MSP430 嵌入式应用程序二进制接口

Application Note

Literature Number: ZHCADC5A
JUNE 2013 – REVISED JUNE 2020

目录

插图清单

表格清单

This page intentionally left blank.

章节 1

简介

本文档是德州仪器 (TI) 的 MSP430 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、和目标文件格式。

本规范旨在使 MSP430 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 - MSP430

在 TI 的 MSP430 编译器工具 4.0 版本发布之前，MSP430 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。

TI 编译器工具的 4.0 版本引入了一种名为 MSP430 EABI 的新 ABI。它基于 ELF 目标文件格式。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、程序加载和初始化。目前裸机是唯一受支持的平台。*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。

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。

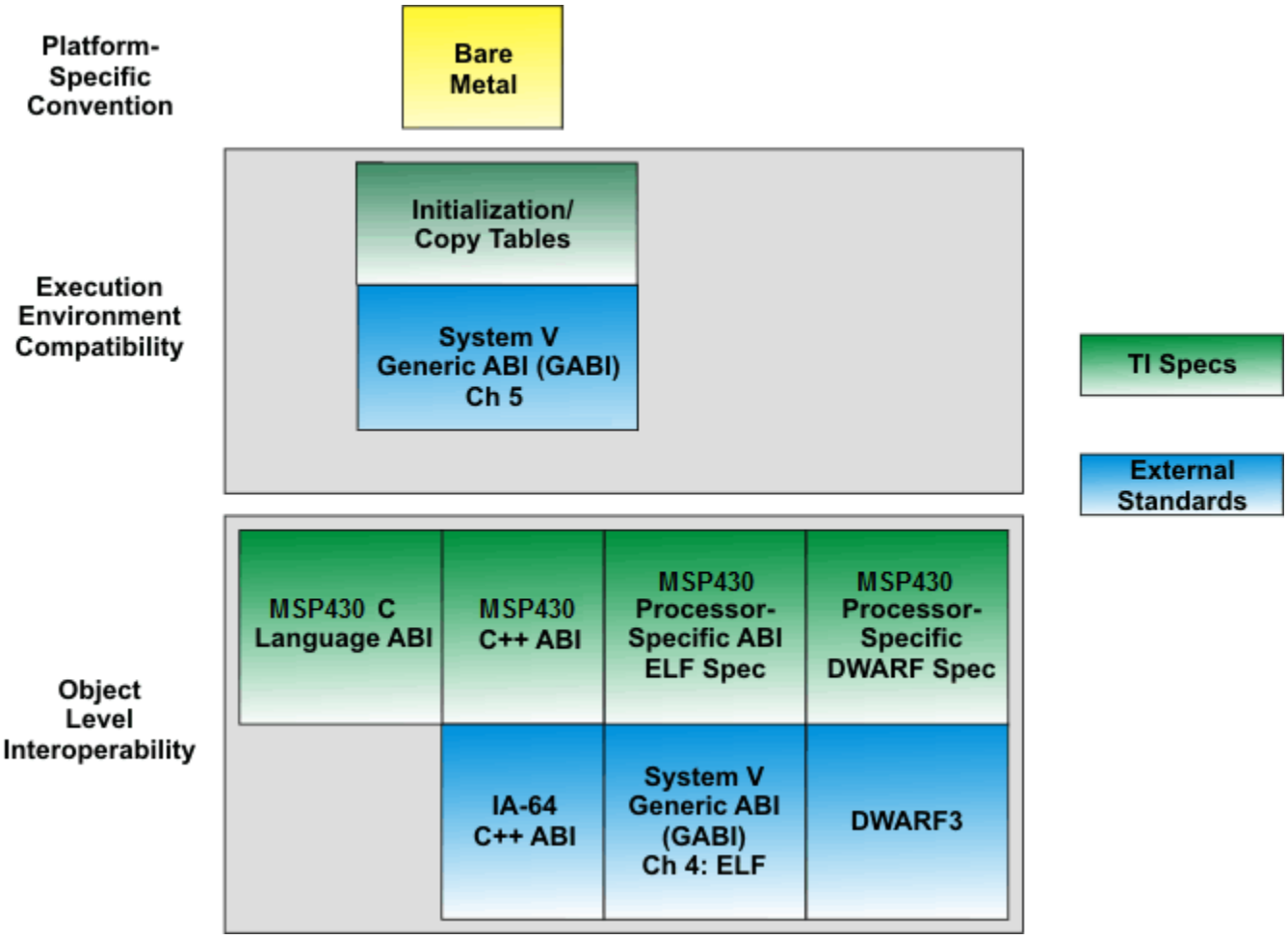


图 1-1. ABI 规范的组成部分

C 语言 ABI（章节 2、章节 3、章节 4、章节 5、章节 6 和章节 7）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 8）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 9）的详细信息。MSP430 C++ ABI 基于流行的 IA-64 (Itanium) C++ ABI。

DWARF 组成部分（章节 10）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 11）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 13）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。

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。

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 14）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。

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。ABI 中存在可以使用或不使用的模型变体。ABI 在使用此类变体的情况下对实现进行了标准化。有些变体彼此不兼容。如果任何对象使用特定的模型，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。

- **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，因此不需要位置独立性。

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 MSP430 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。

