR_GOT_U15_W_TPR_H
LDH    *A4[A5], B4                ;
LDW  *+DP($GOT_TPR_word(x)), A5   ;reloc R_C6000_SBR_GOT_U15_W_TPR_W
LDW    *A4[A5], B4                ;
LDW  *+DP($GOT_TPR_dword(x)), A5  ;reloc R_C6000_SBR_GOT_U15_W_TPR_D
LDDW   *A4[A5], B4                ;
```

重定位 R_C6000_SBR_GOT_U15_W_TPR_[B|H|W] 使得链接器为 x 的 TPR 偏移量创建一个 GOT 条目：

```
GOT[m]          ;reloc R_C6000_TPR_U32_B  (symbol x)
GOT[n]          ;reloc R_C6000_TPR_U32_H  (symbol y)
GOT[o]          ;reloc R_C6000_TPR_U32_W  (symbol z)
GOT[p]          ;reloc R_C6000_TPR_U32_D  (symbol z)
```

_TPR_U32_[B|H|W|DW] 重定位由动态加载器以线程指针的偏移量 x 来解析。这些重定位根据访问宽度进行缩放。

如果必须使用 far DP 寻址来访问 GOT，则序列如下：

```
        callp  __c6xabi_get_tp()          ;Returns TP in A4; Can be CSEed
        MVKL   $DPR_GOT_TPR_byte(x), A5    ;reloc R_C6000_SBR_GOT_L16_W_TPR_B
        MVKH   $DPR_GOT_TPR_byte(x), A5    ;reloc R_C6000_SBR_GOT_H16_W_TPR_B
        ADD    DP, A5, A5
        LDW    *A5, A5
        LDB    *A4[A5], A6
        MVKL   $DPR_GOT_TPR_hword(x), A5   ;reloc R_C6000_SBR_GOT_L16_W_TPR_H
        MVKH   $DPR_GOT_TPR_hword(x), A5   ;reloc R_C6000_SBR_GOT_H16_W_TPR_H
        ADD    DP, A5, A5
        LDW    *A5, A5
        LDH    *A4[A5], A6
        MVKL   $DPR_GOT_TPR_word(x), A5    ;reloc R_C6000_SBR_GOT_L16_W_TPR_W
        MVKH   $DPR_GOT_TPR_word(x), A5    ;reloc R_C6000_SBR_GOT_H16_W_TPR_W
        ADD    DP, A5, A5
        LDW    *A5, A5
        LDW    *A4[A5], A6
        MVKL   $DPR_GOT_TPR_dword(x), A5    ;reloc R_C6000_SBR_GOT_L16_W_TPR_D
        MVKH   $DPR_GOT_TPR_dword(x), A5    ;reloc R_C6000_SBR_GOT_H16_W_TPR_D
        ADD    DP, A5, A5
        LDW    *A5, A5
        LDDW    *A4[A5], A6
```

7.5.1.4 局部可执行文件 TLS 访问模型

这是初始可执行文件模型的优化。创建程序的初始 TLS 映像（通常称为静态 TLS 映像）时，TLS 块始终放置在线程指针的已知偏移量处。通常情况下，这是线程控制块 (TCB) 加上 TLS 块基址偏移量。因此，可执行文件的自带线程局部变量具有线程指针相对偏移量，该偏移量是静态链接时常量。在这种情况下，可使用内联常量偏移量来访问线程局部变量；无需 GOT 条目。使用该访问模型的对象不能用于构建动态库。

```
        CALLP  __c6xabi_get_tp()   ; Returns TP in A4. Can be CSEed.
        MVK    $TPR_byte(x), A5    ; reloc R_C6000_TPR_U15_B
        LDB    *A4[A5], A4         ; A4 contains the value of thread-local char x
        MVK    $TPR_hword(y), A5   ; reloc R_C6000_TPR_U15_H
        LDH    *A4[A5], A4         ; A4 contains the value of thread-local short y
        MVK    $TPR_word(z), A5    ; reloc R_C6000_TPR_U15_W
        LDW    *A4[A5], A4         ; A4 contains the value of thread-local int z
        MVK    $TPR_dword(l), A5   ; reloc R_C6000_TPR_U15_D
        LDDW   *A4[A5], A7:A6      ; A7:A6 contains the value of thread-local long long l
```

TPR_U15 重定位对 15 位无符号 TPR 偏移量（TP 指向地址的偏移量）进行编码，以便实现 near TPR 寻址。重定位根据访问宽度进行缩放。前面的寻址可访问大小为 32KB 的 TLS 块。本规范将总静态 TLS 的大小限制为 32KB，因为预计这一限制足以满足大多数用例。因此，未定义 far TPR 地址。可定义 far TBR 寻址，但这样做最多会使用 8 个新的重定位，最好是保留 ELF 允许的有限重定位数量 (256)。

7.5.2 静态可执行文件 TLS 模型

静态可执行文件 TLS 模型可作为一个实现质量 (QOI) 项，由 C6x EABI 一致性编译器提供支持。无需满足 C6x EABI 合规性要求。

对于静态可执行文件，只有一个 TLS 块，并且每个线程局部变量的 TLS 偏移在静态链接时已知。对线程局部变量的访问是 *(TLS base + offset)。

图 7-2 显示了 TLS 的运行时布局。TP 是指向当前线程的 TLS 块的线程指针。x 的偏移在静态链接期间已知。

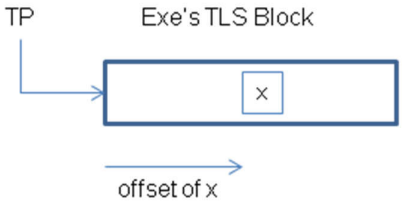


图 7-2. 静态可执行文件 TLS 运行时表示

7.5.2.1 静态可执行文件寻址

就静态可执行文件 TLS 模型而言，线程局部访问代码与 Linux 本地可执行文件模型（节 7.5.1.4）相同。至于静态可执行文件，不存在线程控制块 (TCB)，因此 TPR 偏移与 TLS 块基本相对地址相同。

理想情况下，我们可以为该情况生成 TBR 寻址。但是，编译器选项可用来使用裸机动态链接模型进行构建，这需要 TCB。因此，我们为静态可执行文件模型生成 TPR 寻址，如下所示：

```
        CALLP   __c6xabi_get_tp()    ; Returns TP in A4. Can be CSEed.
        MVK     $TPR_byte(x), B4     ; reloc R_C6000_TPR_U15_B
        LDB     *A4[B4], A4          ; A4 contains the value of thread-local char x
        MVK     $TPR_hword(y), B4    ; reloc R_C6000_TPR_U15_H
        LDH     *A4[B4], A4          ; A4 contains the value of thread-local short y
        MVK     $TPR_word(z), B4     ; reloc R_C6000_TPR_U15_W
        LDW     *A4[B4], A4          ; A4 contains the value of thread-local int z
        MVK     $TPR_dword(l), B4    ; reloc R_C6000_TPR_U15_D
        LDDW     *A4[B4], A7:A6      ; A4 contains the value of thread-local int l
```

TPR 重定位由静态链接器使用可执行文件 TLS 块中变量的偏移进行解析。如果 TLS 块预计大于 32KB，则可以使用 far TPR 重定位。

7.5.2.2 静态可执行文件 TLS 运行时架构

在动态链接系统中，动态加载器创建主线程，而线程库创建附加线程。在创建主线程的过程中，动态加载器会分配主线程的 TLS 并将其初始化。此外，动态加载器可以使用段类型轻松找到 TLS 初始化映像。

至于静态可执行文件，没有动态加载器来扮演这些角色。静态链接模型应支持以下要求：

- 在调用 main () 或 init_array 中的任何用户代码之前，分配主线程的 TLS 并将其初始化。
- 在主线程执行期间，__c6xabi_get_tp() 应返回指向主线程 TLS 的指针。即使没有线程库，也必须支持此函数。
- 线程库应该能够访问 TLS 初始化映像，以便它能够将其创建的线程的 TLS 块初始化。

节 7.5.2.3.1 至节 7.5.2.5 包含工具链特定的信息。其中包括一些提到 .TI.tls_init 和 .TI.tls 段、__TI_tls_init_table 复制表、__TI_TLS_MAIN_THREAD_BASE 和 __TI_TLS_BLOCK_SIZE 符号以及 __TI_tls_init() 函数的内容，作为示例来演示工具链如何实现 TLS 模型。

7.5.2.3 静态可执行文件 TLS 分配

为了在静态可执行文件模型中支持 TLS，需要分配三个存储器区域：初始化映像、主线程的 TLS 块，以及使线程库可在其中为其创建的线程分配 TLS 块的 TLS 区域。

7.5.2.3.1 TLS 初始化映像分配

TLS 初始化映像在输出段 .TI.tls_init 中创建。此段为只读。用户可以按如下所示为此输出段指定分配：

```
        .TI.tls_init > ROM
```

如果不指定分配，则会使用 .cinit 分配来分配此输出段。如果未为 .cinit 指定分配，则使用默认分配。用户无法为此段指定段说明符。

.TI.tls_init 输出段通过组合以下链接器创建的组件来形成：

- **tdata.load** — 压缩的 TLS 初始化段
- **tbss.load** — 通过 Zero-init 段将未初始化段进行零初始化
- **__TI_tls_init_table** — 通过复制表将 TLS 块初始化。此复制表有两个复制记录，这些初始化段各一个。

7.5.2.3.2 主线程的 TLS 分配

用户可以使用以下命令来指定主线程 TLS 块的分配：

```
        .TI.tls > RAM
```

此未初始化的输出段在引导时使用 _TI_tls_init_table 复制表进行初始化。用户无法为此段指定段说明符。

如果未为此段指定分配，则将使用 .fardata 输出段的分配来分配此段。如果未为 .fardata 指定分配，则将使用 .far 分配。否则，将使用默认分配。

链接器将符号 _TI_TLS_MAIN_THREAD_BASE 定义为指向 .TI.tls 输出段的开头。

7.5.2.3.3 线程库的 TLS 区域分配

分配供线程库使用的 TLS 区域是特定于库的操作。规范中未规定具体的实现方法。一种分配 TLS 区域的可能方式如下：

```
        .tls_region { . += 0x2000; } START(TLS_REGION_START) > RAM
```

线程库可以使用符号 TLS_REGION_START 来定位 TLS 区域。用户可能希望为 N 个线程分配 TLS 块，而且知道 TLS 块的大小很有用。用户可以采取以下操作：

```
        .tls_region { . += MAX_THREADS * __TI_TLS_BLOCK_SIZE; } > RAM
```

静态链接器将定义符号 __TI_TLS_BLOCK_SIZE，并将其设置为 TLS 块的大小。

7.5.2.4 静态可执行文件 TLS 初始化

为了在静态可执行文件模型中支持 TLS，需要将两个存储器区域初始化：主线程的 TLS 块，以及使线程库可在其中为其创建的线程分配 TLS 块的 TLS 区域。

7.5.2.4.1 主线程的 TLS 初始化

在启动过程中，启动代码调用运行时支持 (RTS) 函数 __TI_tls_init(NULL) 来初始化主线程的 TLS 块。如果传递了 NULL 实参，RTS 函数将初始化主线程的 TLS。

7.5.2.4.2 线程库进行 TLS 初始化

一旦线程库为给定线程创建 TLS 块，就必须将这些块初始化。静态可执行文件 TLS 模型为此定义了一个新的 RTS 函数：

```
    __TI_tls_init(void * dest_addr);
```

线程库必须将要初始化的 TLS 块的地址传递给该函数。

此 RTS 函数使用复制表执行初始化。但是，关于该函数如何将 TLS 块初始化，基于静态链接器与该 RTS 函数之间的接口，该接口未来可能会发生变化。因此，线程库必须仅使用此 RTS 函数作为接口来初始化 TLS 块。

7.5.2.5 线程指针

在静态可执行文件 TLS 模型中，需要调用函数 __c6xabi_get_tp() 以获取当前线程的线程指针值。如果使用了线程库，则由它负责提供此函数。

线程库知道它所创建线程的 TLS 块的地址。但是，主线程不由线程库创建，因此线程库需要一种标准方法来查找主线程的 TLS 块的地址。如前所述，静态链接器会定义符号 __TI_TLS MAIN_THREAD_BASE，以用于此目的。

TI RTS 为 __c6xabi_get_tp() 函数提供以下定义：

```
extern __attribute__((weak)) far const void * __TI_TLS_MAIN_THREAD_Base;
__attribute__((weak)) void * __c6xabi_get_tp(void)
{
    return &__TI_TLS_MAIN_THREAD_Base;
}
```

该函数被定义为“弱”函数，因此将使用线程库中的强定义（如果存在）。

我们来考虑一种不太可能的情况，即用户声明线程局部变量，但不包含线程库。显然，它们无法创建任何新线程。但是，主线程应该会起作用，并且主线程的线程局部变量应该可以访问。在这类情况下，前面提到的 RTS 函数会被链接进来，并提供对主线程的 TLS 的访问。

7.5.3 裸机动态链接 TLS 模型

裸机动态链接仅涉及最初加载的模块。目前，裸机动态链接不支持经过 dlopen 处理的模块。为静态可执行文件编译的对象可用于创建裸机动态可执行文件或库。

7.5.3.1 用于裸机动态链接的默认 TLS 寻址

TLS 的默认代码生成应适用于静态可执行文件和裸机动态链接。对于静态可执行文件，请使用 TPR 寻址生成以下寻址：

```
        CALLP   __c6xabi_get_tp()   ; Returns TP in A4. Can be CSEed.
        MVK     $TPR_byte(x), B4    ; reloc R_C6000_TPR_U15_B
        LDB     *A4[B4], A4         ; A4 contains the value of thread-local char x
```

默认情况下为裸机动态链接生成的代码可假设所有模块都是初始加载的。这意味着线程局部变量的偏移量是动态链接时常量，如 图 7-3 所示。因此，可使用 TPR 寻址。唯一的区别是，在裸机动态链接的情况下，需要 64 位 TCB 以使代码兼容今后的任何 dlopen() 支持。对于静态可执行文件，不存在 TCB。TPR 寻址仍可用于这两种模型。对于静态可执行文件，静态链接器将使用大小为零的 TCB，对于裸机动态链接，将使用大小为 64 位的 TCB。

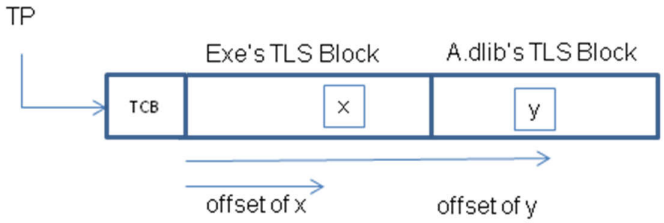


图 7-3. 裸机默认 TLS 运行时表示

如上所述，最初加载的模块是连续放置的，可执行文件的 TLS 块位于 TCB 之后。在这种情况下，可使用来自 TP 的静态链接时间常量偏移量来访问可执行文件中的变量。可使用来自 TP 的动态链接时间常量偏移量来访问动态库中定义的变量。

生成该寻址后，模块标记为 DF_STATIC_TLS。

构建动态可执行文件时，静态链接器将可执行文件（自带数据）中定义符号的 TPR 重定位解析为 TP 偏移量。如果导入符号，则会复制重定位到动态重定位表，以便动态加载器进行解析。构建动态库时，会将 TPR 重定位复制到动态重定位表中。

裸机中的线程局部访问会导致代码区段中的动态重定位。这意味着得到的模块不是真正的 PIC（与位置无关的代码）。TI 编译器支持具有 --gen_pic 选项的裸机 PIC。使用该选项时，应从 GOT 条目访问 TPR 偏移量，以便生成与位置无关的代码。

7.5.3.2 TLS 块创建

对于裸机动态链接系统，动态加载器负责创建主线程的 TLS 块。加载 ELF 文件时，动态加载器应加载 PT_TLS 段，并应为线程库提供一种访问 PT_TLS 初始化映像的方法，以便线程库可以使用它来为其创建的线程初始化 TLS 块。在构建动态可执行文件/库时，静态链接器会根据 ELF 要求生成 PT_TLS 段。

每个动态模块（可执行文件、共享对象或动态库）会获取自己的 TLS 块。PT_TLS 段包含在给定模块中定义的 TLS 对象的初始值。

7.6 线程局部符号解析和弱引用

线程局部引用只能通过线程局部定义进行解析。链接器应强制执行此要求。此外，线程局部定义和普通全局定义同名是一个错误。

线程局部变量可以定义为或声明为弱变量。弱线程局部定义意味着它可以被强定义（如果可用）覆盖。如果找不到强定义，则使用弱定义。无需特别注意支持线程局部弱定义。

如果找不到定义，弱线程局部符号引用会解析为零地址。这要求在每个 TLS 寻址模型中进行特殊处理。

7.6.1 通用和局部动态 TLS 弱引用寻址

在通用和局部动态 TLS 模型中，都会调用函数 __tls_get_addr() 来获取线程局部变量的地址。通用和本地动态 TLS 模型中的模块 id 都是从 GOT 获得的。偏移量是在通用动态模型中从 GOT 获得的，并在局部动态模型中作为静态链接时常量。在弱未定义引用的情况下，没有线程局部定义来解析弱引用。由于没有定义，模块 id 和 TBR 偏移量都解析为零。

对于弱线程局部引用，为访问引用而生成的代码没有变化。如果线程局部引用是弱引用且未定义，则 R_C6000_TLSMOD 重定位和所有 R_C6000_TBR 重定位都解析为零。

模块 id 和偏移量为零时，__tls_get_addr() 函数返回零。这样可确保将未定义的弱引用地址解析为零。

7.6.2 初始和局部可执行文件 TLS 弱引用寻址

线程指针相对寻址不能用于弱引用，因为如果不定义符号，便无法生成零地址。因此，初始可执行文件访问模型必须使用通用动态寻址进行弱引用。同样，局部可执行文件访问模型必须使用局部动态寻址进行弱引用。

7.6.3 静态可执行文件和裸机动态 TLS 模型弱引用

线程指针相对寻址不能用于弱引用，因为如果不定义符号，便无法生成零地址。因此，在静态可执行文件和裸机动态链接访问模型中，必须将局部动态形式用于弱引用。

在静态和裸机动态链接中，会为弱引用生成以下寻址：

```
        MVK   $TPR_S16(x), A5              ; reloc R_C6000_TPR_S16
          ||   CALLP    __c6xabi_get_addr,B3 ; A4 has the address of x at return
```

C6x eabi 函数 __c6xabi_get_addr() 具有以下签名：

```
    void * __c6xabi_get_addr(ptrdiff_t TPR_offst);
```

此函数接受 32 位 TPR 偏移，会返回线程局部变量的地址。TPR 偏移的特殊值 -1 用于指示弱未定义引用。这种情况下返回零。

静态链接器和动态链接器会将 TPR_S16 重定位解析为 -1，表示弱未定义引用。

章节 8

辅助函数 API

若要使使用一个工具链构建的目标文件能够与另一个工具链构建的运行时支持 (RTS) 库链接，必须指定它们之间的 API。接口包含两部分。第一部分指定函数，编译器依赖该函数来实现指令集不直接支持的语言方面。这些函数称为*辅助函数*，并记录在本节中。第二部分涉及源语言库标准的编译时方面的标准化，例如 C、C99 或 C++ 标准库，各节将进行介绍。

8.1 浮点行为

浮点行为因器件和工具链而异，因此难以标准化。ABI 旨在提供符合 C、C99 和 C++ 标准的基础。其中，在浮点方面，C99 是指定得最好的。C99 标准的附录 F 根据 IEEE 浮点标准（ISO IEC 60559:1989，之前指定为 ANSI/IEEE 754−1985）定义了 C 语言行为的浮点行为。

C6000 ABI 指定：本节中操作浮点值的辅助函数必须符合 C99 标准附录 F 指定的行为。

C99 允许通过 <fenv.h> 标头文件定制和访问浮点行为环境。为标准化辅助函数的行为，ABI 指定其按照基本默认环境进行操作，并具有以下属性：

- 舍入模式为舍入至最接近的值。不支持动态舍入精度模式。
- 不支持浮点异常。
- 表示信令 NaN 的输入行为类似于安静 NaN。
- 辅助函数仅支持 FENV_ACCESS *off* 状态下的行为。也就是说，假设程序在不间断模式下执行，并且假设不访问浮点环境。

工具链使用自带库时可自由实现更完整的浮点支持。调用工具链特定浮点支持的用户可能需要使用该工具链的库（除了符合 ABI 的辅助函数库）进行链接。

8.2 C 辅助函数 API

编译器生成对辅助函数的调用以执行编译器需要支持但架构不直接支持的运算，例如在缺少专用硬件的设备上执行浮点运算。这些辅助函数必须在符合 ABI 的任何工具链的 RTS 库中实现。

辅助函数使用前缀 _ _C6000 命名。具有此前缀的任何标识符都将保留供 ABI 使用。此外，需要 _ _tls_get_addr() 辅助函数来支持对线程局部存储的动态链接访问。

辅助函数遵守标准调用约定，节 8.3 中指定的约定除外。

下表使用 C 表示法和语法指定辅助函数。表中的类型对应于节 2.1 中指定的通用数据类型。

表 8-1 中的函数根据 C 语言的转换规则和节 8.1 指定的浮点行为将浮点值转换为 int 值。

表 8-1. C6000 浮点型至 int 转换

签名	说明
int32 _ _**C6000_fixdi**(float64 x);	将 float64 转换为 int32
int40 _ _**C6000_fixdli**(float64 x);	将 float64 转换为 int40
int64 _ _**C6000_fixdlli**(float64 x);	将 float64 转换为 int64
uint32 _ _**C6000_fixdu**(float64 x);	将 float64 转换为 uint32
uint40 _ _**C6000_fixdul**(float64 x);	将 float64 转换为 uint40
uint64 _ _**C6000_fixdull**(float64 x);	将 float64 转换为 uint64
int32 _ _**C6000_fixfi**(float32 x);	将 float32 转换为 int32
int40 _ _**C6000_fixfli**(float32 x);	将 float32 转换为 int40
int64 _ _**C6000_fixflli**(float32 x);	将 float32 转换为 int64
uint32 _ _**C6000_fixfu**(float32 x);	将 float32 转换为 uint32
uint40 _ _**C6000_fixful**(float32 x);	将单精度浮点型转换为 uint40
uint64 _ _**C6000_fixfull**(float32 x);	将单精度浮点型转换为 uint64

表 8-2 中的函数根据 C 语言的转换规则和节 8.1 指定的浮点行为将 int 值转换为浮点型值。

表 8-2. C6000 int 到浮点型转换

签名	说明
float64 _ _**C6000_fltid**(int32 x);	将 int32 转换为双精度浮点型
float64 _ _**C6000_fltlid**(int40 x);	将 int40 转换为双精度浮点型
float64 _ _**C6000_fltllid**(int64 x);	将 int64 转换为双精度浮点型
float64 _ _**C6000_fltud**(uint32 x);	将 uint32 转换为双精度浮点型
float64 _ _**C6000_fltuld**(uint40 x);	将 uint40 转换为双精度浮点型
float64 _ _**C6000_fltulld**(uint64 x);	将 uint64 转换为双精度浮点型
float32 _ _**C6000_fltif**(int32 x);	将 int32 转换为单精度浮点型
float32 _ _**C6000_fltlif**(int40 x);	将 int40 转换为单精度浮点型
float32 _ _**C6000_fltllif**(int64 x);	将 int64 转换为单精度浮点型
float32 _ _**C6000_fltuf**(uint32 x);	将 uint32 转换为单精度浮点型
float32 _ _**C6000_fltulf**(uint40 x);	将 uint40 转换为单精度浮点型
float32 _ _**C6000_fltullf**(uint64 x);	将 uint64 转换为单精度浮点型

表 8-3 中的函数根据 C 语言的转换规则和节 8.1 指定的浮点行为将浮点型值从一种格式转换为另一种格式。

表 8-3. C6000 浮点型格式转换

签名	说明
float32 _ _**C6000_cvtdf**(float64 x);	将双精度浮点型转换为单精度浮点型
float64 _ _**C6000_cvtfd**(float32 x);	将单精度浮点型转换为双精度浮点型

表 8-4 中的函数根据 C 语言的语义和节 8.1 指定的浮点行为执行浮点运算。

表 8-4. C6000 浮点算术

签名	说明
float64 _ _**C6000_absd**(float64 x);	返回双精度浮点数的绝对值
float32 _ _**C6000_absf**(float32 x);	返回单精度浮点数的绝对值
float64 _ _**C6000_addd**(float64 x, float64 y);	将两个双精度浮点数相加 (x+y)
float32 _ _**C6000_addf**(float32 x, float32 y);	将两个单精度浮点数相加 (x+y)
float64 _ _**C6000_divd**(float64 x, float64 y);	将两个双精度浮点数相除 (x/y)
float32 _ _**C6000_divf**(float32 x, float32 y);	将两个单精度浮点数相除 (x/y)
float64 _ _**C6000_mpyd**(float64 x, float64 y);	将两个双精度浮点数相乘 (x*y)
float32 _ _**C6000_mpyf**(float32 x, float32 y);	将两个单精度浮点数相乘 (x*y)
float64 _ _**C6000_negd**(float64 x);	返回取反的双精度浮点数 (-x)
float32 _ _**C6000_negf**(float32 x);	返回取反的单精度浮点数 (-x)
float64 _ _**C6000_subd**(float64 x, float64 y);	将两个双精度浮点数相减 (x-y)
float32 _ _**C6000_subf**(float32 x, float32 y);	将两个单精度浮点数相减 (x-y)
int64 _ _**C6000_trunc**(float64 x);	将双精度浮点数趋零截尾
int32 _ _**C6000_truncf**(float32 x);	将单精度浮点数趋零截尾

表 8-5 中的函数根据 C 语言的语义和节 8.1 指定的浮点行为执行浮点比较。

如果 x 小于 y，则 C6000_cmp* 函数返回一个小于 0 的整数；如果 x 等于 y，则返回 0；如果 x 大于 y，则返回一个大于 0 的整数。如果任一操作数为 NaN，则结果未定义。

显式比较函数与无序 (NaN) 操作数可正常运行。也就是说，如果比较结果为 true，则它们返回非零值，即使其中一个操作数为 NaN 也不例外，否则返回 0。

表 8-5. 浮点数比较

签名	说明
int32 _ _**C6000_cmpd**(float64 x, float64 y);	双精度比较
int32 _ _**C6000_cmpf**(float32 x, float32 y);	单精度比较
int32 _ _**C6000_unordd**(float64 x, float64 y);	针对无序操作数的双精度检查
int32 _ _**C6000_unordf**(float32 x, float32 y);	针对无序操作数的单精度检查
int32 _ _**C6000_eqd**(float64 x, float64 y);	双精度比较：x == y
int32 _ _**C6000_eqf**(float32 x, float32 y);	单精度比较：x == y
int32 _ _**C6000_neqd**(float64 x, float64 y);	双精度比较：x != y
int32 _ _**C6000_neqf**(float32 x, float32 y);	单精度比较：x != y
int32 _ _**C6000_ltd**(float64 x, float64 y);	双精度比较：x < y
int32 _ _**C6000_ltf**(float32 x, float32 y);	单精度比较：x < y
int32 _ _**C6000_gtd**(float64 x, float64 y);	双精度比较：x > y
int32 _ _**C6000_gtf**(float32 x, float32 y);	单精度比较：x > y
int32 _ _**C6000_led**(float64 x, float64 y);	双精度比较：x <= y
int32 _ _**C6000_lef**(float32 x, float32 y);	单精度比较：x <= y
int32 _ _**C6000_ged**(float64 x, float64 y);	双精度比较：x >= y
int32 _ _**C6000_gef**(float32 x, float32 y);	单精度比较：x >= y

表 8-6 中的整数除法和余数函数根据 C 语言的语义进行运算。_ _C6000_divremi 和 _ _C6000_divremu 函数计算商 (x/y) 和余数 (x%y)，在 A4 中返回商，在 A5 中返回余数。_ _C6000_divremll 和 _ _C6000_divremull 函数计算 64 位整数的商 (x/y) 和余数 (x%y)，在 A5:A4 中返回商，在 B5:B4 中返回余数。

表 8-6. C6000 整数除法和余数

签名	说明
int32 _ _**C6000_divi**(int32 x, int32 y);	32 位 signed int 除法 (x/y)
int40 _ _**C6000_divli**(int40 x, int40 y);	40 位 signed int 除法 (x/y)
int64 _ _**C6000_divlli**(int64 x, int64 y);	64 位 signed int 除法 (x/y)
uint32 _ _**C6000_divu**(uint32 x, uint32 y);	32 位 unsigned int 除法 (x/y)
uint40 _ _**C6000_divlu**(uint40 x, uint40 y);	40 位 unsigned int 除法 (x/y)
uint64 _ _**C6000_divllu**(uint64 x, uint64 y);	64 位 unsigned int 除法 (x/y)
int32 _ _**C6000_remi**(int32 x, int32 y);	32 位 signed int 取模 (x%y)
int40 _ _**C6000_remli**(int40 x, int40 y);	40 位 signed int 取模 (x%y)
int64 _ _**C6000_remlli**(int64x. int64 y);	64 位 signed int 取模 (x%y)
uint32 _ _**C6000_remu**(uint32 x, uint32 y);	32 位 unsigned int 取模 (x%y)
uint40 _ _**C6000_remul**(uint40, uint40);	40 位 unsigned int 取模 (x%y)
uint64 _ _**C6000_remull**(uint64, uint64);	64 位 unsigned int 取模 (x%y)
_ _**C6000_divremi**(int32 x, int32 y);	32 位组合除法和模数
_ _**C6000_divremu**(uint32 x, uint32 y);	32 位无符号组合除法和模数
_ _**C6000_divremull**(uint64 x, uint64 y);	64 位无符号组合除法和模数

表 8-7 中的宽整数算术函数根据 C 语言的语义进行运算。

表 8-7. C6000 宽整数算术

签名	说明
int64 _ _**C6000_negll**(int64 x);	64 位整数取反
uint64 _ _**C6000_mpyll**(uint64 x, uint64 y);	64x64 位相乘
int64 _ _**C6000_mpyiill**(int32 x, int32 y);	32x32 位相乘
uint64 _ _**C6000_mpyuiill**(uint32 x, uint32 y);	32x32 位无符号相乘
int64 _ _**C6000_llshr**(int64 x, uint32 y);	64 位有符号向右移位 (x>>y)
uint64 _ _**C6000_llshru**(uint64 x, uint32 y);	64 位无符号向右移位 (x>>y)
uint64 _ _**C6000_llshl**(uint64 x, uint32 y);	64 位向左移位 (x<<y)

下面的章节将介绍表 8-8 中的其他辅助函数。

表 8-8. C6000 其他辅助函数

签名	说明
void _ _**C6000_strasgi**(int32 *dst, const int32 *src, uint32 cnt);	中断安全块复制；cnt >= 28
void _ _**C6000_strasgi_64plus**(int32*, const inst32*, uint32) ;	中断安全块复制；cnt >= 28
void _ _**C6000_abort_msg**(const char *string);	报告断言失败
void _ _**C6000_push_rts**(void);	压入所有被调用者保存的寄存器
void _ _**C6000_pop_rts**(void);	弹出所有被调用者保存的寄存器
void _ _**C6000_call_stub**(void);	保存调用者保存的寄存器；调用 B31
void _ _**C6000_weak_return**(void);	导入的弱调用的解析目标
void _ _**C6000_get_addr**(ptrdiff_t TPR_offst);	获取线程指针寄存器 (TPR) 偏移量的地址。
void _ _**C6000_get_tp**(void);	获取当前线程的线程指针值。

表 8-8. C6000 其他辅助函数（续）

签名	说明
void * **_ _tls_get_addr**(struct TLS_descriptor);	获取线程局部变量的地址。

_ _C6000_strasgi

函数 _ _C6000_strasgi 由编译器为高效行外结构或数组复制操作生成。cnt 实参表示大小（以字节为单位），它必须是 4 的倍数且大于或等于 28（7 个字）。它进行以下假设：

- src 和 dst 地址按字对齐。
- 源对象和目标对象不重叠。

7 个字的最小值是允许在 C64x+上使用软件流水线循环的阈值。对于较小的对象，编译器通常会生成内联序列的加载/存储指令。_ _C6000_strasgi 不会禁用中断，并且可以安全地中断。

函数 _ _C6000_strasgi_64plus 是 _ _C6000_strasgi 针对 C64x+架构进行了优化的版本。

_ _C6000_abort_msg

生成函数 _ _C6000_abort_msg 是为了在运行时断言（例如 C 断言宏）失败时输出诊断消息。该函数不得返回。也就是说，该函数必须调用中止或通过其他方式终止程序。

_ _C6000_push_rts 和 _ _C6000_pop_rts

在优化代码大小时，在 C64x+ 架构上使用函数 _ _c6x_push_rts。许多函数可保存和恢复大多数或所有被调用者保存的寄存器。为避免复制逻辑程序中的保存代码，并在每个此类函数的收尾程序中恢复代码，编译器可改为采用此库函数。该函数将所有 13 个被调用者保存的寄存器压入堆栈，并根据节 4.4.4 中的协议将 SP 递减 56 个字节。

函数 _ _c6x_push_rts 的实现方式如下所示：

```
__c6xabi_push_rts:
        STW       B14, *B15--[2]
        STDW  A15:A14, *B15--
        STDW  B13:B12, *B15--
        STDW  A13:A12, *B15--
        STDW  B11:B10, *B15--
        STDW  A11:A10, *B15--
        STDW  B3:B2,   *B15--
        B     A3
```

（这是连续的未调度表示法。请参考 TI 运行时库中的源代码以了解实际实现。）

函数 _ _C6000_pop_rts 会根据 _ _C6000_push_rts 压入的方式恢复被调用者保存的寄存器，并将栈递增（弹出）56 个字节。

_ _C6000_call_stub

函数 _ _C6000_call_stub 还用于帮助优化 C64x+ 函数的代码大小。许多调用站点都有多个在整个调用期间活跃的调用者保存的寄存器。这些寄存器不会由调用保留，因此必须由调用者保存和恢复。编译器可以通过 _ _C6000_call_stub 路由调用，该函数会执行以下运算序列：

- 将选定的调用者保存的寄存器保存在堆栈上
- 调用函数
- 恢复保存的寄存器
- 返回

这样，所选的寄存器会在整个调用中保留，而无需调用者保存和恢复它们。由 _ _C6000_call_stub 保留的寄存器有：A0、A1、A2、A6、A7、B0、B1、B2、B4、B5、B6 和 B7。

调用者调用 _ _C6000_call_stub 的方式是将要调用的函数的地址放置在 B31 中，然后跳转到 _ _C6000_call_stub。（如往常一样，返回地址在 B3 中。）

函数 _ _C6000_call_stub 的实现方式如下：

```
__c6xabi_call_stub:
        STW    A2, *B15--[2]
        STDW   A7:A6, *B15--
        STDW   A1:A0, *B15--
        STDW   B7:B6, *B15--
        STDW   B5:B4, *B15--
        STDW   B1:B0, *B15--
        STDW   B3:B2, *B15--
        ADDKPC __STUB_RET, B3, 0
        CALL   B31
__STUB_RET:
        LDDW   *++B15, B3:B2
        LDDW   *++B15, B1:B0
        LDDW   *++B15, B5:B4
        LDDW   *++B15, B7:B6
        LDDW   *++B15, A1:A0
        LDDW   *++B15, A7:A6
        LDW    *++B15[2], A2
        B      B3
```

（这是连续的未调度表示法。请参考 TI 运行时库中的源代码以了解实际实现。）

由于 _ _C6000_call_stub 使用非标准约定，因此无法通过 PLT 条目调用它。其在库中的定义必须标记为 STV_INTERNAL 或 STV_HIDDEN，以防止从共享库中导入它。

_ _C6000_weak_return

函数 _ _C6000_weak_return 是只返回的函数。链接器应将其包含在动态可执行文件或共享对象中（其包含对导入的弱符号的任何未解析调用）。如果这些调用在动态加载时保持未解析状态，动态链接器可以使用它来解析这些调用。

_ _C6000_get_addr

函数 _ _C6000_get_addr 接受 32 位 TPR 偏移量并返回线程局部地址。特殊值 -1 用于指示未定义的弱引用，并且在这种情况下返回 0。在编译静态可执行文件和裸机动态 TLS 访问模型时使用此函数。有关线程局部存储的详细信息，请参阅章节 7。

_ _C6000_get_tp

函数 _ _C6000_get_tp 返回当前线程的线程指针值。此函数不修改返回寄存器 A4 以外的任何寄存器。此函数可以通过 PLT 调用，因此调用者应假设 B30 和 B31 已通过调用此函数进行修改。有关线程局部存储的详细信息，请参阅章节 7 和节 14.1.4。

_ _tls_get_addr

函数 _ _tls_get_addr 返回线程局部变量的地址。有关此函数以及传递给它的 TLS_descriptor 结构以指定线程局部变量的偏移量的详细信息，请参阅节 7.5.1.1。当编译静态可执行文件和裸机动态 TLS 访问模型以外的所有访问模型时使用此函数。有关线程局部存储的详细信息，请参阅章节 7。

8.3 辅助函数的特殊寄存器约定

除前文特别指明的以外，辅助函数遵循标准调用约定。然而，典型的实现只需要可用寄存器的一小部分。如果调用者所使用的寄存器通常需要在调用过程中保留（即调用者保存寄存器），但已知辅助函数不会使用该寄存器，则调用者可以不保存该寄存器。因此，ABI 会按函数更改这些寄存器的命名，以便调用者无需不必要地保留未使用的寄存器。

请注意，从编译器的角度来看，使用此信息是可选项，仅提供优化的机会。从库实现者的角度来看，ABI 要求辅助函数的替代实现必须符合附加限制。

具有特殊寄存器约定的辅助函数不能通过 PLT 条目调用（请参阅节 6.5）。因此，它们的定义必须标记 STV_INTERNAL 或 STV_HIDDEN，防止它们可从共享库导入。

表 8-9 列出了那些具有修改后寄存器保存约定的辅助函数。如果此表中列出了某个函数，则给定的寄存器是调用该函数时唯一修改的寄存器。如果函数未列出，则它遵循标准规则。

表 8-9. 辅助函数的 C6000 寄存器约定

函数	修改的寄存器
_ _C6000_divi	A0、A1、A2、A4、A6、B0、B1、B2、B4、B5、B30、B31
_ _C6000_divu	A0、A1、A2、A4、A6、B0、B1、B2、B4、B30、B31
_ _C6000_remi	A1、A2、A4、A5、A6、B0、B1、B2、B4、B30、B31
_ _C6000_remu	A1、A4、A5、A7、B0、B1、B2、B4、B30、B31
_ _C6000_divremi	A1、A2、A4、A5、A6、B0、B1、B2、B4、B30、B31
_ _C6000_divremu	A0、A1、A2、A4、A6、B0、B1、B2、B4、B30、B31
_ _C6000_strasgi_64plus	A31、A30、B31、B30、ILC、RILC、B30、B31
_ _C6000_push_rts	A15、A3、B3、B30、B31
_ _C6000_pop_rts	B10、B11、B12、B13、B14、B30、B31
_ _C6000_call_stub	A3-A5、A8、A9、A16-A31、B8、B9、B16-B31、ILC、RILC

B30 和 B31 视为会被任何调用修改，即使被调用者不使用它们也不例外。这样一来，它们就可以用作蹦床函数的暂存寄存器。请参阅节 3.7。

8.4 复数类型的辅助函数

这些函数支持复数类型的乘法和除法。该行为由 C99 标准的附录 G 指定。

表 8-10. 复数类型的辅助函数

签名	说明
float64 complex **_ _C6000_mpycd**(float64 complex x, float64 complex y);	双精度复数乘法
float32 complex **_ _C6000_mpycf**(float32 complex x, float32 complex y);	单精度复数乘法
float64 complex **_ _C6000_divcd**(float64 complex x, float64 complex y);	双精度复数除法 (x/y)
float32 complex **_ _C6000_divcf**(float32 complex x, float32 complex y);	单精度复数除法 (x/y)

8.5 C99 的浮点辅助函数

这些函数尚未实现，但名称保留供 C99 编译器使用。TI 库当前未实现这些函数。与 C99 相关的 API 可能发生变化。

表 8-11. 保留的浮点分类辅助函数

签名	说明
int32 _ _**C6000_isfinite**(float64 x);	如果 x 是一个可表示的值，则为真
int32 _ _**C6000_isfinitef**(float32 x);	如果 x 是一个可表示的值，则为真
int32 _ _**C6000_isinf**(float64 x);	如果 x 表示“无穷大”，则为真
int32 _ _**C6000_isinff**(float32 x);	如果 x 表示“无穷大”，则为真
int32 _ _**C6000_isnan**(float64 x);	如果 x 表示“非数字”，则为真
int32 _ _**C6000_isnanf**(float32 x);	如果 x 表示“非数字”，则为真
int32 _ _**C6000_isnormal**(float64 x);	如果 x 未去规范化，则为真
int32 _ _**C6000_isnormalf**(float32 x);	如果 x 未去规范化，则为真
int32 _ _**C6000_fpclassify**(float64 x);	将浮点值分类
int32 _ _**C6000_fpclassifyf**(float32 x);	将浮点值分类

函数 **_ _C6000_fpclassify** 用于将浮点数分类。运行如下：

```
int32 __C6000_fpclassify(float64 x)
{
   if      (isnormal(x)) return 3;
   else if (isinf(x))    return 1;
   else if (isnan(x))    return 2;
   else                  return 4;
}
```

以下 C99 函数执行舍入和向零截断。

表 8-12. 保留的浮点舍入函数

签名	说明
float64 _ _**C6000_nround**(float64 x);	舍入到最接近的整数
float32 _ _**C6000_roundf**(float32 x);	舍入到最接近的整数
float64 _ _**C6000_trunc**(float64 x);	向零截断
float32 _ _**C6000_truncf**(float32 x);	向零截断

章节 9

标准 C 库 API

以下各节描述适用于 C 标准头文件的任何约定。这些议题涵盖虽未在 ANSI C 标准中规定，但必须予以遵循的任何要求，这样才能使工具链支持 C6000 ABI。

9.1 保留符号

与 ABI 相同，保留了许多符号以用于 RTS 库。包括以下符号：

- _ftable
- _ctypes_

此外，还保留了节 13.4.4 中列出的任何符号或节 13.1 中列出的任何带有前缀的符号。

9.2 <assert.h> 实现

该库必须将断言实现为宏。如果其表达式实参为 false，则它最终必须调用辅助函数 __c6xabi_abort_msg 来打印失败消息。辅助函数是否实际导致打印内容是由实现定义的。根据 C 标准的规定，该辅助函数必须通过调用 abort 来终止。请参阅节 8.2。

```
    void __c6xabi_abort_msg(const char *);
```

9.3 <complex.h> 实现

C99 标准要求将复数表示为一个结构，该结构包含一个由相应实数类型的两个元素组成的数组。元素 0 是实部，元素 1 是虚部。例如，_Complex double 为：

```
    { double _val[2]; } /* where 0=real 1=imag */
```

TI 的 C6000 工具集支持 C99 复数并提供此头文件。

9.4 <ctype.h> 实现

ctypes.h 函数与区域设置相关，因此可能不会内联。这些函数包括：

- isalnum
- isalpha
- isblank（C99 函数；TI 工具集尚未提供该函数）
- iscntrl
- isdigit
- isgraph
- islower
- isprint
- ispunct
- isspace
- isupper
- isxdigit
- isascii（此函数已过时，不是标准 C99 函数）
- toupper（当前由 TI 编译器内联，但随时会变更）
- tolower（当前由 TI 编译器内联，但随时会变更）
- toascii（此函数已过时，不是标准 C99 函数）

9.5 <errno.h> 实现

errno 是一个宏，可扩展为涉及函数调用的表达式，如下所示：

```
#define errno (*__c6xabi_errno_addr())
extern int *__c6xabi_errno_addr(void);
```

请注意，该定义受 C6000 线程局部支持的影响。请参阅章节 7。

以下是定义为与 errno 一起使用的一些常量。有关完整列表，请参阅 errno.h 文件。

```
#define EDOM   33
#define ERANGE 34
#define ENOENT 2
#define EFPOS  152
#define EILSEQ 88
```

9.6 <float.h> 实现

该文件中的宏是以自然方式定义的。Float 为 IEEE-32；double 和 long double 为 IEEE-64。

9.7 <inttypes.h> 实现

该文件中的宏、函数和 typedef 都是根据架构的整数类型以自然方式定义的。请参阅节 2.1。

9.8 <iso646.h> 实现

该文件中的宏完全由 C 标准指定，并且是以自然方式定义的。

9.9 <limits.h> 实现

除 MB_LEN_MAX 外，该文件中的宏都是根据架构的整数类型以自然方式定义的。请参阅节 2.1。

MB_LEN_MAX 定义如下：

```
#define MB_LEN_MAX 1
```

9.10 <locale.h> 实现

TI 的工具集仅提供“C”区域设置。LC_* 宏定义如下：

```
#define LC_ALL         0
#define LC_COLLATE     1
#define LC_CTYPE       2
#define LC_MONETARY    3
#define LC_NUMERIC     4
#define LC_TIME        5
```

Lconv 结构中字段的顺序如下：

（这些是 C89 字段。不包括为 C99 添加的其他字段。）

```
char *decimal_point;
char *grouping;
char *thousands_sep;
char *mon_decimal_point;
char *mon_grouping;
char *mon_thousands_sep;
char *negative_sign;
char *positive_sign;
char *currency_symbol;
char  frac_digits;
char  n_cs_precedes;
char  n_sep_by_space;
char  n_sign_posn;
char  p_cs_precedes;
char  p_sep_by_space;
char  p_sign_posn;
char *int_curr_symbol;
char  int_frac_digits;
```

9.11 <math.h> 实现

此库定义的宏必须是浮点常量（不是库变量）。

- HUGE_VALF 必须为 float 无穷大。
- HUGE_VAL 必须为 double 无穷大。
- HUGE_VALL 必须为 long double 无穷大。
- INFINITY 必须为 float 无穷大。
- NAN 必须为无声 NaN。
- 当前未指定 MATH_ERRNO。
- 当前未指定 MATH_ERREXCEPT。

定义了以下 FP_* 宏：

```
#define FP_INFINITE  1
#define FP_NAN       2
#define FP_NORMAL    (-1)
#define FP_SUBNORMAL (-2)
#define FP_ZERO      0
```

当前未指定其他 FP_* 宏。

9.12 <setjmp.h> 实现

jmp_buf 的类型和大小在 setjmp.h 中定义

jmp_buf 的大小和对齐与一个由 13 个“int”组成的数组（即 32 位 * 13）相同。

setjmp 和 longjmp 函数不得内联，因为 jmp_buf 不透明。也就是说，结构体的字段不由标准定义，因此除 setjmp() 和 longjmp() 之外，无法访问结构体的内部，而这两个函数必须来自同一库的外联调用。这些函数不能作为宏实现。

9.13 <signal.h> 实现

TI 的工具集不实现信号库函数。

TI 的工具集会为“int”创建以下 typedef。

```
    typedef int sig_atomic_t;
```

TI 的工具集定义以下常量：

```
    #define SIG_DFL ((void (*)(int)) 0)
    #define SIG_ERR ((void (*)(int)) -1)
    #define SIG_IGN ((void (*)(int)) 1)
    #define SIGABRT 6
    #define SIGFPE  8
    #define SIGILL  4
    #define SIGINT  2
    #define SIGSEGV 11
    #define SIGTERM 15
```

9.14 <stdarg.h> 实现

接口中仅显示 va_list 类型。宏用于实现 va_start、va_arg 和 va_end。有关 va_list 中实参的格式，请参阅章节 3。

一旦调用以省略号 (…) 声明的可变实参 C 函数，最后声明的实参和任何其他实参将立即如节 3.3 中所述在栈上传递，并使用 <stdarg.h> 中的宏进行访问。这些宏使用持久实参指针，该指针通过调用 va_start 初始化，并通过调用 va_arg 进行高级操作。以下约定适用于这些宏的实现。

- va_list 的类型是 char *。
- 调用宏 va_start(ap, parm) 会将 ap 设置为指向分配给 parm 的最后一（最大）地址之后 1 个字节。
- 每次接连调用 va_arg(ap, type)，都会使 ap 指向为实参对象（通过类型来表示）保留的最后地址之后 1 个字节。

9.15 <stdbool.h> 实现

对于 C++，类型“bool”属于内置类型。

对于 C99，类型“_Bool”属于内置类型。对于 C99，头文件 stdbool.h 会定义一个将扩展为 _Bool 的宏“bool”。

这些类型各自都表示为一种 8 位无符号类型。

9.16 <stddef.h> 实现

类型 size_t 和 ptrdiff_t 在 stddef.h 中定义。请参阅节 2.1。

9.17 <stdint.h> 实现

该头文件中的宏和类型定义会根据架构的整数类型，以自然方式进行定义。请参阅节 2.1。

9.18 <stdio.h> 实现

TI 工具集定义了以下与 stdio.h 库一起使用的常量：

```
#define _IOFBF 1
#define _IOLBF 2
#define _IONBF 4
#define BUFSIZ 256
#define EOF (-1)
#define FOPEN_MAX
#define FILENAME_MAX
#define TMP_MAX
#define L_tmpnam
#define SEEK_SET 0
#define SEEK_CUR 1
#define SEEK_END 2
#define stdin &_ftable[0]
#define stdout &_ftable[1]
#define stderr &_ftable[2]
```

FOPEN_MAX、FILENAME_MAX、TMP_MAX 和 L_tmpnam 值实际上是最小极大值。该库可以自由地支持更多/更大的值，但必须至少提供指定的值。

由于 TI 工具集将 stdout 和 stderr 定义为 &_ftable[1] 和 &_ftable[2]，因此 FILE 的大小对实现来说必须已知。

在 TI 头文件中，stdin、stdout 和 stderr 扩展为数组 _ftable 中的引用。为了成功地与此类文件互连，任何其他实现都需要准确使用该名称来实现 FILE 数组。C6000 EABI 没有“兼容模式”（与 ARM EABI 中的模式类似），在这种模式下，stdin、stdout 和 stderr 是链接时符号，而非宏。缺少兼容模式意味着，对于那些需要与直接引用 stdin 的模块互连的链接器，此时需要支持 _ftable。

如果程序不使用 stdin、stdout 或 stderr 宏（或实现为宏且引用上述宏之一的函数），则 FILE 数组没有问题。

通常实现为宏的 C I/O 函数（getc、putc、getchar、putchar）不得内联。

fpos_t 类型定义为 int。

9.19 <stdlib.h> 实现

如下所示，TI 工具集定义了 stdlib.h 结构：

```
typedef struct { int quot; int rem; } div_t;
typedef struct { long int quot; long int rem; } ldiv_t;
typedef struct { long long int quot; long long int rem; } lldiv_t;
```

如下所示，TI 工具集定义了与 stdlib.h 库一起使用的常量：

```
#define EXIT_SUCCESS  0
#define EXIT_FAILURE  1
#define MB_CUR_MAX 1
```

rand 函数的结果不由 ABI 规范定义。该函数必须为线程局部函数。请参阅章节 7。

此 ABI 规范不需要库来实现 getenv 或系统函数。TI 工具集确实提供了 getenv 函数，这需要调试程序支持。TI 工具集不提供系统函数。

9.20 <string.h> 实现

strtok 函数不得内联，因为它具有静态状态。strcoll 和 strxfrm 函数也不能内联，因为它们依赖区域设置。

9.21 <tgmath.h> 实现

C99 标准全面规定了该头文件。TI 工具集不提供此头文件。

9.22 <time.h> 实现

为该库定义的一些类型定义和常量依赖于执行环境。为了使代码可移植，代码不得对 time_t 或 clock_t 的类型和范围做出假设。

CLOCKS_PER_SEC 的类型为 clock_t。

9.23 <wchar.h> 实现

TI 工具集定义了以下与此库一起使用的类型和常量：

```
typedef int wint_t;
#define WEOF ((wint_t)-1)
```

类型 mbstate_t 是 int 的大小和对齐。。

9.24 <wctype.h> 实现

TI 工具集定义了以下与此库一起使用的类型：

```
typedef void * wctype_t;
typedef void * wctrans_t;
```

章节 10

C++ ABI

C++ ABI 指定 C++ 语言实现中的一些方面，为了使不同工具链中的代码能够互操作，必须对这些方面进行标准化。C6000 C++ ABI 基于最初为 IA-64 开发的通用 C++ ABI，但现在广泛用于 C++ 工具链，包括 GCC。基本标准称为 “GC++ABI”，可在 http://refspecs.linux-foundation.org/cxxabi-1.83.html 上找到。

本节说明该基础文档的增补和偏离。

10.1 限制 (GC++ABI 1.2)

由于 RTTI 实现，GC++ABI 将包含非虚拟基址子对象的完整对象中的偏移量约束为由 56 位有符号整数表示。对于 C6000 ，约束减至 24 位。这意味着对于基址类大小的实际限制为 2^{23} -1（或 0x7fffff）字节。

10.2 导出模板 (GC++ABI 1.4.2)

ABI 当前未指定导出模板。

10.3 数据布局（GC++ABI 第 2 章）

POD（简单旧数据）的布局在本文档 章节 2 中指定。非 POD 数据的布局由基础文档指定。对于位字段，布局略有不同，在 节 2.7 中介绍了这些字段。

10.4 初始化保护变量 (GC++ABI 2.8)

保护变量是一种单字节字段，存储在 32 位容器的第一个字节中。保护变量的非零值指示初始化已完成。这遵循 IA-64 方案，但容器为 32 位，而不是 64 位。

下面是辅助函数 _ _cxa_guard_acquire 的参考设计，该函数读取保护变量，如果初始化尚未完成就返回 1，否则返回 0：

```
int __cxa_guard_acquire(unsigned int *guard)
{
   char *first_byte = (char *)guard;
   return (*first_byte == 0) ? 1 : 0;
}
```

下面是辅助函数 _ _cxa_guard_release 的参考设计，该函数修改保护对象，指示初始化已完成：

```
void __cxa_guard_release(unsigned int *guard)
{
   char *first_byte = (char *)guard;
   *first_byte = 1;
}
```

10.5 构造函数返回值 (GC++ABI 3.1.5)

C6000 遵循 ARM EABI，C1 和 C2 构造函数根据它返回 *this* 指针。这样就可以对这些函数的调用进行尾调用优化。

类似地，对 D1 和 D2 析构函数的非虚拟调用返回 'this'。对虚拟析构函数的调用使用 thunk 函数，它不返回 'this'。

GC++ABI 的第 3.3 节为数组 new 和 delete 指定了几个库辅助函数，这些函数采用指向构造函数或析构函数的指针作为形参。在 GC++ABI 中，这些形参被声明为返回 void 的函数的指针，但在 C6000 ABI 中，它们被声明为返回 void *（对应于 'this'）的函数的指针。

10.6 一次性构建 API (GC++ABI 3.3.2)

保护变量是一个 8 位字段，存储在 32 位容器的第一个字节中。请参阅节 10.4。

10.7 控制对象构造顺序 (GC++ ABI 3.3.4)

C6000 ABI 没有指定控制对象构造的机制。

10.8 还原器 API (GC++ABI 3.4)

C6000 ABI 不再要求实现需提供函数 _ _cxa_demangle，该函数为还原器提供运行时接口。

10.9 静态数据 (GC++ ABI 5.2.2)

GC++ ABI 要求在 COMDAT 组中定义由内联函数引用的静态对象。如果此类对象具有关联的保护变量，则必须同样在 COMDAT 组中定义保护变量。GC++ABI 允许静态变量及其保护变量位于不同的组中，但不鼓励这种做法。C6000 ABI 完全禁止这种做法；静态变量及其保护变量必须在单个 COMDAT 组中定义，并以静态变量的名称作为签名。

10.10 虚拟表和键函数 (GC++ABI 5.2.3)

GC++ ABI 将类的键函数（它的定义会触发为该类创建虚拟表）定义为第一个*在类定义点*不内联的非纯虚拟函数。C6000 ABI 将其修改为第一个*在转换单元末尾*不内联的非纯虚拟函数。换句话说，如果内联成员在类定义后第一个被声明为内联，则它不是键函数。

10.11 回溯表位置 (GC++ABI 5.3)

本文档的章节 11 中介绍了异常处理。

章节 11

异常处理

C6000 EABI 采用表驱动的异常处理 (TDEH)。对于支持异常的语言（例如 C++），TDEH 可实现异常处理。

TDEH 使用表来编码处理异常所需的信息。这些表是程序只读数据的一部分。发生异常时，运行时支持库中的异常处理代码会通过将栈*展开*为表示函数的栈帧来传播异常，该函数具有捕获异常的 catch 子句。展开栈时，必须在该过程中销毁（通过调用析构函数）局部定义的对象。这些表对有关如何展开栈、何时销毁哪些对象以及最终捕获异常时将控制权转移到何处的信息进行了编码。

链接器使用由编译器生成的可重定位文件中的信息来将 TDEH 表生成为可执行文件。本节指定表的格式和编码，以及如何使用信息来传播异常。符合 ABI 的工具链必须以此处指定的格式来生成表。

11.1 概述

C6000 异常处理表的格式和机制基于 ARM 处理器系列的格式和机制，而 ARM 处理器系列本身基于 IA-64 异常处理 ABI (http://www.codesourcery.com/public/cxx-abi/abi-eh.html)。本节重点介绍特定于 C6000 的部分。

TDEH 数据包括三个主要部分：EXIDX、EXTAB 以及 catch 和 cleanup 块。

异常索引表 (EXIDX) 将程序地址映射到异常操作表 (EXTAB) 中的条目。EXIDX 涵盖程序中的所有地址。

EXTAB 对指令进行编码，以说明如何回溯栈帧（通过恢复寄存器和调整栈指针）以及在传播异常时调用哪些 catch 和 cleanup 块。

catch 和 cleanup 块（统称为*着陆垫*）是执行异常处理任务的代码片段。cleanup 块包含对析构函数的调用。catch 块在用户代码中实现 catch 子句。这些块仅在实际引发异常时执行。当生成函数的其余部分时，会为函数生成这些块，并在与函数相同的栈帧中执行，但可能会放在不同的段中。

11.2 PREL31 编码

EXIDX 和 EXTAB 表的某些字段需要记录程序存储器地址或指向表中其他位置的指针，这两者通常位于代码段或只读段中。为了确保位置无关性，这通过称为 R_C6000_PREL31（以下简写为 PREL31）的专用 PC 相对重定位来完成。PREL31 字段编码为经缩放的有符号 31 位偏移量，它占用 32 位字的最低有效 31 位。剩余（最高有效）位在不同的上下文中用于不同目的。通过将编码的偏移量左移 1 位并将其添加到字段地址，可找到该字段引用的重定位地址。

11.3 异常索引表 (EXIDX)

当源代码中出现抛出语句时，编译器会对名为 _ _cxa_throw 的运行时支持库函数生成调用。当执行抛出时，_ _cxa_throw 调用点的返回地址用于识别哪个函数正在抛出异常。库会在 EXIDX 表中搜索返回地址。

表中的每个条目分别代表一个程序地址范围的异常处理行为，它们可能是一个或多个有着完全相同异常处理行为的函数。每个条目分别对程序地址范围的开头进行编码，并且视为覆盖所有程序地址，直到下一个条目中编码的地址为止。链接器可以将行为相同的相邻函数组合到一个条目中。

每个条目由两个 32 位字组成。每个条目的第一个字是 PREL31 字段，代表一个或多个函数的起始程序地址。第一个字的第 31 位应为 0。第二个字有三种格式，取决于第二个字的第 31 位。如果第 31 位为 0，则第二个字是 PREL31 指针（指向存储器中其他位置的 EXTAB 条目），或者是特殊值 EXIDX_CANTUNWIND。如果第 31 位为 1，则第二个字是内联 EXTAB 条目。后续各小节将详细介绍这三种格式。

11.3.1 指向行外 EXTAB 条目的指针

在此格式中，EXIDX 表条目的第二个字的最高位以及此地址范围中其他位的 EXTAB 条目的 PREL-31 编码地址包含 0。

31	30-0
0	PREL31 Representation of function address
0	PREL31 Representation of EXTAB entry

11.3.2 EXIDX_CANTUNWIND

特殊情况下，如果 EXIDX 的第二个字的值为 0x1，则 EXIDX 表示 EXIDX_CANTUNWIND，指示该函数根本无法展开。如果异常尝试通过此类函数来传播，则展开程序将调用 abort 或 std::terminate，具体取决于语言。

31	30-0
0	PREL31 Representation of function address
0x00000001 (EXIDX_CANTUNWIND)	

11.3.3 内联 EXTAB 条目

如果用于该函数的整个 EXTAB 条目足够小，则可将其置于第二个 EXIDX 字中，并将高位设置为 1。第二个字采用的编码方式与 节 11.4 中描述的 EXTAB 紧凑模型相同，但没有描述符，也没有终止 NULL。这样可节省 4 个本来是指向行外 EXTAB 条目的指针的字节，以及 4 个用于终止 NULL 的字节。

31	30-28	27-24	23-0
0	PREL31 Representation of function address		
1	**000**	PR Index	Data for personality routine specified by 'index'

11.4 异常处理指令表 (EXTAB)

每个 EXTAB 条目都是一个或多个 32 位字，用于对帧回溯指令和描述符进行编码，以便处理捕获和清理。第一个字描述该条目的*个性*，即条目的格式和解释。

当抛出异常时，EXTAB 条目会通过运行时支持库中提供的“个性例程”进行解码。表 11-1 中列出了 ABI 规定的个性例程。

11.4.1 EXTAB 通用模型

通过将第一个字的位 31 设置为 0 来指示通用 EXTAB 条目。第一个字具有 PREL31 条目，其表示个性例程的地址。EXTAB 条目中的其余字是传递至个性例程的数据。

31	30-0
0	PREL31 Representation of personality routine address
Optional data for the personality routine	

可选数据的格式由个性例程决定，但长度必须是完整 32 位字的整数倍。展开程序调用个性例程，并将指向可选数据的第一个字的指针传递给该个性例程。

11.4.2 EXTAB 紧凑模型

紧凑型 EXTAB 条目由第一个字的位 31 中的 1 指示。（将 EXTAB 条目编码到 EXIDX 条目的第二个字中时，始终使用紧凑形式。）在紧凑形式中，个性例程由条目第一个字节中的 4 位 PR 索引编码。其余 3 个字节包含由个性例程指定的展开指令。在非内联 EXTAB 条目中，以附加连续 32 位字的形式提供附加数据：任何附加展开指令都可选地后跟操作描述符，并以 NULL 字终止。

31	30-28	27-24	23-0
1	**000**	PR Index	Encoded unwinding instructions
Zero or more additional 32-bit words of unwinding instructions (out-of-line EXTAB only)			
Zero or more catch, cleanup, or FESPEC descriptors (out-of-line EXTAB only)			
32-bit NULL terminator (out-of-line EXTAB only)			

11.4.3 个性化例程

C6000 具有以下 ABI 指定的个性化例程。前三个与 ARM EABI 的格式相同。下表列出了个性化例程及其 PR 索引。

表 11-1. C6000 TDEH 个性化例程

PR 索引（位 27-24）	个性化	例程名称	回溯指令	范围字段的宽度	注释
0000	PR0 (Su16)	_ _C6000_unwind_cpp_pr0	最多 3 个一字节指令	16	
0001	PR1 (Lu16)	_ _C6000_unwind_cpp_pr1	无限个一字节指令	16	
0010	PR2 (Lu32)	_ _C6000_unwind_cpp_pr2	无限个一字节指令	32	如果 16 位范围字段无法达到，则必须使用
0011	PR3	_ _C6000_unwind_cpp_pr3	24 位	16	特定于 C6x 的优化回溯格式
0100	PR4	_ _C6000_unwind_cpp_pr4	24 位	16	与 PR3 相同，但函数收尾程序使用另一 C64x+ *紧凑帧* 布局。

使用紧凑模型 EXTAB 条目时，可重定位文件必须以 R_C6000_NONE 重定位的形式包含 EXTAB 段对相应个性化例程符号的引用，以此来明确指明它所依赖的例程。

11.5 回溯指令

回溯帧时，是通过模拟函数的收尾程序来执行的。在函数收尾程序中可执行的任何操作都需要在 EXTAB 条目中进行编码，这样栈回溯器就可以将信息解码并模拟收尾程序。

回溯指令会对 C6x 栈布局做出假设；特别是，总是假设*被调用者保存的寄存器安全调试顺序* ，但当使用 C64x+-specific _ _C6000_push_rts 布局时除外。

11.5.1 通用序列

理论上，所有展开序列都采用以下形式：

1. 恢复 SP
   a. 如果使用了 FP，则 SP := FP
   b. 否则，SP := SP + 常量
2. （可选）从被调用者保存的寄存器恢复 B3
3. （可选）恢复被调用者保存的寄存器（reg1 := SP[0]；reg2 := SP[-1]，以此类推）
4. 通过 B3 返回

步骤 1：恢复 SP

在恢复被调用者保存的寄存器之后，实际的收尾程序才会恢复 SP，但由于堆栈展开是虚拟操作，TDEH 的仿真展开可能会先执行 SP 恢复。这简化了其他被调用者保存的寄存器的恢复操作。

SP 将通过从 FP 复制或递增一个常量来恢复。在后一种情况下， 除了显式递增之外，SP 也会隐式递增，以考虑被调用者保存的区域的大小。如果 SP 从 FP 恢复，则不暗示该额外递增。

第 2 步：恢复 B3

在返回发生之前，返回地址必须位于 B3 中。如果它存储在被调用者保存的寄存器（比如 “R” ）中，则需要在步骤 3 恢复 R 本身之前从 R 中恢复 B3。

步骤 3：恢复寄存器

理论上，被调用者保存的寄存器以*寄存器安全调试*顺序（节 4.4.2）恢复，从（旧的）SP 指向的位置开始，并移动到较低的地址。除非使用 C6000_push_rts 布局（节 4.4.4），否则 TDEH 会强制执行安全调试排序。

对于使用*压缩帧*方法（节 4.4.4）创建的堆栈帧，由于优化有利于可压缩指令，已保存的寄存器之间可能存在间隙。展开程序必须知道用于布置寄存器和相应调整寄存器位置的算法。

在大端字节序模式下，为了方便使用 LDDW 和 STDW，如果一对中的两个寄存器占用相同的对齐双字，则交换对的顺序。该值是在使用安全调试顺序确定布局之后计算的，因此不会交换某些寄存器对。

通常，SP (B15) 不是通过显式寄存器恢复来恢复的；当 FP 不可用时，它是针对 DATA_MEM_BANK 布局显式恢复的（节 4.4.3）。

步骤 4：返回

每个展开序列以隐式或显式“RET B3”结束，这表示当前帧的展开已完成。

11.5.2 字节编码展开指令

个性化例程 PR0、PR1 和 PR2 使用字节编码的指令序列来说明如何展开帧。前几条指令被封装到 EXTAB 第一个字的剩余三个字节中；附加指令被封装到后续字中。最后一个字中未使用的字节由“RET B3”指令填充。

尽管指令是字节编码的，但它们始终封装成从 MSB 开始的 32 位字。因此，在小端字节序模式下，第一个展开指令将不会位于最低地址字节。

个性化例程 PR0 最多允许三条展开指令，所有这些指令都存储在第一个 EXTAB 字中。如果有三个以上的展开指令，则必须使用其他个性化例程之一。

31	30-28	27-24	23-16	15-8	7-0
1	**000**	0000 (PR0)	First unwind instruction	Second unwind instruction	Third unwind instruction
Optional descriptors					
NULL					

对于 PR1 和 PR2，位 23-16 编码展开指令的额外 32 位字的数量，该数量可为 0。

31	30-28	27-24	23-16	15-8	7-0
1	**000**	PR Index	Number of additional unwinding words	First unwind instruction	Second unwind instruction
Third unwind instruction			Fourth unwind instruction	...	...
Optional descriptors					
NULL					

表 11-2 总结了展开指令集。表格后面对每条指令进行了更详细的介绍。

表 11-2. 堆栈展开指令

编码	指令	说明
00kk kkkk	SP += (k << 3) + 8	将 SP 递增，增量为一个小常数
1101 0010 kkkk kkkk ...	SP += (ULEB128 << 3) + 0x408	将 SP 递增，增量为一个 ULEB128 编码的常数
1000 0000 0000 0000	CANTUNWIND	函数无法展开，但可能会捕获异常
100x xxxx xxxx xxxx	POP bitmask	弹出一个或多个寄存器 [x != 0]
101x xxxx xxxx xxxx	POP bitmask	从 C64x + 压缩帧 [x ! = 0] 中弹出一个或多个寄存器
1100 nnnn xxxx xxxx ...	POP register	n 表示要弹出的寄存器数量，这些寄存器在后面的 4 位半字节中进行编码
1101 0000	MV FP, SP	从 FP 恢复 SP 而不是将 SP 递增
1101 0001	_ _C6000_pop_rts	模拟对 _ _C6000_pop_rts 的调用
1110 0111	RET B3	此帧的展开完成
1110 xxxx	RETURN 或恢复 B3	b3 : = 寄存器 x (x != B3)

所有其他位模式均保留。

以下各段详细说明了展开指令的解释。

小增量

7	6	5	4	3	2	1	0
0	**0**	k	k	k	k	k	k

k 的值从编码的低 6 位中提取。此指令可以使 SP 递增一个介于 0x8 到 0x200 之间的值，包括边界值。0x208 到 0x400 范围内的增量应该用这些指令中的两个来完成。

大增量

7	6	5	4	3	2	1	0
1	**1**	**0**	**1**	**0**	**0**	**1**	**0**
k	k	k	k	k	k	k	k
...							

值 ULEB128 在 8 位操作码之后的字节中进行 ULEB128 编码。此指令可以将 SP 递增 0x408 或更大的值。增量小于 0x408 应使用 1 条或 2 条小增量指令实现。

CANTUNWIND

7	6	5	4	3	2	1	0
1	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**

此指令指示函数不能展开，通常是因为它是中断函数。但是，中断函数仍然可以有 try/catch 代码，因此 EXIDX_CANTUNWIND 不适用。

POP Bitmask

7	6	5	4	3	2	1	0
1	**0**	**0**	A15	B15	B14	B13	B12
B11	B10	B3	A14	A13	A12	A11	A10

这条两个字节的指令指示：根据位掩码指定，最多应该从虚拟栈弹出 13 个被调用者保存的寄存器。恢复寄存器的顺序必须与它们在*安全调试*顺序中的出现顺序相同。

当使用“POP bitmask”指令弹出任何寄存器时，SP 首先根据被调用者保存的寄存器区域的大小**隐式递增**，四舍五入为最多 8 个字节。这是对任何显式 SP 增量指令的补充。但是，如果使用过“MV FP, SP”指令，则“POP bitmask”**不会**隐式递增 SP。

POP Bitmask; C64x+ Compact Frame

7	6	5	4	3	2	1	0
1	**0**	**1**	A15	B15	B14	B13	B12
B11	B10	B3	A14	A13	A12	A11	A10

与 POP Bitmask 相同，但指示使用 C64x+ *紧凑帧*布局，这可能会在堆栈上留下空洞以有利于使用 SP 自动递减存储。展开程序必须知道用于放置空洞并进行相应补偿的算法。

POP Register

7	6	5	4	3	2	1	0
1	**1**	**0**	**0**	n			
r0				r1			
r2				...			

如果编译器无法保持安全调试顺序，或者对于选择不同布局的编译器，可以个别弹出每个被调用者保存的寄存器。4 位操作码后的前四位指示要弹出的寄存器数量。每个后续的 4 位半字节表示被调用者保存的寄存器的编码，或表示*空洞*的特殊值 0xF。如果指示有空洞，则应递减虚拟 SP，但不应弹出寄存器。

4 位寄存器编码如下：

表 11-3. 展开指令中的寄存器编码

编码	寄存器	编码	寄存器
0000	A15	1000	A14
0001	B15	1001	A13
0010	B14	1010	A12
0011	B13	1011	A11
0100	B12	1100	A10
0101	B11	1101	被保留
0110	B10	1110	保留
0111	B3	1111	“空洞”

MV FP, SP

7	6	5	4	3	2	1	0
1	**1**	**0**	**1**	**0**	**0**	**0**	**0**

此指令从 FP (A15) 恢复 SP，而不是递增 SP。当 FP 可用时，只需从 FP 恢复 SP 值会容易一些。对于 DATA_MEM_BANK 布局，这可能是恢复 SP 的唯一方法。

_ _C6000_pop_rts

7	6	5	4	3	2	1	0
1	**1**	**0**	**1**	**0**	**0**	**0**	**1**

此指令指示所有寄存器恢复都通过调用 _ _C6000_pop_rts 来完成。此函数的行为应由展开程序进行模拟。_ _C6000_pop_rts 隐式恢复 B3 并执行 RET B3。

恢复 B3

7	6	5	4	3	2	1	0
1	**1**	**1**	**0**	r	r	r	R

如果 r 表示 B3 以外的任何寄存器，则此指令对用于从“reg”恢复 B3 的“MV reg, B3”进行编码。如果 POP 会覆盖寄存器，则必须在任何 POP 指令之前执行此操作。

RET B3

7	6	5	4	3	2	1	0
1	**1**	**1**	**0**	**0**	**1**	**1**	**1**

此指令对模拟的 *return* 进行编码，表示此帧的展开已完成。请注意，编码与“恢复 B3”相同，但源寄存器指示为 B3 本身。

每个展开指令序列都以显式或隐式“RET B3”结束。可以从显式展开指令中省略此指令，展开程序将隐式添加它。

11.5.3 24 位展开编码

PR3 和 PR4 使用经过优化的编码。大多数函数都使用 PR3。如果您正在针对大小优化 C64x+ 代码，请使用 PR4。

31	30-28	27-24	23-17	16-4	3-0
1	**000**	Index	Stack increment	Register bitmask	Return register

堆栈增量类似于字节编码的小常数增量，但不会偏置 8。特殊增量值 0x7F 用于编码“MV FP, SP”。如果堆栈增量的值不是 0x7F，则 SP 递增（值 << 3）。

返回寄存器字段使用表 11-3 中的编码对存储返回地址的寄存器进行编码。如果此寄存器是除 B3 本身之外的任何寄存器，则在执行 POP 操作（下一段）之前，应从此寄存器恢复 B3。

该位掩码在字节编码的 POP 位掩码指令中进行解释。如果个性例程为 PR3，则执行非紧凑 POP 指令；如果个性例程为 PR4，则执行紧凑帧 POP 指令。这包括 SP 的可能隐式递增。

11.6 描述符

如果需要销毁任何局部对象，或者如果该函数捕获到异常，则 EXTAB 包含*描述符*，描述待执行的操作以及所针对的异常类型。

如果存在，则描述符遵循展开指令。描述符格式为：一个描述符条目序列，后跟 32 位零 (NULL) 字。每个描述符都以*范围*开头，该开头标识描述符类型，并指定了描述符适用的程序地址范围。其他的描述符特定字跟在范围后面。

应按深度优先顺序列出描述符，以便能够一次处理所有适用描述符。

带描述符的 EXTAB 条目的一般形式如下：

31	30-28	27-24	23-0
1	**000**	PR Index	Unwinding instructions
Zero or more additional 32-bit words of unwinding instructions			
Zero or more catch, cleanup, or FESPEC descriptors			
32-bit NULL terminator			

11.6.1 类型标识符编码

Catch 描述符和 FESPEC 描述符 (节 11.6.5) 可对类型标识符进行编码，用于根据 catch 子句和异常规范来匹配抛出对象的类型。编码这些字段，以便引用对应于指定类型的 type_info 对象。特殊重定位类型 R_C6000_EHTYPE 用于在 EXTAB 中标记 type_info 引用。

链接器将 type_info 字段编码为 type_info 对象的 DP 相对偏移量，从而使表与位置无关。偏移量是相对于模块的静态基址的，该模块定义了包含所引用 catch 子句或异常规范的函数。

11.6.2 作用域

作用域可以标识描述符类型，并指定一个应进行操作的程序地址范围。该范围对应一个可能抛出的调用点。回溯器会在描述符列表中查找哪一个描述符的作用域包含调用点；找到匹配项后，将会激活该描述符。

作用域会对程序地址范围进行编码，方法是指定函数起始地址的偏移量和长度，二者均以字节为单位。如果长度和偏移量都适合 15 位无符号字段，则作用域将使用短格式编码，并且 EXTAB 条目的其余部分可以编码为 PR0、PR1、PR3 或 PR4。如果长度或偏移量超过 15 位，则作用域将使用长格式编码，并且必须使用 PR2。

31-17	16	15-1	0
Length	**X**	Offset	**Y**
Data for descriptor			

图 11-1. 短格式作用域

短格式作用域不能与 PR2 (Lu32) 一起使用。

31-1	0
Length	**X**
Offset	**Y**
Data for descriptor	

图 11-2. 长格式作用域

如果长度或偏移量需要长格式作用域，则必须使用个性例程 PR2 (Lu32)。

作用域编码中的 X 位和 Y 位会指示作用域后面的描述符种类：

X	Y	Descriptor
0	0	Cleanup descriptor
1	0	Catch descriptor
0	1	Function exception specification (FESPEC) descriptor

11.6.3 Cleanup 描述符

Cleanup 描述符控制析构以下局部对象：已完全构造并且即将超出范围、因而必须被析构的局部对象。

31-0

Scope (long or short form)	
0	PREL31 program address of landing pad

Cleanup 描述符仅包含指向清理代码块的单个指针（该代码块包含对析构函数的一个或多个调用）。

11.6.4 catch 描述符

catch 描述符控制何时捕获哪些异常。一个函数可能包含几个 catch 子句，每个子句都应用于可能抛出的函数调用的不同子集。一个调用点可以含有多个 catch 描述符，每个描述符具有不同的类型。

如果 catch 描述符中的类型与抛出的类型匹配，则控制权被转移给 *landing pad*（表示 catch 块的代码片段）。catch 块在用户代码中实现 *catch* 子句。这些块仅在实际引发异常时执行。当生成函数的其余部分时，会为函数生成这些块，并在与函数相同的栈帧中执行，但可能会放在不同的段中。

31-0

Scope (long or short form)	0
R	PREL31 program address of landing pad
Type	

如果位 R 为 1，则 catch 子句的类型是由 TYPE 表示的引用类型。如果位 R 为 0，则类型不是引用类型。

类型字段是对 type_info 对象的引用(通过 R_C6000_EHTYPE relocation 进行重定位）或两个特殊值之一：

- 特殊值 0xFFFFFFFF (-1) 表示*任何*类型 [”catch(...)”]。
- 特殊值 0xFFFFFFFE (-2) 既表示*任何*类型 [”catch(...)”]，又指示个性化例程应立即返回 _URC_FAILURE。在这种情况下，landing pad 地址应设置为 0。此习语可用于防止异常传播到该范围涵盖的代码之外。

11.6.5 函数异常规范 (FESPEC) 描述符

FESPEC 描述符强制执行用户代码中的 throw() 声明。如果使用了抛出声明，则将为该函数创建 FESPEC 描述符，以确保仅抛出列出的那些类型。如果抛出了未列出的类型，展开器通常就会调用 std::unexpected（但也存在例外）。

31-0

Scope (long or short form)	
D	Number of type info pointers
Reference to type_info object	
Reference to type_info object	
. . .	
0	(if D == 1) PREL31 program address of landing pad

描述符的第一个字由 31 位无符号整数组成，用于指定后跟 type_info 字段的数量。

如果位 D 为 1，则 type_info 列表后跟 32 位字，其中包含代码片段的 PREL31 程序地址，如果列表中没有与抛出类型相匹配的类型，则调用该代码片段。该字的位 31 设置为 0。

如果位 D 为 0，且列表中没有与抛出类型相匹配的类型，则展开代码应调用 _ _cxa_call_unexpected。如果任意描述符与该形式匹配，则 EXTAB 段必须包含至 _ _cxa_call_unexpected 的 R_C6000_none 重定位。

11.7 特殊段

所有异常处理表都存储在两个段中。EXIDX 表存储在名为 .C6000.exidx 且类型为 SHT_C6000_UNWIND 的段中。链接器必须将所有输入 .C6000.exidx 段合并为一个连续的 .C6000.exidx 输出段，并且保持与它们引用的代码段相同的相对顺序。也就是说，EXIDX 表中的条目按地址排序。可重定位文件中的每个 EXIDX 段都必须设置 SHF_LINK_ORDER 标志，以便指示此要求。

EXTAB 表存储在名为 .C6000.extab 且类型为 SHT_PROGBITS 的段中。EXTAB 不需要连续，也没有排序要求。

异常表可以链接到存储器中的任何位置。对于动态链接的模块，这些表应与代码放置在同一段中，以便实现位置无关性。

11.8 与非 C++ 代码交互

11.8.1 EXIDX 条目自动生成

如果函数没有 EXIDX 条目，链接器会自动为其创建一个，因在使用 TDEH 的应用程序中可以使用库中在未启用异常处理的情况下编译的函数（例如仅支持 C 语言的库）。自动生成的条目将是 EXIDX_CANTUNWIND，因此，如果在未启用异常处理支持的情况下编译的函数调用会传播异常的函数，则将调用 std::terminate 并且应用程序将停止。

11.8.2 手工编码的汇编函数

手工编码的汇编函数可用于处理或传播异常。仅当函数调用可能传播异常的函数时才需要这样做，并且该异常必须传播到汇编函数之外。用户必须创建适当的 EXIDX 条目以及至少包含展开指令的 EXTAB。

11.9 与系统功能交互

11.9.1 共享库

异常处理表可以在可执行文件或共享库内传播异常。在不同负载模块之间的各调用中传播异常需要操作系统的帮助。

11.9.2 覆盖块

覆盖块不得包含可能会传播异常的 C++ 函数。EXIDX 查询表不处理覆盖函数，并且无法区分特定位置的不同函数。

11.9.3 中断

中断、硬件异常和操作系统信号都不能由异常直接处理。

中断函数可能在任何位置发生，因此我们不支持从中断函数传播异常。所有中断函数都会是 EXIDX_CANTUNWIND。但是，中断函数可调用本身可能抛出异常的函数，因此中断函数必须位于 EXIDX 表中，并且可具有描述符，但永远不会有展开指令。

希望使用异常来表示中断的应用必须安排使用中断函数来捕获中断，该函数必须设置全局易失性对象来指示中断已发生，然后使用该变量的值来在中断函数返回后抛出异常。

如果操作系统提供信号，则必须类似地处理表示信号的异常。

11.10 TI 工具链中的汇编语言运算符

这些实现细节与 TI 工具链相关，而不是 ABI 的一部分。

TI 编译器使用特殊的内置汇编器函数来向汇编器指示异常处理表中的某些表达式应该进行特殊处理。

$EXIDX_FUNC

该实参是要使用 PREL31 表示法进行编码的函数地址。

$EXIDX_EXTAB

该实参是要使用 PREL31 表示法进行编码的 EXTAB 标号。

$EXTAB_LP

该实参是要使用 PREL31 表示法进行编码的 landing pad 标号。

$EXTAB_RTTI

该实参是表示类型的唯一 type_info 对象的标号。（这些对象是为识别运行时类型而生成的。）该字段会在 R_C6000_EHTYPE 重定位时进行重定位。

$EXTAB_SCOPE

该实参是函数的偏移量。该表达式将在作用域描述符中用于指示应在函数的哪个部分应用。

章节 12

DWARF

C6000 使用 DWARF Debugging Information Format Version 3（也称为 DWARF3）来表示目标文件中符号调试器的信息。http://www.dwarfstd.org/doc/Dwarf3.pdf 中记录了 DWARF3。本节通过指定 C6000 特定表示部分来扩充该标准。

12.1 DWARF 寄存器名称

DWARF3 寄存器使用寄存器名称运算符（请参阅 DWARF3 标准的第 2.6.1 节）。寄存器名称运算符的操作数是表示结构寄存器的寄存器编号。 表 12-1 定义了从 DWARF3 寄存器编号/名称到 C6000 寄存器的映射。

表 12-1. C6000 的 DWARF3 寄存器编号

DWARF 名称	C6000 ISA 寄存器	说明
0 – 15	A0-A15	
16-31	B0-B15	
32	被保留	
33	PCE1	E1 阶段程序计数器
34	IRP	中断返回指针寄存器
35	IFR	中断标志寄存器
36	NRP	NMI 返回指针寄存器
37-52	A16-A31	
53-68	B16-B31	
69	AMR	地址模式寄存器
70	CST	控制状态寄存器
71	ISR	中断集寄存器
72	ICR	中断清除寄存器
73	IER	中断启用寄存器
74	ISTP	中断服务表指针寄存器
75	IN	无文档记载的控制寄存器
76	OUT	无文档记载的控制寄存器
77	ACR	无文档记载的控制寄存器
78	ADR	无文档记载的控制寄存器
79	FADCR	浮点加法器配置寄存器
80	FAUCR	浮点辅助配置寄存器
81	FMCR	浮点乘法器配置寄存器
82	GFPGFR	伽罗瓦域多项式发生器函数寄存器
83	DIER	无文档记载的控制寄存器
84	REP	受限入口点寄存器
85	TSCL	时间戳计数器 - 低半部分
86	TSCH	时间戳计数器 - 高半部分
87	ARP	无文档记载的控制寄存器
88	ILC	SPLOOP 内部环路计数寄存器
89	RILC	SPLOOP 重新加载内部环路计数寄存器
90	DNUM	DSP 内核数寄存器
91	SSR	饱和状态寄存器
92	GPLYA	GMPY 多项式 - A 侧寄存器
93	GPLYB	GMPY 多项式 - B 侧寄存器
94	TSR	任务状态寄存器
95	ITSR	中断任务状态寄存器
96	NTSR	NMI/异常任务状态寄存器
97	EFR	异常标志寄存器
98	ECR	异常清除寄存器
99	IERR	内部异常报告寄存器
100	DMSG	无文档记载的控制寄存器

表 12-1. C6000 的 DWARF3 寄存器编号（续）

DWARF 名称	C6000 ISA 寄存器	说明
101	CMSG	无文档记载的控制寄存器
102	DT_DMA_ADDR	无文档记载的控制寄存器
103	DT_DMA_DATA	无文档记载的控制寄存器
104	DT_DMA_CNTL	无文档记载的控制寄存器
105	TCU_CNTL	无文档记载的控制寄存器
106	RTDX_REC_CNTL	无文档记载的控制寄存器
107	RTDX_XMT_CNTL	无文档记载的控制寄存器
108	RTDX_CFG	无文档记载的控制寄存器
109	RTDX_RDATA	无文档记载的控制寄存器
110	RTDX_WDATA	无文档记载的控制寄存器
111	RTDX_RADDR	无文档记载的控制寄存器
112	RTDX_WADDR	无文档记载的控制寄存器
113	MFREG0	无文档记载的控制寄存器
114	DBG_STAT	无文档记载的控制寄存器
115	BRK_EN	无文档记载的控制寄存器
116	HWBP0_CNT	无文档记载的控制寄存器
117	HWBP0	无文档记载的控制寄存器
118	HWBP1	无文档记载的控制寄存器
119	HWBP2	无文档记载的控制寄存器
120	HWBP3	无文档记载的控制寄存器
121	覆盖层	无文档记载的控制寄存器
122	PC_PROF	无文档记载的控制寄存器
123	ATSR	无文档记载的控制寄存器
124	TRR	无文档记载的控制寄存器
125	TCRR	无文档记载的控制寄存器
126	DESR	无文档记载的控制寄存器
127	DETR	无文档记载的控制寄存器
128	STRM_HOLD	无文档记载的控制寄存器
129	PDATA_O	无文档记载的控制寄存器
130	TCR	无文档记载的控制寄存器

12.2 调用帧信息

调试器需要能够在函数执行过程中查看和修改任何函数的局部变量。

DWARF3 通过让编译器跟踪函数存储数据的位置（在寄存器或堆栈中）来实现这一点。编译器使用 DWARF3 标准第 6.4 节中指定的字节编码语言对这些信息进行编码。这使得调试器可以通过解释字节编码语言来逐渐重新创建以前的状态。每个函数激活由一个基址（称为规范栈帧地址 (Canonical Frame address, CFA)）以及一组与激活期间机器寄存器内容相对应的值表示。只要提供激活执行进行到的点，调试器就可以找出所有函数数据所在的位置，并将堆栈展开到先前的状态，包括先前的函数激活。

DWARF3 标准建议使用一个非常大的展开表，每个代码地址一行，每个寄存器一列（虚拟或非虚拟，包括 CFA）。每节单元格包含该寄存器在该时间点（代码地址）的展开指令。

CFA 的定义和构成状态的寄存器组都特定于架构。

寄存器组包括 表 12-1 中列出的所有寄存器，按第一列中的 DWARF 寄存器编号索引。

对于 CFA，C6000 ABI 遵循 DWARF3 标准中建议的约定，将其定义为（调用过程的）先前帧中的调用点处 SP 的值 (B15)。

如 DWARF3 标准的第 6.4.4 节中所建议那样，展开表中没有针对虚拟返回地址的不同列。根据调用约定，返回地址由展开表的 B3 列表示。

展开表可能包括并非所有 C6000 ISA 上都存在的寄存器。因此，可能会出现这样的情况：执行程序的 ISA 具有调用帧信息中未提到的寄存器。在这种情况下，解释器的行为方式应该如下：

- 被调用者保存的寄存器应初始化为相同值规则。
- 所有其他寄存器应初始化为未定义的规则。

12.3 供应商名称

DW_AT_producer 属性用于标识生成目标文件的工具链。操作数是以供应商前缀开头的字符串。以下前缀为特定供应商而保留：

TI	C6000 德州仪器 (TI) 的代码生成工具
GNU	GNU 编译器套装 (GCC)

12.4 供应商扩展

DWARF 标准允许工具链供应商定义一些附加标签和属性，用于表示架构或工具链特定的信息。TI 定义了其中一些。本节用于记录通常适用于 C6000 架构的标签和属性。

遗憾的是，所有供应商共享一组允许值，因此 ABI 不能强制在供应商之间使用标准值。我们只能让生产者使用相同的语义定义他们自己的供应商特定标签和属性（如果可能，使用相同的值），并且请消费者使用 DW_AT_producer 属性，以便解释因工具链而异的供应商特定值。

表 12-2 为 C6000 定义了 TI 的供应商特定 DIE 标签。表 12-2 定义了 TI 的供应商特定属性。

表 12-2. TI 的供应商特定标签

名称	值	说明
DW_TAG_TI_branch	0x4088	标识调用和返回

DW_TAG_TI_branch

此标记标识用作调用和返回的分支。它作为 DW_TAG_subprogram DIE 的子项生成。它具有一个与分支指令的位置对应的 DW_AT_lowpc 属性。

如果分支是函数调用，则它具有一个非零值 DW_AT_TI_call 属性。它还可能具有 DW_AT_name 属性，用于指示被调用函数的名称；或者，如果被调用者未知（就像通过指针调用一样），则具有 DW_AT_TI_indirect 属性。

如果分支是返回，则它具有一个非零值 DW_AT_TI_return 属性。

表 12-3. TI 的供应商特定属性

名称	值	类	说明
DW_AT_TI_symbol_name	0x2001	string	目标文件名（已改编）
DW_AT_TI_return	0x2009	标志[flag]	分支是返回
DW_AT_TI_call	0x200A	标志[flag]	分支是调用
DW_AT_TI_asm	0x200C	标志[flag]	函数为汇编语言
DW_AT_TI_indirect	0x200D	标志[flag]	分支是间接调用
DW_AT_TI_plt_entry	0x2012	标志[flag]	函数是一个 PLT 条目
DW_AT_TI_max_frame_size	0x2014	常量	激活记录大小

DW_AT_TI_call、**DW_AT_TI_return**、**DW_AT_TI_indirect**：如前所述，这些属性适用于 DW_TAG_TI_branch DIE。

DW_AT_TI_symbol_name：此属性可能出现在任何具有 DW_symbol_name 的 DIE 中。它提供与变量或函数关联的目标文件级名称；即具有由工具链对源代码级别名称所应用的任何改编或其他修改。

DW_AT_TI_plt_entry：此属性（具有非零值）添加到与过程链接表条目相对应的 DW_TAG_subprogram DIE。它的含义与 DW_AT_trampoline 类似。

DW_AT_TI_max_frame_size：此属性可能出现在 DW_TAG_subprogram DIE 中。它以字节为单位，指示函数激活所需的栈空间量。它的预期用途是用于执行静态栈深度分析的下游工具。

章节 13

ELF 目标文件（处理器补充）

C6000 ABI 基于 ELF 目标文件格式。ELF 的基本规范由大型 System V ABI 规范 (http://www.sco.com/developers/gabi/2003-12-17/contents.html) 的第 4 章和第 5 章组成。

下面的小节包含针对规范第 4 章（目标文件）的 C6000 处理器特定补充。本文档的 章节 14 包含针对规范第 5 章（程序加载和动态链接）的处理器特定补充。

13.1 注册供应商名称

编译器工具集可创建和使用特定于供应商的符号。为了避免潜在冲突，TI 鼓励供应商定义和使用特定于供应商的命名空间。表 13-1 列出了当前注册供应商及其首选的简写名称。

表 13-1. 注册供应商

名称	供应商
cxa，_ _cxa	C++ ABI 命名空间。适用于 C++ ABI 指定的所有符号。
c6xabi，_ _c6xabi	适用于 C6000 EABI 指定的符号的通用命名空间。
C6000	适用于 C6000 指定的符号的通用命名空间。
TI，_ _TI	为特定于 TI 工具链的符号保留。这也代表了所有 TI 处理器 ABI 的复合命名空间。
gnu，_ _gnu	为特定于 GCC 工具链的符号保留。

备注

TI 或 _ _TI 规范定义了特定于处理器的段类型、特殊段等的名称。如果不同 TI 处理器之间存在共性，则此类实体使用 TI 命名，而不是为每个处理器定义不同的名称。例如，对于所有 TI 处理器，Exception Table Index Table 段类型为 SHT_TI_EXIDX，而不是 C6000 的段类型为 SHT_C6000_EXIDX、C2000 的段类型为 SHT_C2000_EXIDX 等。

13.2 ELF 标头

ELF 标头提供了许多用于指导文件解释的字段，其中大部分都在 System V ELF 规范中指定。本节使用 C6000 的特定详细信息来扩充基本标准。

e_indent

16 字节 ELF 标识字段将文件标识为目标文件，并提供与机器无关的数据，用于解码和解释文件的内容。表 13-2 指定了将用于 C6000 目标文件的值。

表 13-2. ELF 标识字段

索引	符号值	数值	说明
EI_MAG0		0x7f	根据 System V ABI
EI_MAG1		E	根据 System V ABI
EI_MAG2		L	根据 System V ABI
EI_MAG3		F	根据 System V ABI
EI_CLASS	ELFCLASS32	1	32 位 ELF
EI_DATA	ELFDATA2LSB	1	小端字节序
	ELFDATA2MSB	2	大端字节序
EI_VERSION	EV_CURRENT	1	
EI_OSABI	ELFOSABI_C6000_ELFABI	64	裸机动态链接平台
EI_OSABI	ELFOSABI_C6000_LINUX	65	无 MMU 的 Linux 平台
EI_ABIVERSION		0	

EI_OSABI 字段应为 ELFOSABI_NONE，除非由特定平台的约定覆盖。裸机动态链接模型 (节 14.4) 和 Linux (章节 15) 是两个定义该字段特定值的平台。

ELFOSABI_NONE 以外的值表示断言：该文件符合对应于指定值的特定 ABI 变体的约定。仅此类文件对该特定平台有效。可以为 ABI 定义的特定变体以外的平台构建对象；这些对象应标识为 ELFOSABI_NONE，表示没有任何断言。确定此类文件是否与给定环境兼容时，不受 ABI 影响。

e_type

当前无 C6000 特定目标文件类型。保留 ET_LOPROC 和 ET_HIPROC 之间的所有值，以在本规范的未来修订版中使用。

e_machine

符合本规范的目标文件必须具有值 EM_TI_C6000 (140，0x8c) 。

e_entry

如果应用程序没有入口点，则基本 ELF 规范要求该字段为零。尽管如此，某些应用程序可能需要零入口点 (例如，通过复位向量) 。

平台标准可指定可执行文件始终具有入口点，在这种情况下，e_entry 指定该入口点，即使该入口点为零。

e_flags

该成员保存与文件相关的处理器特定标志，有一个 C6000 特定标志。

名称	值	注释
EF_C6000_REL	0x1	文件包含静态重定位信息

EF_C6000_REL 标志用于指示可执行文件 (ET_EXEC) 或共享对象 (ET_DYN) 中是否存在静态重定位信息。具有静态重定位信息的共享对象称为*可重定位模块*，通常用于可静态或动态链接的库。

13.3 段

未定义处理器特定的特殊段索引。该规范的未来修订版本保留所有处理器特定值。

13.3.1 段索引

ABI 定义一个特殊段索引：

名称	值	注释
SHN_C6000_SCOMMON	0xFF00	支持 near DP 寻址的通用块符号

SHN_C6000_SCOMMON 索引可以标识使用 near DP 寻址进行寻址的通用块符号；请参阅节 13.4.2。

13.3.2 段类型

ELF 规范为处理器特定的值保留了段类型 0x70000000 及更高的段。TI 将该范围分为了两部分：0x70000000 到 0x7EFFFFFF 的值为处理器特定值，而 0x7F000000 到 0xFFFFFFFF 的值对应多个 TI 架构共用的 TI 特定段。表 13-3 中列出了组合集。

并非所有这些段类型都在 C6000 ABI 中使用。一些特定于 TI 工具链，但超出 ABI；一些则由 TI 工具链用于除 C6000 之外的架构。本文档对其进行了介绍以保证完整性，同时保留标记值。

表 13-3. ELF 和 TI 段类型

名称	值	注释
SHT_C6000_UNWIND	0x70000001	用于栈回溯的回溯函数表
SHT_C6000_PREEMPTMAP	0x70000002	DLL 动态链接抢占映射
SHT_C6000_ATTRIBUTES	0x70000003	目标文件兼容性属性
SHT_TI_ICODE	0x7F000000	用于链接时优化的中间代码
SHT_TI_XREF	0x7F000001	符号交叉参考信息
SHT_TI_HANDLER	0x7F000002	保留
SHT_TI_INITINFO	0x7F000003	用于初始化 C 变量的压缩数据
SHT_TI_PHATTRS	0x7F000004	扩展程序标头属性
SHT_TI_SH_FLAGS	0x7F000005	扩展段标头属性
SHT_TI_SYMALIAS	0x7F000006	符号别名表
SHT_TI_SH_PAGE	0x7F000007	每段存储器空间表

SHT_C6000_UNWIND 识别包含用于栈回溯的回溯函数表的段。有关详细信息，请参阅章节 11。

SHT_C6000_PREMPTMAP 识别包含 C6000 DLL 动态链接抢占映射的段。

SHT_C6000_ATTRIBUTES 识别包含对象兼容性属性的段。请参阅章节 17。

SHT_TI_ICODE 识别包含 TI 特定源代码中间表示的段，该代码用于链接时重新编译和优化。

SHT_TI_XREF 识别包含符号交叉参考信息的段。

SHT_TI_HANDLER 当前未使用。

SHT_TI_INSTINFO 识别包含用于初始化 C 变量的压缩数据的段。此段包含一个指示源地址和目标地址的记录表，以及通常为压缩格式的数据本身。请参阅章节 18。

SHT_TI_PHATTRS 识别包含可执行文件或共享目标文件中程序段的附加属性的段。请参阅章节 19。

SHT_TI_SH_FLAGS 识别包含 TI 特定段标头标志表的段。

SHT_TI_SYMALIAS 识别包含一个用于将符号定义为等同于其他符号 (可能是外部定义的符号) 的表的段。TI 链接器使用该表来消除仅转发给其他函数的平凡函数。

SHT_TI_SH_PAGE 仅在具有不同 (可能重叠) 地址空间 (页) 的目标上使用。段中包含一个将其他段与页码相关联的表。此段类型不在 C6000 上使用。

13.3.3 扩展段标头属性

未定义处理器特定段属性标志。该规范的未来修订版本保留所有处理器特定值。程序标头属性如 章节 19 中所述。

13.3.4 子段

C6000 目标文件采用一种段命名约定来提供更高的粒度，同时保留在链接时段合并默认规则的便利性。名称中包含冒号的段称为*子段*。子段在各方面与普通段相同，但在将段组合到输出文件中时，子段的名称会引导链接器。子段的根名是指一直到冒号 (但不包括冒号) 的名称。后缀包括冒号后的所有字符。默认情况下，链接器会将所有具有匹配根的段全部组合成使用该名称的单个段。例如，.text、text:func1 和 .text:func2 会组合成名为 .text 的单个段。用户可以通过工具链特定的方式覆盖此默认行为。

如果有多个冒号，则段组合过程将从最右边的冒号开始以递归方式进行。例如，除非用户另外指定，否则默认规则将组合 .bss:func1:var1 和 .bss:func1:var2，然后组合成 .bss。

如节 13.3.5 中的定义，对于根名与特殊段匹配的子段，它们的 ABI 定义属性与匹配段相同。例如，.text:func1 是 .text 段的实例。

13.3.5 特殊段

System V ABI 以及此 ABI 的其他基础文档和其他部分，定义了几个具有专门用途的段。表 13-4 整合了 C6000 使用的一些专用段，并且按功能进行了分组。

ABI 不强制要求具有段名。特殊段应按类型而不是名称进行标识。但是，通过遵循这些约定可以提高工具链之间的互操作性。例如，有时需要编写自定义链接器命令来链接由不同编译器构建的可重定位文件，而使用这些名称可以降低这样做的可能性。

ABI 强制要求名称与表中条目匹配的段必须用于指定用途。例如，编译器不需要将代码生成到名为 .text 的段中，但不允许在生成的名为 .text 的段中包含除代码以外的任何内容。

下表中列出的所有段名都是前缀。类型和属性会应用于名称以这些字符串开头的所有段。

表 13-4. C6000 特殊段

前缀	类型	属性
	代码段	
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.plt	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
	Near 数据段	
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.neardata	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.rodata	SHT_PROGBITS	SHF_ALLOC
	Far 数据段	
.far	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.fardata	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.const	SHT_PROGBITS	SHF_ALLOC
.fardata:const	SHT_PROGBITS	SHF_ALLOC
	动态数据段	
.got	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.dsbt	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
	异常处理数据段	
.C6000.exidx	SHT_C6000_UNWIND	SHF_ALLOC + SHF_LINK_ORDER
.C6000.extab	SHT_PROGBITS	SHF_ALLOC

表 13-4. C6000 特殊段（续）

前缀	类型	属性
	初始化和终止段	
.init	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.fini	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.preinit_array	SHT_PREINIT_ARRAY	SHF_ALLOC + SHF_WRITE
.init_array	SHT_INIT_ARRAY	SHF_ALLOC + SHF_WRITE
.fini_array	SHT_FINI_ARRAY	SHF_ALLOC + SHF_WRITE
	ELF 结构	
.rel	SHT_REL	无
.rela	SHT_RELA	无
.symtab	SHT_SYMTAB	无
.symtab_shndx	SHT_SYMTAB_SHNDX	无
.strtab	SHT_STRTAB	SHF_STRINGS
.shstrtab	SHT_STRTAB	SHF_STRINGS
.note	SHT_NOTE	无
	动态加载结构	
.dynamic (1)	SHT_DYNAMIC	SHF_ALLOC
.dynsym (1)	SHT_DYNSYM	SHF_ALLOC
.dynstr (1)	SHT_STRTAB	SHF_ALLOC + SHF_STRINGS
.hash (1)	SHT_TAB	SHF_ALLOC
.interp	SHT_PROGBITS	无
	构建属性	
.C6000.attributes	SHT_C6000_ATTRIBUTES	无
	符号调试段	
.debug (2)	SHT_PROGBITS	无
	符号版本控制段(1)	
.gnu.version	SHT_GNU_versym	SHF_ALLOC
.gnu.version_d	SHT_GNU_verdef	SHF_ALLOC
.gnu.version_r	SHT_GNU_verneed	SHF_ALLOC
	线程局部存储段	
.tbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.tdata	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.tdata1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + SHF_TLS
.TI.tls_init	SHT_PROGBITS	SHF_ALLOC
	TI 工具链特定段	
.stack	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.sysmem	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.cio	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.switch	SHT_PROGBITS	SHF_ALLOC
.binit	SHT_PROGBITS	SHF_ALLOC
.cinit	SHT_TI_INITINFO	SHF_ALLOC
.const:handler_table	SHT_PROGBITS	SHF_ALLOC
.ovly	SHT_PROGBITS	SHF_ALLOC
.ppdata	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.ppinfo	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.TI.crctab	SHT_PROGBITS	SHF_ALLOC
.TI.icode	SHT_TI_ICODE	无

表 13-4. C6000 特殊段（续）

前缀	类型	属性
.TI.phattrs	SHT_TI_PHATTRS	无
.TI.preempt.map	SHT_C6000_PREEMPTMAP	SHF_ALLOC
.TI.xref	SHT_TI_XREF	无
.TI.section.flags	SHT_TI_SH_FLAGS	无
.TI.symbol.alias	SHT_TI_SYMALIAS	无
.TI.section.page	SHT_TI_SH_PAGE	无
位于 System V ABI 中但未被 C6000 EABI 使用的段		
.comment		
.data		
.data1		
.line		
.rodata1		

(1) .dynamic 段和相关段是否分配到存储器中因平台而异。
(2) 此外还使用名称为 .debug_info 和 .debug_line 的其他段。与其他段名称一样，.debug 段名称是前缀。类型和属性会应用于名称以 .debug 开头的所有段。

上表中的“**TI 工具链特定段**”由 TI 工具链以各种工具链特定的方式使用。ABI 不强制要求使用这些段（但使用它们可促进互操作性），但它确实会保留这些名称。

上表中的“**位于 System V ABI 中但未被 C6000 EABI 使用的段**”由 System V ABI 指定，但不在 C6000 ABI 下使用或定义。TI 将其他段用于其他器件；这些名称予以保留。

有关线程局部存储的详细信息，请参阅章节 7。

此外，.common 和 .scommon 是链接器使用的段名称。这些是抽象段，不是目标文件中的实际段。这些名称是链接器命令文件中用于放置变量的约定。这些段不应用于其他目的。

13.3.6 段对齐

包含 C6000 代码的段必须至少为 32 字节对齐，并填充到 32 字节边界。后一项要求是为了避免在 C64+ 及后续架构上，将相邻数据误解为取指数据包标头。

平台标准可能会对它们能够保证的最大对齐量设置一定的限制（通常为虚拟存储器页大小）。

13.4 符号表

没有处理器特定的符号类型或符号绑定。该规范的未来修订版本保留所有处理器特定值。

在全局和弱符号定义以及符号值的含义方面，C6000 ABI 遵循 ELF 规范。

13.4.1 符号类型

此规范遵循与符号类型相关的 ARM ELF 规范，即：

- 从目标文件导出的所有代码符号 (具有绑定 STB_GLOBAL 的符号) 都应具有 STT_FUNC 类型。
- 所有外部数据对象都应具有 STT_OBJECT 类型。任何 STB_GLOBAL 数据符号都不应具有 STT_FUNC 类型。
- 未定义符号的类型应为 STT_NOTYPE 或其预期定义的类型。
- 其他在可执行文件段中定义的符号的类型可以是 STT_NOTYPE。

此外，线程局部符号的符号类型为 STT_TLS。

13.4.2 通用块符号

如 ELF 规范中所述，类型为 STT_COMMON 的符号由链接器分配。 C6000 ABI 扩展了通用块机制以适应 near 和 far 数据寻址。如果使用相对于 DP 的 near 寻址来寻找通用块符号的地址，则该符号必须将特定于处理器的值 SHN_C6000_SCOMMON 作为其节索引。链接器将此类符号分配到 near 数据节，通常为 .bss。

如基本 ELF 规范中所述，用其他寻址形式寻址的通用块符号应具有节索引 SHN_COMMON。此类符号可以分配到 far 数据节中，通常为 .far。

13.4.3 符号名称

用于对 C 或汇编语言实体进行命名的符号应具有该实体的名称。例如，名为 *func* 的 C 函数会生成名为 *func* 的符号。 (与之前的 COFF ABI 不同，没有前导下划线)。符号名称区分大小写，并由链接器精确匹配。

C6000 编译器遵循以下临时符号命名约定：

- 解析器生成的符号带有 P 前缀
- 优化器生成的符号带有 O 前缀
- 代码生成的符号带有 C 前缀

13.4.4 保留符号名称

本规范的本次修订版和将来修订版保留以下符号：

- 以 $ 开头的局部符号 (STB_LOCAL)
- 以表 13-1 中列出的任何供应商名称开头的全局符号 (STB_GLOBAL、STB_WEAK)。
- 以 $$Base 或 $$Limit 中的任何一个结尾的全局符号 (STB_GLOBAL、STB_WEAK)
- 与模式 ${Tramp}${I|L|S}[$PI]$$symbol 匹配的符号
- 编译器生成的以 P、O、C 开头的临时符号 (如节 4.5 中所述)

13.4.5 映射符号

映射符号是用于对程序数据进行分类的局部符号。目前，ABI 未指定任何使用映射符号的行为。不过，保留了以下两个名称以供将来使用：$code 和 $data。

13.5 重定位

C6000 的 ELF 重定位经过了专门定义，以便将与执行重定位所需的所有信息包含在重定位条目、对象字段和关联符号中。除了解压对象字段之外，链接器不需要解码指令来执行重定位。这导致重定位类型略多于较旧的 C6000 COFF ABI。COFF 与 ELF 之间的重定位类型不兼容。

重定位指定为对可重定位字段进行操作。大致来说，可重定位字段是受重定位影响的程序映像位。此字段根据可寻址容器定义，其地址由重定位条目的 r_offset 字段提供。此字段在容器中的大小和位置以及重定位值的计算由重定位类型指定。重定位操作包括提取可重定位的字段、执行操作和将结果值重新插入此字段。

ELF 重定位可以是 Elf32_Rela 或 Elf32_Rel 类型。Rela 条目包含用于重定位计算的显式加数。Rel 类型的条目使用可重定位字段本身作为加数。某些重定位仅标识为 Rela。大多数情况下，这些重定位对应于 32 位地址的高 16 位，其中结果值取决于来自此字段中不可用低位的进位传播。如果指定了 Rela，则实现必须遵守此要求。实现可选择将 Rel 或 Rela 类型重定位用于其他重定位。

13.5.1 重定位类型

重定位类型在两个表中进行了说明。 表 13-5 提供了重定位类型的数值，并汇总了重定位值的计算。此表之后描述了重定位类型及其使用示例。 表 13-6 描述了每种类型的确切计算，包括重定位字段的提取和插入、溢出检查以及任何缩放或其他调整。

表 13-5 中使用了以下表示法。

S	与重定位关联的符号的值，由重定位条目的 r_info 字段中包含的符号表索引指定。
A	用于计算可重定位字段的值的加数。对于 ELF32_rel 重定位，A 根据 表 13-6 编码到可重定位字段中。对于 Elf32_Rela 重定位，A 由重定位条目的 r_addend 字段提供。
PC	包含字段的容器的地址。
FP(x)	包含地址 x 处指令的取指数据包的地址；即：FP(x) := x & 0xFFFFFFE0。
P	被重定位的指令的取指数据包地址；即：P := FP(PC)。
B	当前加载模块的数据段的基址。此位置由符号 _ _C6000_DSBT_BASE 标记，在执行程序时是 DP 寄存器的值。
GOT(S)	重定位相关符号 (S) 的全局偏移表 (GOT) 条目的地址。
TBR(x)	x 相对于线程局部存储 (TLS) 块基址的偏移量。有关线程局部存储的详细信息，请参阅章节 7。
TPR(x)	x 相对于线程指针 (TP) 的偏移量。
TLS(x)	x 的 TLS 描述符，其中包含 x 的模块 ID 和 TBR 偏移量。
TLSMOD(x)	定义 x 的模块的 TLS 模块标识符。

表 13-5. C6000 重定位类型

名称	值	运算	约束条件
R_C6000_NONE	0		
R_C6000_ABS32	1	S + A	
R_C6000_ABS16	2	S + A	
R_C6000_ABS8	3	S + A	
R_C6000_PCR_S21	4	S + A – P	
R_C6000_PCR_S12	5	S + A – P	
R_C6000_PCR_S10	6	S + A – P	
R_C6000_PCR_S7	7	S + A – P	
R_C6000_ABS_S16	8	S + A	
R_C6000_ABS_L16	9	S + A	
R_C6000_ABS_H16	10	S + A	仅限 Rela
R_C6000_SBR_U15_B	11	S + A – B	
R_C6000_SBR_U15_H	12	S + A – B	
R_C6000_SBR_U15_W	13	S + A – B	

表 13-5. C6000 重定位类型（续）

名称	值	运算	约束条件
R_C6000_SBR_S16	14	S + A – B	
R_C6000_SBR_L16_B	15	S + A – B	
R_C6000_SBR_L16_H	16	S + A – B	
R_C6000_SBR_L16_W	17	S + A – B	
R_C6000_SBR_H16_B	18	S + A – B	仅限 Rela
R_C6000_SBR_H16_H	19	S + A – B	仅限 Rela
R_C6000_SBR_H16_W	20	S + A – B	仅限 Rela
R_C6000_SBR_GOT_U15_W	21	GOT(S) + A – B	
R_C6000_SBR_GOT_L16_W	22	GOT(S) + A – B	
R_C6000_SBR_GOT_H16_W	23	GOT(S) + A – B	仅限 Rela
R_C6000_DSBT_INDEX	24	此静态链路单元的 DSBT 索引	
R_C6000_PREL31	25	S + A – PC	
R_C6000_COPY	26	被抢占符号的加载时副本	仅限 ET_EXEC
R_C6000_JUMP_SLOT	27	S + A	ET_EXEC/ET_DYN
R_C6000_EHTYPE	28	S + A - B	
R_C6000_PCR_H16	29	S - FP(P - A)	仅限 Rela
R_C6000_PCR_L16	30	S - FP(P - A)	仅限 Rela
保留	31		
保留	32		
R_C6000_TBR_U15_B	33	TBR(S)	仅限静态
R_C6000_TBR_U15_H	34	TBR(S)	仅限静态
R_C6000_TBR_U15_W	35	TBR(S)	仅限静态
R_C6000_TBR_U15_D	36	TBR(S)	仅限静态
R_C6000_TPR_S16	37	TBR(S)	
R_C6000_TPR_U15_B	38	TPR(S)	
R_C6000_TPR_U15_H	39	TPR(S)	
R_C6000_TPR_U15_W	40	TPR(S)	
R_C6000_TPR_U15_D	41	TPR(S)	
R_C6000_TPR_U32_B	42	TPR(S)	仅限动态
R_C6000_TPR_U32_H	43	TPR(S)	仅限动态
R_C6000_TPR_U32_W	44	TPR(S)	仅限动态
R_C6000_TPR_U32_D	45	TPR(S)	仅限动态
R_C6000_SBR_GOT_U15_W_TLSMOD	46	GOT(TLSMOD(S)) + A – B	仅限静态
R_C6000_SBR_GOT_U15_W_TBR	47	GOT(TBR(S)) + A - B	仅限静态
R_C6000_SBR_GOT_U15_W_TPR_B	48	GOT(TPR(S))+A-B	仅限静态
R_C6000_SBR_GOT_U15_W_TPR_H	49	GOT(TPR(S))+A-B	仅限静态
R_C6000_SBR_GOT_U15_W_TPR_W	50	GOT(TPR(S))+A-B	仅限静态
R_C6000_SBR_GOT_U15_W_TPR_D	51	GOT(TPR(S))+A-B	仅限静态
R_C6000_SBR_GOT_L16_W_TLSMOD	52	GOT(TLSMOD(S)) + A - B	仅限静态
R_C6000_SBR_GOT_L16_W_TBR	53	GOT(TBR(S)) + A - B	仅限静态
R_C6000_SBR_GOT_L16_W_TPR_B	54	GOT(TPR(S))+A-B	仅限静态
R_C6000_SBR_GOT_L16_W_TPR_H	55	GOT(TPR(S))+A-B	仅限静态
R_C6000_SBR_GOT_L16_W_TPR_W	56	GOT(TPR(S))+A-B	仅限静态
R_C6000_SBR_GOT_L16_W_TPR_D	57	GOT(TPR(S))+A-B	仅限静态
R_C6000_SBR_GOT_H16_W_TLSMOD	58	GOT(TLSMOD(S)) + A - B	仅限静态

表 13-5. C6000 重定位类型 (续)

名称	值	运算	约束条件
R_C6000_SBR_GOT_H16_W_TBR	59	GOT(TBR(S)) + A - B	仅限静态
R_C6000_SBR_GOT_H16_W_TPR_B	60	GOT(TPR(S))+A-B	仅限静态
R_C6000_SBR_GOT_H16_W_TPR_H	61	GOT(TPR(S))+A-B	仅限静态
R_C6000_SBR_GOT_H16_W_TPR_W	62	GOT(TPR(S))+A-B	仅限静态
R_C6000_SBR_GOT_H16_W_TPR_D	63	GOT(TPR(S))+A-B	仅限静态
R_C6000_TLSMOD	64	TLSMOD(S)	仅限动态
R_C6000_TBR_U32	65	TBR(S)	仅限动态
R_C6000_ALIGN	253	无	仅限 ET_REL
R_C6000_FPHEAD	254	无	仅限 ET_REL
R_C6000_NOCMP	255	无	仅限 ET_REL

R_none 重定位不执行任何运算。它用于创建一个段对另一段的引用，以确保如果引用段链接进来，被引用的段也会链接进来。

R_C6000_ABS8/16/32 重定位直接将符号的重定位地址编码为 8 位、16 位或 32 位字段，这些字段通常用于初始化数据，而非指令。字段的符号未指定；也就是说，这些字段既可用于有符号值，也可用于无符号值。

```
        .field X,32         ; R_C6000_ABS32
        .field X,16         ; R_C6000_ABS16
        .field X,8          ; R_C6000_ABS8
```

PCR 重定位对有符号的 PC 相对分支位移进行编码。这些位移缩放为 32 位 (字) 单元。位移是相对于源指令的取指数据包计算的。

```
        B       func        ; R_C6000_PCR_S21
        CALLP   func,B3     ; R_C6000_PCR_S21
        BNOP    func        ; R_C6000_PCR_S12
        BPOS    func,A10    ; R_C6000_PCR_S10
        BDEC    func,A1     ; R_C6000_PCR_S10
        ADDKPC  func,B3,4   ; R_C6000_PCR_S7
```

名称中包含 L16 的重定位对 32 位地址或偏移量的低 16 位进行编码。包含 H16 的重定位对高 16 位进行编码，并且始终为 Rela。包含 S16 的重定位对有符号的 16 位值 (通常不是地址的一部分) 进行编码。包含 U15 的重定位对无符号的 15 位 DP 相对位移进行编码。

```
        MVHL    sym,A0      ; R_C6000_ABS_L16
        MVKH    sym,A0      ; R_C6000_ABS_H16
        MVK     const16,A0  ; R_C6000_ABS_S16  sign extend const16 into A0
        MVKLH   const16,A0  ; R_C6000_ABS_L16  move const16 into A0[16:31]
```

PCR_L16 和 PCR_H16 重定位分别对目标地址与参考 PC (“基础 PC”) 的取指数据包地址之间的 PC 相对偏移量的低位和高位进行编码。从当前指令的取指数据包到基础 PC 的偏移量编码到加数字段中；即 A := (P-base)。重定位随后计算 S-FP(P-A)，得出 S 与 FP(base) 之间的偏移量。这些重定位用于使用 PC 相对寻址来寻址不同段中的对象，如节 5.1 所述。

```
        MVK     $PCR_OFFSET(sym,base),A0    ; R_C6000_PCR_L16
        MVKH    $PCR_OFFSET(sym,base),A0    ; R_C6000_PCR_H16
```

SBR_U15 重定位对 15 位无符号 DP 相对偏移量进行编码，以实现 near DP 数据寻址。重定位根据访问宽度进行缩放：32 位字 (_W)、16 位半字 (_H) 或字节 (_B)。

```
        LDB     *+DP(sym),A1 ; R_C6000_SBR_U15_B
        ADDAB   DP,sym,A2    ; R_C6000_SBR_U15_B
        LDH     *+DP(sym),A1 ; R_C6000_SBR_U15_H
        ADDAH   DP,sym,A2    ; R_C6000_SBR_U15_H
```

```
        LDW     *+DP(sym),A1 ; R_C6000_SBR_U15_W
        ADDAW   DP,sym,A2    ; R_C6000_SBR_U15_W
```

其他 SBR 重定位用于对 32 位 DP 相对偏移量的高位和低位部分进行编码，以实现 far DP 相对寻址。在以下示例中：

- $bss 表示数据段基址，对应于 _ _C6000_DSBT_BASE (DP 中的值)
- $DPR_byte(sym) 表示 DP 相对偏移量 (以字节为单位)
- $DPR_hword(sym) 表示 DP 相对偏移量除以 2
- $DPR_word(sym) 表示 DP 相对偏移量除以 4

```
        MVK     (sym - $bss),A0     ; R_C6000_SBR_S16
        MVKL    $DPR_byte(sym),A0   ; R_C6000_SBR_L16_B
        MVKH    $DPR_byte(sym),A0   ; R_C6000_SBR_H16_B
        MVKL    $DPR_hword(sym),A0  ; R_C6000_SBR_L16_H
        MVKH    $DPR_hword(sym),A0  ; R_C6000_SBR_H16_H
        MVKL    $DPR_word(sym),A0   ; R_C6000_SBR_L16_W
        MVKH    $DPR_word(sym),A0   ; R_C6000_SBR_H16_W
```

SBR_GOT 重定位对应于与 SBR 重定位相同的指令和编码，但引用的是被引用符号 (而非符号本身) 的 DP 相对 GOT 地址。通常，GOT 通过 near DP 相对寻址访问，因此使用 R_C6000_DBR_GOT_U15_W。当 GOT 为 far 时，偏移量通过包含其他两个重定位的 MVKL/MVKH 生成 (请参阅节 6.6)。在以下示例中，

- GOT(sym) 是 sym 的 GOT 条目的 DP 相对偏移量 (以字节为单位)
- $DPR_GOT(sym) 是 sym 的 GOT 条目的 DP 相对偏移量 (以字节为单位)

```
        LDW     *+DP[GOT(sym)],A0   ; R_C6000_SBR_GOT_U15_W
        MVKL    $DPR_GOT(sym), A0   ; R_C6000_SBR_GOT_L16_W
        MVKH    $DPR_GOT(sym), A0   ; R_C6000_SBR_GOT_H16_W
```

R_C6000_DSBT_INDEX 将索引编码到当前加载模块的数据段基表中。它仅存在于使用 DSBT 模型实现位置无关性的文件中。请参阅节 6.7。

```
        LDW     *+DP($DSBT_INDEX(__C6000_DSBT_BASE)),DP   ; R_C6000_DSBT_INDEX
```

R_C6000_COPY 用于标记可执行文件中定义的重复符号，该可执行文件抢占库定义，符合节 15.9 中所述的“作为自有导入”约定。加载此可执行文件时，动态加载器必须将库定义中的任何初始值复制到可执行文件的初始值。此重定位类型仅存在于可执行文件 (ET_EXEC) 的动态重定位表中。

R_6000_JUMP_SLOT 用于标记引用导入函数且仅从 PLT 条目引用的 GOT 条目，因此应遵循节 15.6 所述的延迟绑定。R_C6000_JUMP_SLOT 重定位仅发生在可执行文件和共享对象中，并且仅在动态重定位表的 DT_JMPREL 段中。

R_C6000_PREL31 用于对异常处理表中的代码地址进行编码。R_C6000_EHTYPE 用于对异常处理表中的 typeinfo 地址进行编码。请参阅节 11.2。

值为 33 到 65 的重定位用于线程局部存储 (TLS)。这些重定位包括 R_C6000_TBR_*、R_C6000_TPR_*、R_C6000_SBR_GOT_*_W_T*、R_C6000_TLSMOD 和 R_C6000_TBR_U32 重定位。有关线程局部存储的详细信息，请参阅章节 7。节 7.5 提供了使用这些 TLS 重定位的示例。

R_C6000_ALIGN 和 R_C6000_FPHEAD 用作 C64+ 压缩器的标记。它们在 ABI 下无效。将可重定位文件 (ET_REL) 组合到其他可重定位文件的下游工具 (例如部分链接) 应保留这些文件，或使用 R_C6000_NOCMP 标记它们所在的段。

R_C6000_NOCMP 将段标记为不可压缩。

13.5.2 重定位操作

表 13-6 提供了有关如何编码和执行每个重定位的详细信息。此表使用以下表示法：

F	可重定位字段。此字段使用元组 **[CS, O, FS]** 指定，其中 CS 是容器大小，O 是从容器的 LSB 到字段的 LSB 的起始偏移量，FS 是字段的大小。所有值均以位数表示。
R	重定位操作的算术结果
EV	要存储回重定位字段的编码值
SE(x)	x 的符号扩展值。从概念上讲，符号扩展是针对地址空间的宽度执行的。
ZE(x)	x 的零扩展值。从概念上讲，零扩展是针对地址空间的宽度执行的。

对于启用了溢出检查的重定位类型，如果编码值 (包括其符号，如果有) 无法编码到可重定位字段中，则会发生溢出。即：

- 如果编码值落在半开区间 [-2^{FS-1}...2^{FS-1}) 之外，则有符号的重定位会溢出。
- 如果编码值落在半开区间 [0 ⋯ 2^{FS}) 之外，则无符号的重定位会溢出。
- 如果编码值落在半开区间 [-2^{FS-1}⋯ 2^{FS}) 之外，则符号指示为*任一*的重定位会溢出。
- 如果编码值等于或大于模块 DSBT 表的大小，则 R_C6000_DSBT_INDEX 重定位会溢出。

表 13-6. C6000 重定位操作

重定位名称	符号	字段 [CS, O, FS] (F)	加数 (A)	结果 (R)	溢出检查	编码值 (EV)
R_C6000_NONE	无	[32、0、32]	无	无	否	无
R_C6000_ABS32	任一	[32、0、32]	F	S + A	否	R
R_C6000_ABS16	任一	[16、0、16]	SE(F)	S + A	是	R
R_C6000_ABS8	任一	[8、0、8]	SE(F)	S + A	是	R
R_C6000_PCR_S21	有符号	[32、7、21]	SE(F << 2)	S + A – P	是	R >> 2
R_C6000_PCR_S12	有符号	[32、16、12]	SE(F << 2)	S + A – P	是	R >> 2
R_C6000_PCR_S10	有符号	[32、13、10]	SE(F << 2)	S + A – P	是	R >> 2
R_C6000_PCR_S7	有符号	[32、16、7]	SE(F << 2)	S + A – P	是	R >> 2
R_C6000_ABS_S16	有符号	[32、7、16]	SE(F)	S + A	是	R
R_C6000_ABS_L16	无	[32、7、16]	F	S + A	否	R
R_C6000_ABS_H16	无	[32、7、16]	r_addend	S + A	否	R >> 16
R_C6000_SBR_U15_B	无符号	[32、8、15]	ZE(F)	S + A – B	是	R
R_C6000_SBR_U15_H	无符号	[32、8、15]	ZE(F << 1)	S + A – B	是	R >> 1
R_C6000_SBR_U15_W	无符号	[32、8、15]	ZE(F << 2)	S + A – B	是	R >> 2
R_C6000_SBR_S16	有符号	[32、7、16]	SE(F)	S + A – B	是	R
R_C6000_SBR_L16_B	无符号	[32、7、16]	ZE(F)	S + A – B	否	R
R_C6000_SBR_L16_H	无符号	[32、7、16]	ZE(F << 1)	S + A – B	否	R >> 1
R_C6000_SBR_L16_W	无符号	[32、7、16]	ZE(F << 2)	S + A – B	否	R >> 2
R_C6000_SBR_H16_B	无符号	[32、7、16]	r_addend	S + A – B	否	R >> 16
R_C6000_SBR_H16_H	无符号	[32、7、16]	r_addend	S + A – B	否	R >> 17
R_C6000_SBR_H16_W	无符号	[32、7、16]	r_addend	S + A – B	否	R >> 18
R_C6000_SBR_GOT_U15_W	无符号	[32、8、15]	ZE(F << 2)	GOT(s) + A – B	是	R >> 2
R_C6000_SBR_GOT_L16_W	无符号	[32、7、16]	ZE(F << 2)	GOT(s) + A – B	否	R >> 2
R_C6000_SBR_GOT_H16_W	无符号	[32、7、16]	r_addend	GOT(s) + A – B	否	R >> 18
R_C6000_DSBT_INDEX	无符号	[32、8、15]	无	DSBT 索引	是	R
R_C6000_PREL31	无	[32、0、31]	SE(F << 1)	S + A - PC	否	R >> 1
R_C6000_COPY	无	[32、0、32]	无	F	否	F
R_C6000_JUMP_SLOT	任一	[32、0、32]	F	S + A	否	R

表 13-6. C6000 重定位操作（续）

重定位名称	符号	字段 [CS, O, FS] (F)	加数 (A)	结果 (R)	溢出检查	编码值 (EV)
R_C6000_EHTYPE	任一	[32、0、32]	F	S + A – B	否	R
R_C6000_PCR_H16	有符号	[32、7、16]	r_addend	S-FP（P-A）	否	R >> 16
R_C6000_PCR_L16	无	[32、7、16]	r_addend	S-FP（P-A）	否	R
R_C6000_TBR_U15_B	无符号	[32,8,15]	ZE(F)	TBR(S)	是	R
R_C6000_TBR_U15_H	无符号	[32,8,15]	ZE(F<<1)	TBR(S)	是	R >> 1
R_C6000_TBR_U15_W	无符号	[32,8,15]	ZE(F<<2)	TBR(S)	是	R >> 2
R_C6000_TBR_U15_D	无符号	[32,8,15]	ZE(F<<3)	TBR(S)	是	R >> 3
R_C6000_TPR_S16	有符号	[32,7,16]	SE(F)	TBR(S)	是	R
R_C6000_TPR_U15_B	无符号	[32,8,15]	ZE(F)	TPR(S)	是	R
R_C6000_TPR_U15_H	无符号	[32,8,15]	ZE(F<<1)	TPR(S)	是	R >> 1
R_C6000_TPR_U15_W	无符号	[32,8,15]	ZE(F<<2)	TPR(S)	是	R >> 2
R_C6000_TPR_U15_D	无符号	[32,8,15]	ZE(F<<3)	TPR(S)	是	R >> 3
R_C6000_TPR_U32_B	无符号	[32,0,326]	ZE(F)	TPR(S)	否	R
R_C6000_TPR_U32_H	无符号	[32,0,326]	ZE(F<<1)	TPR(S)	否	R >> 1
R_C6000_TPR_U32_W	无符号	[32,0,326]	ZE(F<<2)	TPR(S)	否	R >> 2
R_C6000_TPR_U32_D	无符号	[32,0,326]	ZE(F<<3)	TPR(S)	否	R >> 3
R_C6000_SBR_GOT_U15_W_TLSMOD	无符号	[32,8,15]	ZE(F<<2)	GOT(TLSMOD(S)) + A - B	是	R >> 2
R_C6000_SBR_GOT_U15_W_TBR	无符号	[32,8,15]	ZE(F<<2)	GOT(TBR(S)) + A - B	是	R >> 2
R_C6000_SBR_GOT_U15_W_TPR_B	无符号	[32,8,15]	ZE(F<<2)	GOT(TBR(S)) + A - B	是	R >> 2
R_C6000_SBR_GOT_U15_W_TPR_H	无符号	[32,8,15]	ZE(F<<2)	GOT(TBR(S)) + A - B	是	R >> 2
R_C6000_SBR_GOT_U15_W_TPR_W	无符号	[32,8,15]	ZE(F<<2)	GOT(TBR(S)) + A - B	是	R >> 2
R_C6000_SBR_GOT_U15_W_TPR_D	无符号	[32,8,15]	ZE(F<<2)	GOT(TBR(S)) + A - B	是	R >> 2
R_C6000_SBR_GOT_L16_W_TLSMOD	无符号	[32,7,16]	ZE(F<<2)	GOT(TLSMOD(S)) + A - B	否	R >> 2
R_C6000_SBR_GOT_L16_W_TBR	无符号	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	否	R >> 2
R_C6000_SBR_GOT_L16_W_TPR_B	无符号	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	否	R >> 2
R_C6000_SBR_GOT_L16_W_TPR_H	无符号	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	否	R >> 2
R_C6000_SBR_GOT_L16_W_TPR_W	无符号	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	否	R >> 2
R_C6000_SBR_GOT_L16_W_TPR_D	无符号	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	否	R >> 2
R_C6000_SBR_GOT_H16_W_TLSMOD	无符号	[32,7,16]	ZE(F<<2)	GOT(TLSMOD(S)) + A - B	否	R >> 18
R_C6000_SBR_GOT_H16_W_TBR	无符号	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	否	R >> 18
R_C6000_SBR_GOT_H16_W_TPR_B	无符号	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	否	R >> 18
R_C6000_SBR_GOT_H16_W_TPR_H	无符号	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	否	R >> 18
R_C6000_SBR_GOT_H16_W_TPR_W	无符号	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	否	R >> 18
R_C6000_SBR_GOT_H16_W_TPR_D	无符号	[32,7,16]	ZE(F<<2)	GOT(TBR(S)) + A - B	否	R >> 18
R_C6000_TLSMOD	无符号	[32,0,32]	F	TLSMOD(S)	否	R
R_C6000_TBR_U32	无符号	[32,0,32]	F	TBR(S)	否	R
R_C6000_FPHEAD	无	无	无	无	否	无
R_C6000_NOCMP	无	无	无	无	否	无

13.5.3 未解析的弱引用的重定位

应满足引用未定义弱符号的重定位的以下条件：

- 在绝对重定位类型 (R_C6000_abs*) 中使用时，引用解析为零。
- 在基址相对重定位类型 (R_C6000_SBR*) 中使用时，引用解析为静态基址 (B)。

在 R_C6000_PCR_S21 重定位中使用时，要重定位的指令具有以下形式：

```
      B.S2   sym      ; R_C6000_PCRS21
```

然后，此指令替换为：

```
      B.B2   B3
```

所有其他情况均不符合 ABI。

备注

根据本规范中的其他要求，如果弱符号经过解析且 21 位 PC 相对地址无法到达目标目的地，则链接器必须生成蹦床函数来实现重定位。

章节 14

ELF 程序加载和动态链接（处理器补充）

一般而言，*程序加载*描述了获取表示为 ELF 文件（或者在动态链接的情况下，多个 ELF 文件）的程序并开始其执行所涉及的步骤。从本质上讲，该过程是特定于平台和系统的。

动态链接是一组相关机制，使程序能够包含在加载时链接和重新定位的单独构建组件，并在多个可执行文件之间共享这些组件。

系统可根据其具体要求来使用机制的子集。例如，仅运行一个进程的裸机平台可能需要动态链接和加载，但不需要与位置无关或共享对象。

这部分 ABI 基于 System V ABI 标准 (http://www.sco.com/developers/gabi/2003-12-17/contents.html) 的第 5 章，其中描述了目标文件信息和创建运行程序的系统操作。本节包含针对元件标准的处理器特定补充，这些元件通用于大多数基于 C6000 的系统。本节还定义了一种特定配置文件，称为裸机动态链接模型。

该 ABI 定义的另一种特定配置文件是 Linux 模型。针对 Linux 的 System V ABI 标准的处理器特定补充，请参阅章节 15。

14.1 程序标头

程序标头包含以下字段。

p_type

C6000 为程序标头中的 p_type 字段定义了一个特定于处理器的段类型。

名称	值	注释
PT_C6000_PHATTR	0x70000000	扩展段属性

PT_C6000_PHATTR 段类型将此段标识为包含有关程序中其他 PT_LOAD 段的额外描述性信息。此段包含 SHT_TI_PHATTRS 类型的单个段。章节 19 详细说明了程序标头属性。

p_vaddr，p_paddr

C6000 当前没有虚拟寻址。p_vaddr 和 p_paddr 字段指示段的执行地址。在一个地址加载并复制到另一地址执行的段在目标文件中由两个不同的段表示：一个是包含此段的代码或数据的加载映像段，其地址字段引用载入地址；另一个是未初始化的运行映像段，其地址字段引用运行地址。应用负责在适当时将加载映像的内容复制到运行地址。

p_flags

为 C6000 定义了一个特定于处理器的段标志。

名称	值	注释
PF_C6000_DPREL	0x10000000	使用 DP 相对寻址访问

PF_C6000_DPREL 标志用于标识使用 DP 相对寻址访问的段，因此受链接后放置约束。与位置无关的模块通常不包含用于 DP 相对寻址的动态重定位。如果存在多个 DP 相对段，则必须维护它们相对于 DP (以及彼此之间) 的位置。此标志用于将此类段标识到动态加载器或其他链接后代理，以便其协调此类段的分配。

TI 工具链使用一些辅助段属性。由于可用标志的数量有限，我们为其他段属性定义了另一种替代机制，即章节 19 所述的程序标头属性表。

p_align

如 System V ABI 中所述，可加载段在文件中对齐，以使其 p_vaddr (存储器中的地址) 和 p_offset (文件中的偏移量) 一致，模数为 p_align。在具有虚拟存储器的系统中，p_align 通常指定页面大小。除非针对特定平台指定，在 C6000 中，未指定 p_align 的含义和设置。

14.1.1 基址

与位置无关的代码可在任何地址加载并运行，不必是在程序头文件的 p_vaddr 字段中指定的地址，并且不需要加载时重定位。但是，由于段可能会使用相对偏移来相互引用，因此即使它们加载到通过 p_vaddrs 所指定位置以外的其他位置，也必须保持它们的相对位置。System V ABI 是指段之间的位移。指定实际地址作为*基址*。

可在不同地址加载与位置无关的段的程度取决于平台。不过，存在一些通用规则：

- 与位置无关的段要么必须在其指定的地址加载，要么必须在加载时重新定位。
- 具有 PHA_bound 属性的段必须在其指定的地址加载。

14.1.2 段内容

基本 ABI (本节) 不围绕必须存在哪些段或它们的内容定义任何要求。例如，C6000 程序可以包含任意数量的代码段和数据段，包括多个代码段、多个 DP 相对段和多个绝对数据段，如章节 4 和章节 5 中所述。特定的平台可能有自己的要求：例如，一些高级操作系统可能会将程序限制为只有一个代码段和一个数据段，或者可能二者只有一个段。

14.1.3 绑定段和只读段

如节 19.2 中所述，有一种机制可以使用附加属性对段进行注释。这一机制被用来代表应用于基于 ROM 的段的属性。

标有属性 PHA_BOUND 的段将被绑定到其指定地址，并且在下游重新链接、动态链接或动态加载步骤期间无法更改。此属性适用于本身位于 ROM 中或使用 ROM 内代码中的绝对地址引用的段。

标有属性 PHA_READONLY 的段表示其内容已锁定，不会受到任何重定位或其他下游更改的影响。此属性适用于位于 ROM 中的部分。动态加载器可以使用此示例作为提示，以避免对此类段进行重定位处理。

PHA_READONLY 段和一个在其程序头文件中具有 PF_R (只读) 段权限的段之间的区别是：PF_R 段通常可由加载器修改，但不能由程序本身修改，而 PHA_READONLY 段两者都不可修改。

14.1.4 线程局部存储

线程局部存储 (TLS) 是一个存储类，允许程序定义具有静态存储持续时间的线程特定变量。TLS 变量或线程局部变量是一个每线程实例化一次的全局/静态变量。有关线程局部存储的详细信息，请参阅章节 7。

C6000 EABI 支持 TLS，但这取决于运行时操作系统的线程库是否实现 __c6xabi_get_tp() 函数以及 TLS 支持的其他方面。

线程局部变量在 ELF 目标文件和模块中的表示与静态数据类似。不同之处在于，ELF 要求在段中分配线程局部变量，这些段具有在可重定位文件中设置的 SHF_TLS 标志。ELF 规范要求将段名称 .tdata 和 .tbss 分别用于初始化的和未初始化的线程局部存储。这些段具有读写权限。

在模块中，ELF 要求 TLS 段由 PT_TLS 段类型来指示。该段为只读。PT_TLS 段是 TLS 映像。

线程局部符号的符号类型为 STT_TLS。

14.2 程序加载

加载程序并开始执行程序时，有许多系统特定的方面。本节说明了常见于大多数系统的一般过程方面，重点介绍了特定于 C6000 的项目。

这些步骤可以通过结合使用离线代理（例如基于主机的加载器）、目标系统（例如操作系统）的运行时组件或链接到程序本身的库组件（例如自引导代码）来执行。

加载程序通常包括四组操作：创建过程映像、初始化执行环境、执行程序和执行终止操作。

创建过程映像包括将程序及其子组件复制到存储器中，并在需要时执行重定位。这些步骤必须由某些外部代理执行，例如基于主机的加载器或操作系统。

初始化执行环境中的一些步骤必须在程序开始运行之前（即在调用 main 之前）进行。这些步骤可由外部代理或程序本身执行。同样，终止操作在 main 返回（或调用 exit）时发生，并且可在外部执行或由程序执行。

表 14-1、表 14-2 和表 14-3 列出了创建、初始化和终止程序的步骤。虽然步骤的顺序不是绝对的，但必须遵守一定的依赖性。标有“仅 DL”的列表示步骤仅适用于使用动态链接或加载的系统。

表 14-1. 从 ELF 可执行文件创建过程映像的步骤

步骤		仅 DL
1.	确定每个可加载段的地址。在裸机或非动态系统中，这通常是段的程序标头的 p_vaddr 字段中的地址。有关其他注意事项，请参阅节 14.1。	
2.	初始化存储器系统并分配存储器。	
3.	将每段的内容复制到存储器中。如果某个段有未填充空间（即其文件大小小于其存储器大小），则将未填充空间初始化为 0。	
4.	为依赖库创建过程映像。依赖库由动态段中的 DT_NEEDED 条目标识。应检查库与目标处理器、ABI、操作系统和 DSBT 索引的兼容性。	√
5.	为此模块和所有依赖库分配 DSBT 索引。索引在可执行文件及其所有库中必须唯一。即使在多个程序之间共享，库的给定实例也必须只有一个索引。请参阅节 6.7。	√
6.	解析导入和导出符号之间的符号引用。具有动态链接的符号显示在动态符号表中，由动态段中的 DT_SYMTAB 标记标识。具有可见性 STV_DEFAULT 的导出符号可能会被父文件中的定义抢占。对于具有版本信息的符号（由动态段中的 DT_SYMVER 标记标识），加载器应确保引用与适当定义相匹配。	√
7.	如果需要，可进行重定位。加载时重定位由动态段中的 DT_REL 和/或 DT_RELA 标记指示。重定位按照节 13.5 中的说明进行处理。	√
8.	初始化可执行文件和依赖库的 DSBT 条目。此步骤有两个部分。首先，当前可执行文件的 DSBT 必须使用所有已加载模块（包括它在索引 0 处的自身）的静态基址进行初始化。其次，所有其他已加载模块的 DSBT 必须使用此模块的基址（在步骤 5 中分配给此模块的索引处）进行更新。	√
9.	Marshall 命令行实参和环境变量。此步骤特定于平台。	

表 14-2. 初始化执行环境的步骤

步骤		仅 DL
10.	设置 SP。SP (B15) 应设置为符号 _ _TI_STACK_END 的值，在 8 字节边界上正确对齐。	
11.	设置 DP。DP (B14) 应设置为符号 _ _C6000_DSBT_BASE 的值，对应于任何 DP 相对段的最低地址。	
12.	初始化变量。对于基于 ROM 的自引导系统，需要通过某种机制将基于 RAM（读写）的变量初始化为其初始值。此机制特定于工具链和平台。章节 18 说明了在 TI 工具中实现的一种此类机制。	
13.	执行 preinit 调用。这些调用是对初始化函数的调用，初始化函数定义为在依赖库的函数发生之前发生。preinit 调用由 System V ABI 中指定的动态段中的 DT_PREINIT_ARRAY 标记标识。	√
14.	根据 System V ABI“初始化和终止函数”一节中定义的顺序，以递归方式执行依赖库的初始化调用（步骤 15）。	√
15.	执行初始化调用。通常，这些调用是对模块中定义的全局对象的构造函数的调用。这些函数发生在依赖库的函数*之后*。指向初始化函数的指针存储在表中。在包含动态信息的文件中，此表由 DT_INIT_ARRAY 和/或 DT_INIT 标记标识。在其他文件中，此表由一对全局符号分隔：_ _TI_INITARRAY_Base 和 _ _TI_INITARRAY_Limit。	

表 14-2. 初始化执行环境的步骤（续）

步骤		仅 DL
16.	分支到入口点。入口点在 ELF 标头的 e_entry 字段中指定。在具有某些基础软件结构（例如操作系统）的系统上，入口点通常是 main 函数。在裸机系统上，此表列出的大多数初始化步骤可以由程序本身通过在执行 main 函数之前执行的库代码来执行。在此情况下，ELF 入口点是该代码的地址。例如，TI 工具提供了一个名为 _c_int00 的条目例程，此例程会在过程映像创建后开始步骤 10（设置 SP）中的序列。	

表 14-3. 终止步骤

步骤		仅 DL
17.	执行 atexit 调用。以与注册相反的顺序调用由 atexit 注册的函数。	
18.	根据 System V ABI 中定义的顺序，以递归方式对依赖库执行终止调用（步骤 19）。	√
19.	调用当前模块的终止函数，由 DT_FINI 和/或 DT_FINI_ARRAY 标记标识。	√

14.3 动态链接

动态链接是一组相关机制，使程序能够由单独构建的组件组成。这些机制包括：

- **链接机制**—支持单独链接对象之间的引用。它们主要包括动态段和相关子组件，例如动态符号表和动态重定位。
- **共享机制**—因此，共享代码的每个应用程序都可在不同位置拥有其数据的私有副本。具有 MMU 的系统通常依赖于从虚拟到物理地址的转换。C6000 缺少 MMU，依赖于称为数据区段基表的机制，如 章节 6 中所述。
- **寻址机制**—支持链接和共享。章节 6 中也概括性描述了这些机制。

系统可根据其具体要求来使用机制的子集。例如，仅运行一个进程的裸机平台可能需要动态链接和加载，但不需要与位置无关或共享对象。

目前，ABI 定义了两种具有不同功能级别的特定配置文件。一种是裸机动态链接模型，如 节 14.4 中所述。另一种是 Linux 模型，如 章节 15 中所述。

14.3.1 程序解释器

如节 14.2 所述，程序加载由外部代理执行。在 Linux 和可能基于其他操作系统的系统上，负责执行此功能的代理作为程序标头的 PT_INTERP 标记存储在可执行文件本身中。通常这是动态加载器，例如 ld.so。

裸机可执行文件不依赖于解释器；系统负责知晓如何加载程序。裸机动态可执行文件可能在 PT_DYNAMIC 段中包含动态信息，但在 PT_INTERP 条目中不包含动态信息。

14.3.2 动态段

正如 System V ABI 中指定的，动态链接程序在其程序标头中具有 PT_DYNAMIC 类型的条目。该条目指向名为 .dynamic 的特殊段，其段类型为 SHT_DYNAMIC，其中包含与动态链接和加载相关的信息。动态段是指动态符号表段和动态重定位段等其他段，统称为*动态信息*。

动态信息可包含或不包含在程序的可加载映像内（即，在一个或多个 PT_LOAD 区段内），具体取决于平台特定约定。如果动态信息不可加载，则引用对象组件的动态标签将表示为文件偏移量而不是虚拟地址。

System V ABI 中指定了动态段。表 14-4 中列出了一些 C6000 特定动态标签。

表 14-4. C6000 动态标签

名称	容值	d_un	可执行	共享对象
DT_C6000_GSYM_OFFSET	0x6000000D	d_val	可选	可选
DT_C6000_GSTR_OFFSET	0x6000000F	d_val	可选	可选
DT_C6000_PRELINKED	0x60000011	d_val	可选	可选
DT_C6000_DSBT_BASE	0x70000000	d_ptr	强制（如果是 DSBT 模型）	强制（如果是 DSBT 模型）
DT_C6000_DSBT_SIZE	0x70000001	d_val	强制（如果是 DSBT 模型）	强制（如果是 DSBT 模型）
DT_C6000_PREEMPTMAP	0x70000002	d_ptr	可选	可选

表 14-4. C6000 动态标签 （续）

名称	容值	d_un	可执行	共享对象
DT_C6000_DSBT_INDEX	0x70000003	d_val	可选	可选

全局符号标记标签

动态符号表中的符号指定为局部符号或全局符号。局部符号仅在重定位其包含的模块时才需要；局部符号不涉及动态符号解析，因此动态加载器可在重定位模块后丢弃局部符号。在动态符号表中将局部符号分组在全局符号之前，有助于动态加载器在裸机平台上利用这个机会。DT_C6000_GSYM_OFFSET 标签包含动态符号表 (.dynsym) 中第一个全局符号的偏移量。DT_C6000_GSTR_OFFSET 标签包含动态字符串表 (.dynstr) 中第一个全局符号名称的偏移量。

局部符号仍可出现在标签标记的位置之后，但需保证在标记位置之前没有全局符号。

DT_C6000_PRELINKED

该标签仅用于裸机负载模块，表明文件已具有为其分配的虚拟地址，可能是由预链接器或类似工具分配的。该值表示时间戳。

DT_C6000_PRELINKED 与 Linux 预链接器所用 DT_GNU_PRELINKED 标签类似，但由于裸机预链接并不完全相同，因此定义了不同的标签。

DSBT 标签

这些标签用于使用 DSBT 模型来实现位置无关的加载模块中 (请参阅 节 6.7) 。DT_C6000_DSBT_BASE 标签标记数据区段的静态链接位置；它对应于 _ _c6xabi_DSBT_BASE 符号。由于加载模块无需包含符号表，因此在该标签中复制值。

DT_C6000_DSBT_SIZE 标签指定保留用于 DSBT 表的大小。所有加载模块的表大小必须至少与其中编号最大的 DSBT 索引一样大。如果所加载模块的表太小或索引太大，则加载程序无法加载该模块。

如 节 6.7 中所述，模块的 DSBT 索引可由链接器静态分配，也可由加载器动态分配。如果加载模块具有静态分配的索引，则 DT_C6000_DSBT_INDEX 标签指定该索引的值。同一进程中的其他动态链接模块都不能使用相同的索引。具有动态可分配索引的模块省略该标签。

DT_PREEMPTMAP

该标签包含依赖静态绑定来预计算符号占先的平台的占先映射的文件偏移量。

DT_PLTGOT

该标签包含全局偏移量表 (GOT) 的虚拟地址。

动态重定位标签

System V ABI 定义了七个动态标签，用于标识目标文件中动态重定位的位置和类型：

- DT_RELA、DT_RELASZ—这些标签标识动态重定位的开始和大小。
- DT_PLTREL—该标签标识表的 DT_JMPREL 段中的重定位类型。对于 C6000，其值始终为 DT_RELA。
- DT_JMPREL、DT_PLTRELSZ—这些标签标识 DT_RELA 表的子范围，其中包含仅由 PLT 条目引用的符号的重定位。
- DT_REL、DT_RELSZ—C6000 不使用这些标签。

基本规范未明确由 DT_RELA 和 DT_RELASZ 描绘的动态重定位是否包括由 DT_JMPREL 和 DT_PLTRELSZ 描绘的 PLT 特定重定位。C6000 ABI 采用约定：DT_RELA 表包括 DT_JMPREL 表。

14.3.3 共享对象依赖关系

可执行文件可能依赖于库，后者又可能依赖其他库。这些依赖关系会编码到动态段中的 DT_NEEDED 条目内。当可执行文件或库依赖另一个库时，依赖库由引荐者动态段中的 DT_NEEDED 条目来命名。动态链接器必须找到依赖库，并按照节 14.2 中所述将其加载。

某些平台 (如 Linux) 具有用于查找依赖库的标准化搜索机制，例如 LD_LIBRARY_PATH 环境变量 (如 System V ABI 中所述) 。裸机平台没有标准化指导原则。在任何情况下，符号解析都会按照 System V ABI 中描述的广度优先方式进行。

14.3.4 全局偏移量表

某些上下文 (包括多个可执行文件之间的共享库) 需要与位置无关的寻址。为避免将与位置相关的地址编码到代码区段中，特地将此类地址生成到称为全局偏移量表 (GOT) 的表中，该表是每个静态链接单元的数据区段的一部分。程序不直接访问对象，而是从 GOT 读取变量的地址并间接寻址该变量。GOT 是数据段的一部分，始终使用静态链接时固定的偏移量来进行 DP 相对寻址。它由链接器生成，以响应编译器发出的特殊 GOT 生成重定位。当地址已知时，GOT 中的地址会在动态链接时得到修补。

编译器使用特殊重定位条目来引用 GOT。静态链接器自身生成表，以便响应特殊重定位。表条目本身具有 (动态) 重定位，动态加载器使用这些重定位来修补引用对象的最终解析地址。节 6.6 中介绍了基于 GOT 的寻址。节 13.5.1 中介绍了应用于 GOT 条目的重定位。

使用裸机模型的可执行文件和库可能需要、也可能不需要基于 GOT 的寻址。

14.3.5 过程链接表

如节 6.5 所述，过程链接表 (PLT) 是一组存根，用于将一个加载模块的调用连接到另一模块中的导入函数。导入函数的地址在静态链接时未知，因此静态链接器会生成一个与位置无关的存根来调用此函数，并修补初始调用以执行此存根。存根在加载时根据被调用者的动态链接地址进行重定位。

PLT 是代码段的一部分。PLT 条目可以使用绝对寻址或基于 GOT 的寻址来寻址被调用者，具体取决于是否需要位置无关性。

14.3.6 抢占式

当库中定义的符号被*早期*可执行文件或库中的定义屏蔽时，就会发生抢占。所谓早期，根据的是可执行文件及其依赖库形成的依赖树建立的广度优先顺序。

只有对符号的所有引用 (甚至来自定义它的模块) 都使用基于 GOT 的寻址时，此符号才能被抢占。动态链接器通过直接将覆盖符号的地址修补到 GOT 的适当槽中来执行抢占。

14.3.7 初始化和终止

加载模块可能需要在引用或调用之前执行初始化代码，例如模块中静态对象的 C++ 构造函数。类似地，当模块终止时，可能需要析构函数之类的终止代码。

模块使用动态段中的 DT_INIT、DT_INIT_ARRAY、DT_PREINIT_ARRAY、DT_FINI 和 DT_FINI_ARRAY 条目来指定任何所需的初始化和终止，如 System V ABI 所指定。

与初始化一样，加载器和/或执行环境负责根据由模块相关性施加的排序约束来执行终止函数。

14.4 裸机动态链接模型

裸机动态链接模型是一种与平台无关的模型，适用于需要单独链接组件但不受特定操作系统特定惯例约束的应用。可以选择性排除 DSBT 模型和基于 GOT 的寻址，从而降低动态链接几乎为零时的运行时性能损失，但代价是放置和寻址方案更加受限。

该模型在其最小形式、没有 DSBT、不具有位置独立性的情况下，支持*动态链接和加载*库，但不支持在不同的可执行文件之间*共享*库。换言之，在没有 GOT 和 DSBT 的情况下，裸机动态链接模型就会完全使用单个静态链接的裸机可执行文件的寻址方案，从而实现显著的性能优势，但代价是灵活性不足。

当需要更大的灵活性时，可以选择性启用 DSBT，以允许单独构建的库拥有它们自己的数据段。同样，可以选择性启用位置独立性，以允许在可执行文件之间共享库。

14.4.1 文件类型

程序可单独链接为可执行文件（文件类型 ET_EXEC）和依赖库（文件类型 ET_DYN）。在该模型下，这些文件分别称为*裸机动态可执行文件*和*裸机动态库*。这些文件包含通过 PT_DYNAMIC 程序标头引用的动态段中常用动态信息。可选择在加载时动态重定位程序及其库。

14.4.2 ELF 标识

对于符合该模型的可执行文件和共享对象，应在 ELF 标头的 EI_OSABI 字段中用 ELFOSABI_C6000_ELFABI 来标识。可重定位文件标识为 ELFOSABI_NONE。

14.4.3 可见性和绑定

全局符号的默认可见性为 STV_INTERNAL。也就是说，导入或导出的符号必须这样明确进行声明。不支持符号抢占。对于在共享对象之间具有模糊链接的符号（vtbls、rtti 类型信息等），不遵循单定义规则。裸机模型使用强制静态绑定。也就是说，链接器会在静态链接期间强制将导入的引用绑定到其定义。

在动态符号表中，所有具有 STV_DEFAULT 可见性的符号都标记为 STB_GLOBAL。也就是说，如果弱符号具有默认可见性，则它们将转换为全局符号。这是为了简化加载器的实现。

14.4.4 数据寻址

在裸机动态链接模型下，可选择使用 DSBT 模型。在不使用 DSBT 的情况下，程序具有单个 DP，它指向可执行文件的数据区段基地址（第一个 DP 相对区段）。可执行文件本身能够使用 near DP 相对寻址来引用其自带数据。必须使用 *far* 寻址模式（far DP 相对寻址或绝对寻址）来寻址库中的数据。这既适用于寻址导入数据的可执行文件，也适用于寻址其自带数据的库（因为 DP 属于可执行文件）。如果没有 DSBT，库就没有 .bss、.neardata 或 .rodata 段。

启用 DSBT 后，每个单独构建的组件都可拥有自己的 DP 相对区段。

在裸机动态链接模型中，也可选择通过基于 GOT 的寻址来寻址与位置无关的数据。如果没有基于 GOT 的寻址，对导入地址的引用将作为绝对地址编码到代码区段中，或者，对于非 DSBT 可执行文件，也可选择作为可执行文件 DP 的偏移量。此类代码无法访问库的数据区段的单独预处理副本，因此，虽然支持单独链接库，但不支持共享库。如果编译的代码不是与位置无关的，则可能需要加载时固定值。

链接器应强制一致地使用 DSBT 和 GOT 模型。

14.4.5 代码寻址

对导入函数的调用会通过可由编译器或静态链接器生成的 PLT 条目。不支持延迟绑定。根据所需的位置独立性程度，PLT 可使用绝对寻址、相对于 PC 的寻址或基于 GOT 的寻址来寻找函数的地址。

14.4.6 动态信息

动态标签使用文件偏移量（而不是 System V ABI 指定的虚拟地址）来引用动态信息。动态区段不是程序加载映像的一部分，也就是说，任何 PT_LOAD 区段都不包含 PT_DYNAMIC 和相关段。

表 14-5 汇总了裸机动态链接模型的特性，还比较了两种裸机文件类型。

表 14-5. 裸机动态链接文件

特性	裸机动态可执行文件	裸机动态库
ELF 文件类型 (e_type)	ET_EXEC	ET_DYN
ELF 标识 (e_ident)	ELFOSABI_C6X_ELFABI	
动态段可加载	否	
寻址自带数据	可具有 .bss、.neardata 和 .rodata，并使用 near DP 相对寻址来访问它们	带 DSBT：与可执行文件相同 不带 DSBT：Far (DP 相对、绝对或 GOT)
寻址导入数据	Far (DP 相对、绝对或 GOT)	
具有 PT_DYNAMIC 区段	是	
具有 PT_INTERP	否	
可导入/导出符号	可以，使用显式指令	
加载时重定位	可选	是
入口点	强制性	选项

章节 15

Linux ABI

本节指定了用于基于 C6000 Linux 系统的寻址、动态链接和程序加载的约定。我们的目标是尽可能遵循其他嵌入式无 MMU Linux 系统所采用的约定。

这部分 ABI 基于 System V ABI 标准 (http://www.sco.com/developers/gabi/2003-12-17/contents.html) 的第 5 章，其中描述了目标文件信息和创建运行程序的系统操作。本节与该 ABI 的其余部分一起构成了该标准的处理器特定补充，专门针对在 C6000 上的 Linux 下运行的程序。

这些约定适用于用户空间应用程序。内核是独立的，并且可遵循特定于实现的指南。

15.1 文件类型

可单独链接程序，使其由可执行文件（文件类型 ET_EXEC）和共享库（文件类型 ET_DYN）组成。这些文件包含通过 PT_DYNAMIC 程序标头引用的动态段中常用动态信息。可选择在加载时动态重定位程序及其库。

共享库应与位置无关。可执行文件可与位置无关，也可与位置相关。与位置相关的可执行文件需要在加载时重定位到代码区段，并且系统资源的使用效率不理想。

15.2 ELF 标识

对于符合 Linux ABI 的可执行文件和共享对象，应在 ELF 标头的 EI_OSABI 字段中用 ELFOSABI_C6000_LINUX 来标识。可重定位文件标识为 ELFOSABI_NONE。

之所以指定供应商特定的 ELFOSABI_C6000_LINUX 值而不是 ELFOSABI_LINUX，是为了区分该 ABI 变体（对应于无 MMU 的 Linux (uClinux)）与未来可能启用 MMU 的变体。

15.3 程序标头和段

以下是程序标头和段。

p_align

如 System V ABI 中所述，可加载段在文件中对齐，以使其 p_vaddr（存储器中的地址）和 p_offset（文件中的偏移量）一致，模数为 p_align。对于 Linux ABI，p_align 指定为 0x1000。

PT_INTERP 段

PT_INTERP 段包含具有动态加载器的目标文件的名称。对于本文档中描述的 ELF 可执行文件，解释器通常为 ld.so。

只读段

共享对象和可执行文件必须包含具有读取+执行权限的 PT_LOAD 段，此段包含模块的程序代码和可共享常量。可共享常量是指任何不可写入并且其值不包含地址的对象。此段还包括加载和执行程序所需的 ELF 结构体，包括文件标头、PT_INTERP、PT_PHDR、PT_DYNAMIC、PT_NOTE（如果有）和 PT_PHATTR（如果有）段。

与位置相关的可执行文件可能具有其他包含未指定内容的只读段或读取+执行段。如果有多个此类段，则不允许它们之间存在 PC 相对应用。如果加载器对这些段进行重定位，则不需要保留它们相对于彼此的位置。

数据段

共享对象和可执行文件必须包含一个具有读取+写入权限的 PT_LOAD 段，此段包含模块的 DSBT、GOT 和读写数据。此段使用 DP 相对寻址进行寻址，因此标有 PF_C6000_DPREL 标志。

ELF 要求段中未初始化的数据位于所有初始化数据之后。不过，如果 DP 相对段同时包含未初始化的 near 数据（如 .bss）和已初始化的 far 数据（如 .fardata），则未初始化数据可能需要位于初始化数据之前才能处于 DP 范围内。在此情况下，链接器需要用 0 填充该段的未初始化部分。

共享对象和可执行文件可能具有额外的读取+写入段。为确保位置无关性，必须仅使用基于 GOT 的寻址来对这些段进行寻址。与位置相关的可执行文件可使用绝对寻址。

栈段

C6000 Linux ABI 遵循通用约定，定义一个使工具链能够指定可执行文件最小栈分配的附加段类型。

名称	值	注释
PT_GNU_STACK	0x6474E551	栈大小和权限

p_flags 成员可指定包含栈的段的权限，并用于指示栈是否可执行。

如果没有此标头，栈的大小和权限将保持未指定状态。

绑定段和只读段

Linux 可执行文件和共享对象不应包含节 14.1.3 中所述的标记为绑定或只读的段。

15.4 数据寻址

共享库必须完全与位置无关。也就是加载时不会重定位到只读区段。必须通过 GOT 寻址任何具有可见性 STV_DEFAULT 的对象。所有其他静态数据都必须是 DP 相对寻址的。

可执行文件可以选择性地构建为与位置无关。可执行文件可使用 节 15.9 中所述的“作为自有导入”占先机制，避免对 STV_DEFAULT 变量使用基于 GOT 的寻址。

与位置相关的可执行文件可组合使用位置无关寻址与位置相关（绝对）寻址。使用绝对寻址的可执行文件会受加载时重定位的影响。

15.4.1 数据区段基表 (DSBT)

Linux 可执行文件和共享对象必须符合 节 6.7 中所述的 DSBT 模型。Near DP 区段必须包含 DSBT 表，该表的条目数至少与组成程序的所有模块中的最大 DSBT 索引数一样多。为在库供应商之间实现一致性，ABI 将默认 DSBT 表大小标准化为 64 个条目。DSBT 表的存在和大小由 节 14.3.2 中指定的 C6000 特定动态标签表示。

DSBT 索引 0 是为可执行文件保留的。DSBT 索引 1 是为程序解释器保留的。将以 2 开头的唯一 DSBT 索引静态分配给库。为满足与位置无关的约定，不支持动态重新分配 DSBT 索引。

15.4.2 全局偏移量表 (GOT)

如 节 6.6 中所述，可通过全局偏移量表 (GOT) 来实现代码和数据的位置独立性。GOT 是数据段的一部分，始终使用静态链接时固定的偏移量来进行 DP 相对寻址。GOT 由 4 字节*时隙*组成，该时隙包含动态分配的地址。Linux 可执行文件或共享对象必须具有至少有两个时隙（8 字节）的 GOT。GOT 条目标有引用动态符号的动态重定位。GOT 条目由静态链接器初始化，如下所示：

- 标有 R_C6000_JUMP_SLOT 重定位的 GOT 条目使用延迟绑定解析器存根的地址进行初始化，如 节 15.6 中所述。
- 所有其他 GOT 条目都初始化为零。

静态链接器必须保留 GOT 中的前两个时隙，以供延迟绑定器使用。请参阅节 15.6。

15.5 代码寻址

对于对导入的或可能导入的函数的调用，编译器或链接器会生成一个称为过程链接表条目的存根，如节 6.5 中所述。

需要通过 PLT 进行修补的调用通过满足以下所有条件的重定位类型进行标记：

- 重定位类型为 R_C6000_PCR21。
- 被引用符号的可见性为 STV_DEFAULT。
- 被引用符号的类型是 STT_FUNC 或 STT_NONE。

在可执行文件中，附加条件适用：

- 在包含调用的静态链接单元中，该符号未定义。

共享对象或位置无关的可执行文件中的 PLT 条目必须使用位置无关（基于 GOT）寻址来寻找被调用者的地址。在这种情况下，PLT 条目必须遵循节 15.6 中所述的延迟绑定约定。也就是说，PLT 的第一条指令必须加载 R_C6000_JUMP_SLOT relocation 条目的字节偏移量，以将被调用者的 GOT 条目标记到 B0 中。

位置相关的可执行文件中的 PLT 条目可以使用绝对寻址。C6000 不采用其他架构通用的惯例；在这些架构中，可将对函数地址的引用静态解析为 PLT 条目。请参阅节 6.7.3。

15.6 延迟绑定

对于大型程序，加载时符号解析会显著缩短程序启动时间。延迟绑定是一种机制，能够延迟函数符号的解析，直到程序实际调用函数符号为止，从而缩短启动时间并提高整体性能，因为只需要解析实际调用的函数。

一般方法是，通过 PLT 向量进行的首次调用依靠动态链接器中的解析器函数来实现控制，该函数执行解析并将后续调用直接重新路由至函数本身。

解析器需要两个实参。第一个是标识当前模块（该模块包含引用）的模块 id。模块 id 的表示不是 ABI 指定的，而是由加载器确定。第二个实参指定对应于目标函数的重定位条目。而该重定位条目提供目标符号的名称以及 GOT 中引用的位置。重定位条目由其在目标文件中的文件 .dynamic 段中 DT_RELPLT 标记中地址的字节偏移量指定。

所有这些都发生在调用方的后方，因此该机制必须保留影响标准函数调用接口的任何状态。特别是，它不得干扰任何用于实参传递的寄存器或返回地址寄存器，并且必须保留所修改的任何由被调用方保存的寄存器。为避免干扰正常实参寄存器，解析器的两个实参在 B0 和 B1 中传递。

保留全局偏移量表中的两个时隙，以供动态加载器用来实现延迟绑定。加载器使用 GOT[0] 来保存解析器函数的地址。GOT[1] 用于保存模块 id。

描述该机制的序列如下：

1. 静态链接器标识用于延迟绑定的候选。候选是仅由 PLT 条目引用的 GOT 条目；也就是说，仅用于调用导入函数。
2. 静态链接器生成特殊*解析器存根*，或包含库中的该存根。在该描述中，存根称为 PLT0，尽管 ABI 未指定其名称或位置。

3. 静态链接器使用 PLT0 的地址来初始化候选 GOT 条目，并使用 R_C6000_JUMP_SLOT 重定位进行标记。链接器在动态重定位表的段中定位任何此类重定位，该表标有 DT_JMPREL 标签。
4. PLT 条目是使用附加指令生成的，用于延迟解析。该指令使用解析器的两个实参中的第一个实参来加载寄存器 B0：R_C6000_JUMP_SLOT 重定位条目相对于动态重定位表的字节偏移量由 DT_REL[A] 标签指示。然后，它以常规方式从 GOT 加载目标地址并跳转到该地址。作为步骤 3 中初始化的结果，第一次发生该跳转时，控制权转移到 PLT0。
5. PLT0 将解析器的两个实参中的第二个实参从 GOT[1] 加载到 B1 中：一个由加载器定义的值，用于标识当前模块。然后，它从 GOT[0] 加载加载器的解析器函数的地址，并进行尾调用。
6. 解析器函数使用其两个实参来在由模块 id 指定的目标文件中查找指定的动态重定位，它在动态符号表中查找符号以获取函数的实际地址，并用该地址替换 GOT 条目。最后，它跳转到该地址，从而有效地尾调用目标函数。
7. 当输入 PLT 条目以供后续调用时，GOT 已更新为实际地址，因此控制权直接传递给函数。

延迟绑定 PLT 条目

```
$sym$plt:
      MVK   reloc_offset(sym),B0   ;byte offset of GOT reloc entry from DT_RELPLT
      MVKH  reloc_offset(sym),B0
      LDW   *+DP($GOT(sym)),tmp    ;&PLT0 first time, &sym after that
      B     tmp
```

解析器存根—PLT0

```
PLT0:
      LDW   *+DP($GOT(0)),tmp      ; address of resolver
      LDW   *+DP($GOT(4)),B1       ; module id
      B tmp                        ; tail-call resolver
```

全局偏移量表

```
              ; $GOT(0) reserved, initialized to module id
              ; $GOT(4) reserved, initialized to &resolver function
              ; ...
              ; $GOT(sym) R_C6000_JUMP_SLOT initialized to &PLT0
              ; updated to &sym by resolver
```

15.7 可见性

在 Linux 下，全局符号的默认可见性为 STV_DEFAULT。在共享对象中，具有 STV_DEFAULT 可见性的已定义符号会受到抢占，并且必须像导入它们一样对其进行寻址。在可执行文件中，“作为自有导入”约定（参阅节 15.9）允许将具有 STV_DEFAULT 可见性的已定义及未定义变量就像它们是 STV_INTERNAL 一样进行寻址；即使用 DP 相对寻址。

工具链可以实现一些供应商特定选项或扩展，来改变默认的可见性规则。在受影响的符号中，必须对可见性标志使用标准值，以此来反应这些变化。

15.8 抢占式

Linux 采用有关符号解析的约定，动态链接保留静态链接的行为。当同一符号有多个定义时，就会发生抢占：具体来说是指，库中定义的符号被*早期*可执行文件或库中的定义屏蔽。所谓早期，根据的是可执行文件及其依赖库形成的依赖树建立的广度优先顺序。

只有对符号的所有引用（甚至定义它的模块中的引用）都使用基于 GOT 的寻址时，此符号才能被抢占。动态链接器通过直接将覆盖符号的地址修补到 GOT 的适当槽中来执行抢占。

15.9 “作为自有导入”占先

在 Linux 中，外部符号通常具有 STV_DEFAULT 可见性，因此，除非另有声明，否则会受制于占先。这通常会导致对几乎所有对变量（包括在同一模块中定义的变量）的引用都采用基于 GOT 的寻址。换句话说，Linux 模块需要将所有对 extern 变量的引用都视为是导入的，即使它们不是导入的。为避免由此带来的性能损失，可执行文件采用了特殊约定，该约定允许可执行文件进行规避。

可执行文件可以选择将对变量的任何引用视为自带引用（即在可执行文件中定义的引用），从而允许编译器生成高效的 DP 相对寻址。在静态链接时，任何最终要导入的变量都会在可执行文件中获得重复定义。在动态加载时，重复定义会占先库中的原始定义，并且任何初始化值都会从占先定义复制到新定义。

重复定义的大小由源定义的 st_size 字段指定。重复定义的最小对齐方式如下所示：

- 令 *max* 为源模块中定义的给定大小对象所需的最大可能对齐方式。该值可确定为对象大小、章节 2 的对齐要求和由 TAG_ABI_array_object_alignment 构建属性指定的对齐的函数。
- 令 *vaddr* 为源模块中对象的虚拟地址。
- 重复对象的对齐是 *vaddr* 和 *max* 的最大公约数。

（直观上，这定义了重复对象至少与原始对象一样对齐，最多为其所需的最大可能对齐。）

创建过程映像时，存储在原始符号中的任何初始值都必须传播到副本。R_C6000_COPY 重定位可用于该目的。连接器使用 R_C6000_COPY 来标记可执行文件中的重复定义。在加载时，动态加载器在库中找到所引用的符号，并将该位置处的数据复制到可执行文件中的重复定义。

这样，可执行文件就不会因动态链接而受影响。相反，影响是由库承担的，库必须假设其所有外部变量都是导入的，因为占先，库无论如何都必须这样做。

15.10 程序加载

Linux 内核通过将程序的加载段和 PT_INTERP 标头指定的解释器程序的加载段复制或映射到存储器来开始加载程序的过程。内核随后跳转至解释器中的入口点，从而完成加载过程。对于 ELF 可执行文件，解释器通常是动态加载器 ld.so。

首次调用解释器时，它必须通过处理自己的动态重定位来引导自身。然后，它必须加载依赖库，执行任何动态符号解析，并处理程序本身的动态重定位。

内核通过称为加载映射的初始化数据结构将启动信息传递给解释器，声明如下：

程序加载映射数据结构

```
struct elf32_dsbt_loadmap
{
   /* Protocol version number, must be zero. */
   Elf32_Word version;
   /* Pointer to DSBT */
   unsigned   *dsbt_table;
   unsigned    dsbt_size;
   unsigned    dsbt_index;
   /* Number of segments */
   Elf32_Word  nsegs;
   /* The actual memory map. */
   struct elf32_dsbt_loadseg segs[nsegs];
};
struct elf32_dsbt_loadseg
{
   /* Core address to which the segment is mapped. */
   Elf32_Addr addr;
   /* Virtual address recorded in the program header. */
   Elf32_Addr p_vaddr;
   /* Size of this segment in memory. */
   Elf32_Word p_memsz;
};
```

内核通过寄存器中的 4 个实参和栈上的其余实参来调用内核。寄存器实参是：

B4	可执行文件的加载映射的地址
A6	解释器的加载映射的地址
B6	解释器动态段的地址
B14 (DP)	解释器的 _ _c6xabi_DSBT_BASE

内核为此过程分配一个栈并初始化 SP。栈初始内容将提供程序的命令行实参和环境变量：

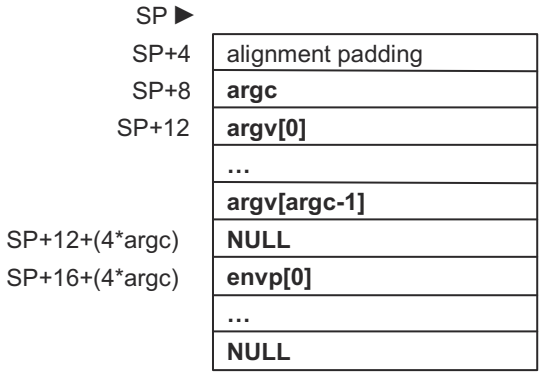


图 15-1. 程序加载映射数据结构 **的栈初始内容**

内核随后跳转至以符号 _start 标记的解释器的入口点。

15.11 动态信息

动态区段包含与程序加载和动态链接相关的信息。它由 System V ABI 指定。节 14.3.2 中指定了 C6000 特定动态标签的值和含义。Linux 模块不包含全局符号标记标签 DT_C6000_GSYM_OFFSET 和 DT_C6000_GSTR_OFFSET。

在 Linux ABI 中，所有动态链接元数据都是程序加载映像的一部分，即 PT_DYNAMIC 区段和相关段都包含在只读 PT_LOAD 区段中。因此，具有地址值 (d_ptr) 的动态标签表示为虚拟地址，而不是像在裸机 ABI 中那样的文件偏移量。

15.12 初始化和终止函数

System V ABI 指定用于可执行文件和共享对象的初始化序列，通过该序列，可在调用 main 之前调用诸如全局对象构造函数的函数。类似地，有一种机制可定义在 main 返回之后调用的函数。这些机制使用由 DT_INIT* 和 DT_FINI* 动态标签标记的函数指针表。

GC++ ABI 的第 3.3.5 节扩充了终止机制，使得 C++ 程序能够正确寄存析构函数，以便在使用共享对象的程序终止之前卸载共享对象时调用该析构函数。该机制使用了 C++ 编译器支持库中名为 _ _cxa_atexit 的 API 函数，其调用如下：

```
__cxa_atexit(dtor, obj, &__dso_handle);
```

（其中，dtor 是指向析构函数的指针，obj 是指向对象的指针。）

第三个实参 _ _dso_handle 是标识共享对象的唯一地址。C6000 ABI 将其值定义为模块 near DP 区段的地址。

另一个函数 _ _cxa_finalize 在卸载共享对象时实现对寄存函数的调用。该函数调用如下：

```
__cxa_finalize(&__dso_handle);
```

链接器必须安排该调用作为第一个终止操作发生，通常通过 DT_FINI* 表实现。由于 _ _cxa_finalize 带一个实参，调用的 DT_FINI 函数不带实参，因此链接器必须为该调用生成空的包装器函数。

如要总结该约定的要求，静态链接器需负责：

- 使用 near DP 区段的地址来生成隐藏符号 _ _dso_handle。
- 生成不带实参的包装器函数，该函数调用 _ _cxa_finalize，如上文所示。
- 将包装器函数寄存为标有 DT_FINI 或 DT_FINI_ARRAY 动态标签的终止函数列表中的第一个调用。

生成包含对 _ _cxa_atexit 的调用的任何可执行文件或共享对象时，应遵循这些要求。

15.13 Linux 模型摘要

表 15-1. Linux 程序文件

特性	位置相关可执行文件	位置无关可执行文件	共享对象
ELF 文件类型 (e_type)	ET_EXEC		ET_DYN
ELF 标识 (e_ident)	ELFOSABI_C6X_LINUX		
只读段	允许多个	One	
DP 相对数据段	One		
其他读写段	绝对	仅限 GOT	
代码寻址	PC 相对或绝对	PC 相对或 GOT	
自有隐藏数据寻址	DP 相对或绝对	DP 相对	
导入的 STV_DEFAULT 数据寻址	Far (DP 相对、绝对或 GOT)	DP 相对或 GOT	GOT
DSBT 模型	必需		
需要加载时重定位到只读段	是	否	
外部符号的默认可见性	STV_DEFAULT		
加载段包括元数据 (PT_INTERP、PT_PHDR、PT_DYNAMIC)	是		

章节 16

符号版本控制

符号版本控制提供了一种机制来支持共享库中的多个符号版本，并保证动态链接组件之间的兼容性。C6000 实现基于 GNU 工具链中使用的版本控制，而它又改编自 Sun Microsystems。关于 GNU 符号版本控制支持，参考文档是 Ulrich Drepper 在 http://people.redhat.com/drepper/symbol-versioning 上发布的论文。据我们所知，没有针对 C6000 的特定补充或偏差。本文档中的描述总结了该机制以供参考。

使用符号版本控制的可执行文件应将 ELF 标头中的 EI_OSABI 字段设置为适当的操作系统特定值。

16.1 ELF 符号版本控制概述

GNU 符号版本控制允许用户为从 DSO 导出的符号指定版本名称。这允许 DSO 中同一符号定义有多个版本，其中之一标记为默认值。当链接到该符号定义时，始终使用默认版本来绑定符号引用。

例如，假设库实现方在 codec_1_0.dso 中定义了 API 函数 api_do_encode。最初只有一个版本，称为 VER1。应用程序链接到该 DSO 时，对 api_do_encode 的所有引用都由 api_do_encode 的 VER1 解析。后来，实现方通过添加更新但不兼容的 api_do_encode 版本来增强 API，但仍然希望支持以前使用旧 API 构建的应用程序。创建新的 codec_2_0.dso 时，实现方可使用原始 VER1 api_do_encode 和同一符号的新 VER2 定义，VER2 现在成为指定的默认版本。新应用程序链接到 codec_2_0.dso 时，对 api_do_encode 的引用将由 VER2 api_do_encode 解析。原始 VER1 api_do_encode 仍然可用于满足根据 codec_1_0.dso 构建的旧应用程序的引用。

有关指定符号版本的机制的详细信息，请参阅 Drepper 的论文。

GNU 符号版本控制信息记录在三个 ELF 段中：

- **版本定义段**

  该段定义与从该可执行文件导出的符号相关的版本名称。该段还定义文件的版本。

  可通过动态段中的 DT_VERDEF 标签条目来定位该段。标签 DT_VERDEFNUM 包含该段所含版本定义的数量。版本定义段的段类型为 SHT_TI_verdef。请注意，该段类型值 0x6FFFFFFD 与 SHT_GNU_verdef 相同。本规范建议该段使用名称 .gnu.version_d。然而，仅应使用段类型来标识该段；不应使用名称。
- **版本需求段**

  该段记录该可执行文件中未定义符号引用需求的版本。每个条目都命名一个 DSO 并指向其需求版本的列表。动态链接器加载可执行文件时，将查找并加载所需的所有 DSO。在公开此类 DSO 之前，动态链接器将首先检查该 DSO 的版本定义是否满足可执行文件的版本需求。静态链接器将引用绑定到来自 DSO 的定义时，会记录该版本需求信息。
- **版本段**

  该段通过将版本号添加到动态符号条目来扩展动态符号表。该段所含条目数与动态符号表相同。符号 id 用于为该版本号表创建索引。如果符号未定义，则版本号与版本需求段中的版本需求条目匹配。如已定义符号，则版本号与版本定义段中的版本定义条目匹配。位 15 清零时，版本定义为*默认值*。

ELF 提供了一种机制，能够在 ELF 可执行文件中定位和标识这些符号版本段。这些段由动态段中的动态标签定位，并使用特殊段类型来标识。

例如，版本定义段由动态标签 DT_VERDEF 定位。DT_VERDEFNUM 标签包含版本定义段中的版本定义数。该段的段类型应为 SHT_GNU_verdef (0x6FFFFFFD)。该段的标称名称为 .gnu.version_d，但实现应依赖于段类型而不是名称。

16.2 版本段标识

表 16-1 列出了标签、段类型和段名称，这三种与符号版本控制关联的符号版本 ELF 段类型。

表 16-1. 版本段标识

ELF 段	动态标签	段类型	段名称
版本定义	DT_VERDEF (0x6FFFFFFC) DT_VERDEFNUM (0x6FFFFFFD)	SHT_GNU_verdef (0x6FFFFFFD)	.gnu.version_d
所需版本	DT_VERNEED (0x6FFFFFFE) DT_VERNEEDNUM (0x6FFFFFFF)	SHT_GNU_verneed (0x6FFFFFFE)	.gnu.version_r
版本	DT_VERSYM (0x6FFFFFF0)	SHT_GNU_versym (0x6FFFFFFF)	.gnu.versym

章节 17

构建属性

ARM ABIv2 规范的 ABI 规范定义了构建属性机制以捕获构建时选项，以便链接器能够强制实现可重定位文件的兼容性。C6x ELF 规范使用相同的结构对构建属性进行编码，如文档编号为 ARM IHI0045A、2007 年 11 月 13 日发布的 *ARM 架构的 ABI* 中的“ARM 附录”和“Errata”中的 ARM ABIv2 构建属性规范中所述。

17.1 C6000 ABI 构建属性子段

此 ABI 指定的属性记录在子段中，带有供应商字符串 C6000。工具链应仅使用这些属性来确定可重定位文件之间的兼容性；除非为此目的提供的 Tag_Compatibility 属性允许，否则不应使用特定于供应商的信息。

C6000 子段中的供应商数据包含任意数量的属性矢量。属性矢量以作用域标签开头，该标签指定它们是应用于整个文件还是仅应用于列出的段或符号。属性矢量具有以下三种格式之一：

1	**length**	**(omitted)**		**attributes**	Apply to file
2	**length**	**section numbers**	**0**	**attributes**	Apply to specified sections
3	**length**	**section numbers**	**0**	**attributes**	Apply to specified sections
ULEB128	uint32	ULEB128[ ]	ULEB128[ ]	See below	

长度字段指定整个属性矢量（包括其他字段）的长度，以字节为单位。符号和段号字段是段或符号索引的序列，以 0 结尾。

属性矢量中的属性表示为标签值对序列。标签表示为 ULEB128 常量。值为 ULEB128 常量或以 NULL 结尾的字符串。

在文件作用域内省略标签的效果等同于包含标签并将其赋值为 0 或 "" 相同，具体取决于形参类型。

为了允许消费者跳过无法识别的标签，对于偶数标签，形参类型会被标准化为 ULEB128，而对于奇数标签，则会被标准化为以 NULL 结尾的字符串。标签 1、2、3（作用域标签）和 32 (Tag_ABI_Compatibility) 是此约定的例外。

随着 ABI 的发展，可能会添加新的属性。为了使较旧的工具链能够可靠地处理可能包含它们无法理解的属性的文件，ABI 采用以下约定：

- 标签 0-63 必须能被使用的工具理解。如果遇到此范围内的未知标签，使用的工具可能会选择生成错误。
- 标签 64-127 用于传达消费者可以安全忽略的信息。
- 如果 N >= 128，则标签 N 与标签 N 模数 128 具有相同的属性。

17.2 构建属性标签

Tag_ISA (=4), ULEB128

此标签指定可以执行文件中所编码指令的 C6000 ISA。定义的值如下：

0	ISA 未指定
1	C62x
2	保留
3	C67x
4	C67x+
5	保留
6	C64x
7	C64x+
8	C6740
9	Tesla
10	C6600

此标签确定对象兼容性，如下所示。在这里，传递关系 A < B 表示 B 与 A 兼容；也就是说，B 可以执行为 A 或 B 生成的代码。组合属性时，应使用可以执行这两个属性的*最大* ISA（本例中为 B）。

- Tesla 与任何其他 ISA 修订版本均不兼容。
- C62x < 除 Tesla 之外的所有 ISA
- C67x < C67x+
- C67x+ < C6740
- C64x < C64x+
- C64x+ < C6740
- C6740 < C6600

下列有向图展示了 C6000 ISA 兼容性，其中有向边 A→B 表示兼容性关系 A < B。

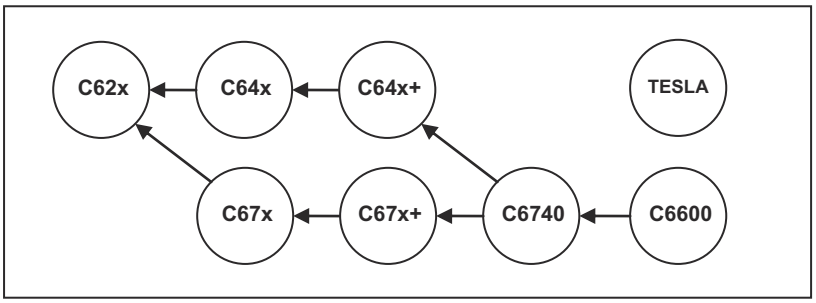


图 17-1. C6000 ISA 兼容性图

Tag_ABI_wchar_t, (=6), ULEB128

0	未使用 wchar_t。
1	wchar_t 的大小为 2 个字节。
2	wchar_t 的大小为 4 个字节。

节 2.1 将 wchar_t 指定为 unsigned int。但是，在某些情况下，TI 工具链将 wchar_t 定义为 unsigned short。利用此标签，可以检测此违规导致的任何不兼容性。

Tag_ABI_stack_align_needed, (=8), ULEB128

0	代码要求在函数边界处进行 8 字节栈对齐。
1	代码要求在函数边界处进行 16 字节栈对齐。

Tag_ABI_stack_align_preserved, (=10), ULEB128

0	代码要求在函数边界处进行 8 字节栈对齐。
1	代码要求在函数边界处进行 16 字节栈对齐。

当前支持的所有 ISA 都使用 8 字节栈对齐。未来的 ISA 预计将采用 16 字节对齐。

需要 16 字节栈对齐的代码与仅保留 8 字节对齐的代码不兼容。合并标签时，结果应反映 TAG_ABI_stack_align_preserved 给出的最小对齐，以及 TAG_ABI_stack_align_needed 给出的最大对齐。

Tag_ABI_DSBT, (=12), ULEB128

0	不使用 DSBT 寻址。
1	使用 DSBT 寻址。

Tag_ABI_PID, (=14), ULEB128

0	数据寻址与位置相关。
1	数据寻址与位置无关；GOT 通过使用 near DP 寻址来访问。
2	数据寻址与位置无关；GOT 通过使用 far DP 寻址来访问。

具有非零 Tag_ABI_PID 的目标文件不对数据使用绝对寻址。所有数据均使用 DP 相对 GOT 寻址，或者在只读常量的情况下使用 PC 相对寻址进行寻址。此类对象可以动态分配其 DP 相关数据段的位置，而不需要按照共享对象的要求进行重定位。

值 2 表示对象依赖于基于 far GOT 的寻址（请参阅 节 6.6）。也就是说，GOT 本身较远。

Tag_ABI_PIC, (=16), ULEB128

0	寻址约定不适用于共享对象。
1	寻址约定适用于共享对象。

Tag_ABI_PIC 指示该对象遵循共享对象所需的寻址约定，尤其是对导入变量的所有引用都通过 GOT 寻址。

链接共享库时，链接器应强制构成该库的所有对象上存在此标签。

名称 Tag_ABI_PIC 可能会产生误导。“位置独立性”一词可能意味着多个相关的属性，这些属性可能等于也可能不等于共享对象所需的属性。因此，此属性是根据后者集合定义的。

Tag_ABI_array_object_alignment, (=18), ULEB128

0	数组变量在 8 字节边界上对齐。
1	数组变量在 4 字节边界上对齐。
2	数组变量在 16 字节边界上对齐。

Tag_ABI_array_object_align_expected,(=20), ULEB128

0	代码假设数组变量采用 8 字节对齐。
1	代码假设数组变量采用 4 字节对齐。
2	代码假设数组变量采用 16 字节对齐。

前两个标签适用于具有外部可见性的数组变量，如 节 2.6 中所述。为了实现兼容性，TAB_ABI_array_align_expected 标签表示的对齐值必须小于或等于 TAG_ABI_array_object_alignment 标签表示的对齐值。合并标签时，结果应反映 TAG_ABI_array_object_alignment 给出的最小对齐和 TAG_ABI_array_object_align_expected 给出的最大对齐。

Tag_ABI_compatibility, (=32), ULEB128, char[]

此标签使供应商能够安排超出 ABI 范围的特定兼容性约定。它有两个操作数、一个 ULEB128 标志和一个以 NULL 结尾的字符串。该字符串指定由安排供应商定义的额外 ABI 约定的名称。该标志根据约定描述对象的特征。在以下说明中，“*ABI 兼容*”一词表示符合此 ABI，并符合本文档中规定的条件，例如其他构建属性标签。标志值为：

0	该对象没有特定于工具链的兼容性要求，因此与任何其他 ABI 兼容对象兼容。
1	该对象与其他 ABI 兼容对象兼容，前提是该对象由符合命名约定的工具链（例如，如果约定指定了供应商，则是该供应商的工具链）进行处理。
N>1	该对象与 ABI 不兼容，但可能与命名约定下的其他对象兼容。在这种情况下，约定对标志的解释作了定义。

请注意，该字符串标识额外的 ABI 约定，不一定是生成该文件的工具链。

如果省略了 ABI 兼容性标签，其含义与标志值为 0 的标签相同（无其他兼容性要求）。

Tag_ABI_conformance, (=67), char[]

此标签指定对象符合的 ABI 版本。标签值是以 NULL 结尾且包含 ABI 版本的字符串。此标准中指定的版本为“1.0”。小数点后面的数字仅供参考，不影响兼容性检查。

为了方便消费者识别 while 文件符合 ABI 的常见情况，此标签应该是 C6000 子段中第一个属性矢量的第一个属性。

表 17-1 总结了由 ABI 定义的构建属性标签。

表 17-1. C6000 ABI 构建属性标签

标签	标签值	形参类型	兼容性规则
Tag_File	1	uint32	
Tag_Section	2	uint32	
Tag_Symbol	3	uint32	
Tag_ISA	4	ULEB128	请参阅前面的说明
Tag_ABI_wchar_t	6	ULEB128	如果不为零，则必须完全匹配
Tag_ABI_stack_align_needed	8	ULEB128	必须与 Tag_ABI_stack_align_preserved 兼容。结合使用最大值。
Tag_ABI_stack_align_preserved	10	ULEB128	必须与 Tag_ABI_stack_align_needed 兼容。结合使用最小值。
Tag_ABI_DSBT	12	ULEB128	准确
Tag_ABI_PID	14	ULEB128	如果不同，则发出警告；结合使用最小值。

表 17-1. C6000 ABI 构建属性标签（续）

标签	标签值	形参类型	兼容性规则
Tag_ABI_PIC	16	ULEB128	构建共享库时如果不存在，则发出警告；结合使用最小值。
TAG_ABI_array_object_alignment	18	ULEB128	必须至少与 TAG_ABI_array_object_align_expected 对齐。 结合使用最大对齐。
TAG_ABI_array_object_align_expected	20	ULEB128	必须 <= TAG_ABI_array_object_alignment 中的对齐。 结合使用最小对齐。
Tag_ABI_compatibility	32	ULEB128 char[]	请参阅文本说明。
Tag_ABI_conformance	67	char[]	未指定

章节 18

复制表和变量初始化

本节提供复制表机制的一般性说明，然后详细介绍了涉及的数据结构。最后还介绍了如何以复制表的基本功能为基础构建 TI 工具链中的变量初始化实现。

18.1 复制表格式

复制表具有以下格式：

```
typedef struct
{
   uint16      rec_size;
   uint16      num_recs;
   COPY_RECORD recs[num_recs];
} COPY_TABLE;
```

rec_size 是 16 位无符号整数，指定表中每个复制记录的大小，以字节为单位。

num_recs 是 16 位无符号整数，指定表中的复制记录数。

表的其余部分由复制记录向量组成，每个复制记录具有以下格式：

```
typedef struct
{
   uint32  load_addr;
   uint32  run_addr;
   uint32  size;
} COPY_RECORD;
```

load_addr 字段是离线存储中源数据的地址。

run_addr 字段是将数据复制到的目标地址。

size 字段已重载：

- 如果大小为零，将会压缩加载数据。源数据具有特定于格式的编码，以表示其大小。在这种情况下，源数据前面的字节对压缩格式进行编码。格式被编码为*处理程序表*中的索引；该表是指向正在使用的每个格式的处理程序例程的指针表。
- 如果大小不为零，则源数据是要复制的数据的准确映像；换句话说，不会进行压缩。copy-in 操作只是将 ***size*** 字节从加载地址复制到运行地址。

源数据的其余部分特定于格式。copy-in 例程读取源数据前面的字节以确定其格式/索引，使用该值索引到处理程序表中，并调用处理程序来完成解压缩和复制数据。

处理程序表具有以下格式：

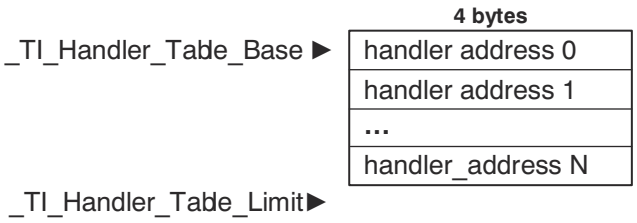


图 18-1. 处理程序表格式

copy-in 例程如所示那样通过由链接器定义的特殊符号引用该表。处理程序索引的分配不是固定的；链接器会根据每个应用程序所需的解压缩例程来为该应用程序重新分配索引。处理程序表生成到可执行文件的 .cinit 节中。

TI 工具链中的运行时支持库包含所有受支持压缩格式的处理程序函数。此函数的第一个实参是指向字节（位于 8 位索引后）的地址。第二个实参是目标地址。

Copy-In 函数的参考实现提供了 copy_in 函数的参考实现：

Copy-In 函数的参考实现

```
typedef void (*handler_fptr)(const unsigned char *src, unsigned char *dst);
extern int __TI_Handler_Table_Base;
void copy_in(COPY_TABLE *tp)
{
   unsigned short i;
   for (i = 0; i < tp->num_recs; i++)
   {
      COPY_RECORD crp = tp->recs[i];
      const unsigned char *ld_addr = (const unsigned char *)crp.load_addr;
      unsigned char       *rn_addr = (unsigned char *)crp.run_addr;
      if (crp.size)       // not compressed, just copy the data.
         memcpy(rn_addr, ld_addr, crp.size);
      else                // invoke decompression routine
      {
         unsigned char index = *ld_addr++;
         handler_fptr  hndl  = ((handler_fptr *)(__TI_Handler_Table_Base))[index];
        (*hndl)(ld_addr, rn_addr);
      }
   }
}
```

18.2 压缩的数据格式

理论上，压缩的源数据具有以下格式：

handler index	compressed data
1 byte	length is format-specific

图 18-2. 压缩的源数据格式

处理程序索引指定解码函数，该函数解释其余数据。复制表目前有两种受支持的压缩格式：行程编码 (RLE) 和 Lempel-Ziv Storer and Szymanski 压缩 (LZSS)。

18.2.1 RLE

8 位索引之后的数据使用行程编码 (RLE) 格式进行压缩。C6000 使用一种可通过以下算法解压缩的简单行程编码：

1. 读取第一个字节并将其分配为定界符 (D)。
2. 读取下一个字节 (B)。
3. 如果 B != D，则将 B 复制到输出缓冲区并转到步骤 2。
4. 读取下一个字节 (L)。
5. 如果 L > 0 且 L < 4，则将 D 复制到输出缓冲区 L 次。转到步骤 2。
6. 如果 L = 4，则读取下一个字节 (B')。将 B' 复制到输出缓冲区 L 次。转到步骤 2。
7. 读取接下来的 16 位 (LL)。
8. 读取下一个字节 (C)。
9. 如果 C != 0，则将 C 复制到输出缓冲区 L 次。转到步骤 2。
10. 处理结束。

TI 工具链中的 RLE 处理程序函数称为 _ _TI_decompress_rle。

18.2.2 LZSS 格式

8 位索引之后的数据使用 LZSS 压缩进行压缩。TI 工具链中的 LZSS 处理程序函数称为 _ _TI_decompress_lzss。有关格式的详细信息，请参阅 RTS 源代码中该函数的实现。

18.3 变量初始化

如节 4.1 中所述，初始化后的读写变量会被收集到目标文件的专门段中，例如 .data。该段包含程序启动时该段初始状态的映像。

TI 工具链支持两种加载此类段的模型。在所谓的 *RAM 模型* 中，一些未指定的外部代理（如加载器）负责将数据从可执行文件获取到它在读写存储器中的位置。这是基于 OS 的系统（在某些情况下为引导加载系统）中使用的典型直接初始化模型。

另一种模型称为 *ROM 模型*，适用于裸机嵌入式系统，它们必须能够在没有操作系统或其他加载器支持的情况下冷启动。初始化程序所需的任何数据必须驻留在持久性离线存储 (ROM) 中，并在启动时复制到其 RAM 位置。TI 工具链通过利用章节 18 中所述的复制表功能实现此目的。初始化机制与复制表在概念上相似，但细节稍有不同。

图 18-3 展示了 ROM 模型变量初始化的概念工作原理。在此模型中，链接器将数据从包含初始化变量的段中*移除*。这些段变为未初始化段，分配到它们在 RAM 中的运行时地址（比如 .bss）。链接器将初始化数据编码为一个特殊段，被称为 .cinit（代表 C 初始化），来自运行时库的启动代码在这里解码并复制到其运行地址。

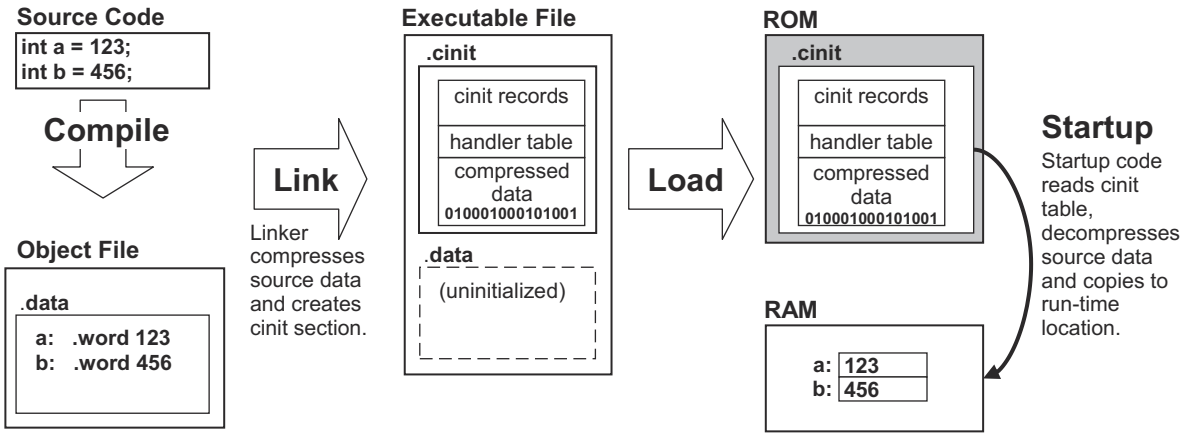


图 18-3. 通过 cinit 进行基于 ROM 的变量初始化

.cinit 表中的源数据可以压缩，也可以不压缩，与复制表类似。如果数据经过压缩，编码和解码方案与复制表相同，可以共享处理程序表和解压缩处理程序。

.cinit 段包含以下项目中的部分或全部：

- **cinit 表**包含 cinit 记录，与复制记录类似。
- **处理程序表**包含指向解压缩例程的指针（如节 18.1 中所述）。处理程序表和处理程序可通过初始化和复制表分享。
- **源数据**包含压缩或未压缩的数据，用于初始化变量。

这些项目顺序不限。

图 18-4 是展示 .cinit 段的原理图。

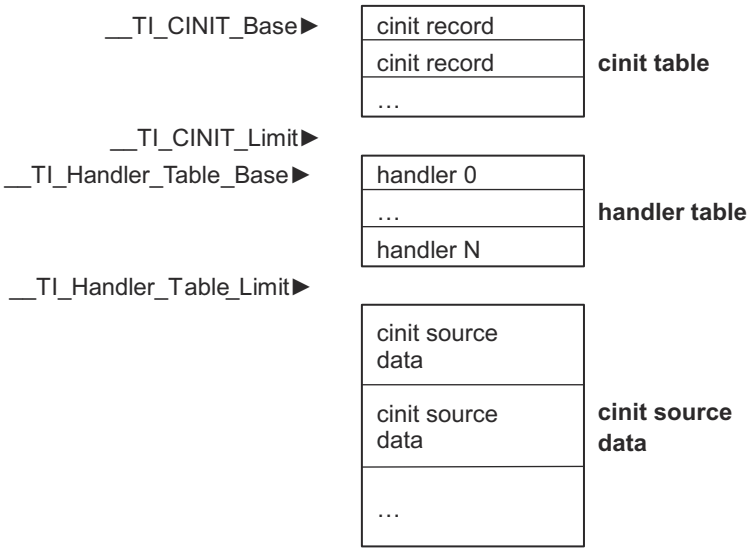


图 18-4. .cinit 段

.cinit 段的段类型为 SHT_TI_INITINFO，将其标识为此格式。工具应识别段类型，而不是名称 .cinit。

定义了两种特殊符号，来分隔 cinit 表：__TI_CINIT_Base 指向 cinit 表，而 __TI_CINIT_Limit 指向表末尾往后一个字节。启动代码使用这些符号来引用该表。

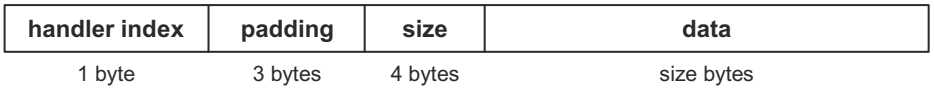
cinit 表中的记录有以下格式：

```
typedef struct
{
   uint32  source_data;
   uint32  dest;
} CINIT_RECORD;
```

- **source_data** 字段指向 cinit 段中的源数据。
- **dest** 字段指向目标地址。与复制表记录不同，cinit 记录不包含 size 字段；大小会始终编码到源数据中。

源数据与复制表压缩源数据格式相同（参阅节 18.1），并且处理程序的接口相同。除了 RLE 和 LZSS 格式，cinit 记录还定义了另外两种格式：未压缩和零初始化。

- 显式**未压缩**格式是必需的，因为与复制表记录不同，cinit 记录没有过载的大小字段。大小字段会始终编码到源数据中，即使未使用压缩也是如此。编码如下：



  编码数据包含一个大小字段，它在处理程序索引之后下一个 4 字节边界上对对齐。大小字段指定数据有效载荷中有多少字节，它紧跟在大小字段之后开始算起。初始化操作会将*大小*字节从数据字段复制到目标地址。TI 运行时库包含一个处理程序，被称为 _ _TI_decompress_none，用于未压缩格式。
- **零初始化**格式是一种紧凑格式，用于变量初始化值为零的常见情况。编码如下：

  大小字段在处理程序索引之后下一个 4 字节边界上对齐。初始化操作会在目标地址将*大小*连续字节填充为零。TI 运行时库包含一个处理程序，被称为 _ _TI_zero_init，用于此格式。

  如果可以使用相同格式便利地进行编码，链接器就可以自由地将相邻对象的初始化合并到单一 cinit 记录中，作为一项优化。对于零初始化对象而言，这通常很重要。

章节 19

扩展程序标头属性

ELF 可执行文件对象和共享库包含程序标头表。程序表中的每个条目都描述了一个区段。与其他元数据一起，程序表允许对区段属性进行有限的处理器特定扩展：最多可允许八个操作系统特定标志和四个处理器特定标志。

处理器特定 ABI 可使用这些标志来表示其他区段属性。

然而，可用的标志非常少，并且不能用于表示具有形参的属性。TI 预计需要在 ELF 程序标头表中指定其他系统/器件/应用程序特定的区段属性。区段标志不足以表示区段属性的全部需求，因此我们扩展了 ELF 格式，以便包括*扩展程序标头属性*。符合 C6000 EABI 的工具可选择将对扩展程序标头属性的支持作为实现质量的主题来实现。不要求对扩展程序标头属性的支持必须符合 C6000 EABI。

在处理器特定段中编码扩展程序标头属性，该段的类型为 SHT_TI_PHATTRS (0x7F000004)，名称为 .TI.phattrs。该段包含在由 PT_TI_PHATTRS (0x70000000) 类型的区段指定的区段中。

19.1 编码

在 <segment id, tag, value> 三元组中编码程序标头属性，可表示如下：

```
typedef struct
{
   Elf32_Half pha_seg_id;       /* Segment id */
   Elf32_Half pha_tag_id;       /* Attribute kind id */
   Union
   {
      Elf32_Off pha_offset;     /* byte offset within the .TI.phattrs section */
      Elf32_Word pha_value;     /* Constant tag value */
   }  pha_un;
}  Elf32_TI_PHAttrs;
```

区段 id 和标签 id 都按照 ELF 文件的字节顺序编码为 2 字节无符号整数。联合体 pha_un 中的字段按照 ELF 文件的字节顺序编码为 4 字节无符号整数。该表示借鉴了动态标签的 <tag, value> 表示。

标签值可以是内联 32 位常量或 .TI.phattrs 段中的偏移量，该偏移量指向固定长度二进制数据 (FLBD) 或空终止字节字符串 (NTBS)。固定长度二进制数据大小应为 32 位对齐。

如果存在扩展程序标头属性区段，则其以 PHA_NULL 标签终止。

属性标签值和属性由 TI 分配和维护，并且是特定于处理器的。所有未定义值都保留供将来使用。

属性标签确定如何解释 pha_un 的值。每个属性都有预定义行为。pha_un 字段可解释为 pha_value 或 pha_offset，也可不使用。如果使用了 pha_offset，则该值指向 NTBS 或 FLBD。如果将 pha_offset 解释为 FLBD，则应预先定义字段长度。

19.2 属性标签定义

德州仪器 (TI) 引入了两个属性来提供原生 ROMing 支持。

表 19-1. ROMing 支持属性

名称	标签 ID	d_un	Length
PHA_NULL	0x0	忽略	无
PHA_BOUND	0x1	忽略	无
PHA_READONLY	0x2	忽略	无

- 属性 **PHA_BOUND** 表示段的地址绑定到最终地址，并且在下游重新链接、动态链接或动态加载步骤期间无法更改。此属性适用于本身位于 ROM 中或使用 ROM 内代码中的绝对地址引用的段。

  PHA_BOUND 还向静态或动态链接器指示此地址已分配，不可用于进一步分配。
- **PHA_READONLY** 表示段包含 *true* 常量数据；也就是说，不允许静态和动态链接器对内容执行任何重定位或以任何方式更改内容。PHA_READONLY 段不应有任何重定位条目。er 可以使用此示例作为提示，以避免对此类段进行重定位处理。

19.3 扩展程序标头属性段格式

扩展程序标头属性段包含三部分：

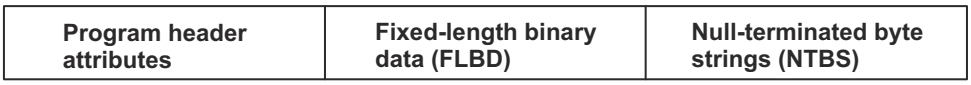


图 19-1. 扩展程序标头属性段的格式

第一部分是向量 Elf32_TI_PHAttrs，以 PHA_NULL 终止。接下来是 FLBD 部分和 NTBS 部分。如果使用，则 pha_un.pha_offset 应使用相对于该段开头的字节偏移量来指向 FLBD 或 NTBS 部分。如果没有标签访问 pha_offset 字段，则 FLBD 和 NTBS 可为空。

下表列出了 2011 年 9 月至 2014 年 3 月期间所做的变更。左列标识了本文档出现该特定改动的版本。

表 20-1. 修订历史记录

	位置	添加/修改/删除
SPRAB89A	章节 7	描述线程局部存储 (TLS) 的新章节。
SPRAB89A	节 13.5.1	添加了线程局部存储重定位类型。
SPRAB89A	节 13.5.2	添加了线程局部存储重定位操作。
SPRAB89A	节 14.1.4	有关线程局部存储的新章节。
SPRAB89A	表 8-8	表之后的讨论是有关线程局部存储的新章节。
SPRAB89A	节 13.3.5	有关线程局部存储的段的信息。

Changes from MARCH 31, 2014 to AUGUST 6, 2025 (from Revision A (March 2014) to Revision B (August 2025)) **Page**

- C6000 目标不再支持 OpenMP..... 54
- C6000 目标不再支持 OpenMP..... 55
- 改正了 R_C6000_JUMP_SLOT 重定位操作的拼写。..... 116

This page intentionally left blank.

重要通知和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、某特定用途方面的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他功能安全、信息安全、监管或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的相关应用。 严禁以其他方式对这些资源进行复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。您应全额赔偿因在这些资源的使用中对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，TI 对此概不负责。

TI 提供的产品受 TI 的销售条款或 ti.com 上其他适用条款/TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265