MSP430 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，以创建可执行文件文件。
- **可执行**文件包含适合执行的程序。

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。

1.7 段

ELF 加载模块（可执行文件）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，这在链接时静态发生。

1.8 MSP430 架构概述

该 MSP 系列具有以下架构：

- MSP430 具有 16 位 CPU 寄存器和一个 16 位地址空间。
- MSP430X 具有 20 位 CPU 寄存器和一个 20 位地址空间。它与 MSP430 的目标代码兼容。

MSP 器件上的存储器仅为小端字节序。

1.9 MSP430 存储器模型

一些 MSP 系列器件支持多种数据和代码存储器模型。这些模型因允许的对象和指针大小而异。编译器使用不同的指令和重定位来实现这些模型。

MSP430 只有一种存储器模型，即小型存储器模型。在此存储器模型中，代码和数据都仅限于较低的 64KB 存储器（16 位地址空间），MSP430 上仅有此存储器。这意味着代码和数据指针的大小均为 16 位。

MSP430X 支持 MSP430 小型存储器模型以实现向后兼容性，并引入了更多模型来利用更大的存储器。代码和数据存储器模型可以独立设置，但若要使用小型代码模型，需要小型数据模型。各种存储器模型相互不兼容；也就是说，具有不同存储器模型的目标文件可能无法链接在一起。

- 对于代码，具有小型和大型代码模型。在小型代码模型中，函数指针为 16 位。在大型代码模型中，函数指针为 20 位。
- 对于数据，具有小型、受限和大型数据模型。在小型数据模型中，对象指针为 16 位。在受限和大型数据模型中，对象指针为 20 位。

有关存储器模型和放置约定的详细信息，请参阅节 4.3.2.1。要了解指针如何因存储器模型而异，请参阅节 2.4。

1.10 参考文档

文档标题	链接或 URL
MSP430 优化 C/C++ 编译器用户指南	SLAU132
MSP430 汇编语言工具用户指南	SLAU131
MSP430x2xx 系列用户指南	SLAU144
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

1.11 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	schar	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	int16	16	16
unsigned int	uint16	16	16
long ，signed long	int32	32	16
unsigned long	uint32	32	16
long long、signed long long	int64	64	16
unsigned long long	uint64	64	16
enum	--	不尽相同（请参阅节 2.9）	16
float	float32	32	16
double	float64	64	16
long double	float64	64	16
指针	--	不尽相同（请参阅节 2.4）	16

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。

默认情况下，*char* 类型是无符号型。这与“signed char”和“unsigned char”类型不同，后者指定了它们的符号行为。在 TI 工具链中，可以使用 `--plain_char=signed` 编译器选项更改“char”类型的默认设置。

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。

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

有关 size_t 和 ptrdiff_t 类型的大小（具体取决于所选代码和数据模型），请参阅 节 2.4。

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 16 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 16 位 的有符号整数值将符号扩展到寄存器的高位。小于 16 位 的无符号值将加零扩展。

大小在 16 至 32 位之间的对象使用寄存器对。寄存器对是任意两个通用寄存器，一个保存值的最低有效部分，另一个则保存最高有效部分。在本文档中，寄存器对表示为 RL:RH，其中，RL 包含 LSB，RH 包含 MSB（例如，R12:R13）。MSP430 指令集不使用该表示法。有符号整数值符号扩展到 RH 的高位。无符号值进行加零扩展。

大小大于 32 位且最多为 64 位的对象使用四倍字寄存器。例如，R8::R11 是由寄存器 R8、R9、R10 和 R11 组成的四倍字寄存器。请参阅节 3.3.4。

2.3 存储器中的数据

MSP430 仅使用小端字节序模式。字节序是指多字节值的存储器布局。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：MOV.B 表示字节、MOV.W 表示字、MOVA 表示 20 位地址 。这些指令正确地考虑了进出存储器时的字节序。

20 位地址有 3 个字节，指定为 0 (LSB) 到 2 (MSB)。在存储器中，20 位值填充为 32 位（4 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：

location	little-endian
N	byte 0 (lsb)
N+1	byte 1
N+2	byte 2 (msb)
N+3	padding

图 2-1. 存储器中 20 位值的表示

2.4 指针类型

一些 MSP 系列器件支持多种数据和代码存储器模型。这些模型因允许的对象和指针大小而异。编译器使用不同的指令和重定位来实现这些模型。

节 1.9 概述了 MSP 系列支持的代码和数据模型。有关存储器模型和放置约定的详细信息，请参阅节 4.3.2.1。

指针的大小根据您使用的代码和数据模型而有所不同。代码和数据模型会影响大小、对齐，以及用于函数指针、数据指针、size_t 类型和 ptrdiff_t 类型的存储空间。始终在大小为 2 位幂的容器中存储大小不是 2 的幂的指针。也就是说，20 位类型存储在 32 位中。

表 2-2. 指针的数据大小

代码或数据模型	类型	大小	存储	对齐
小型代码模型	函数指针	16	16	16
大型代码模型	函数指针	20	32	16
小型数据模型	数据指针	16	16	16
小型数据模型	size_t	16	16	16
小型数据模型	ptrdiff_t	16	16	16
受限数据模型	数据指针	20	32	16
受限数据模型	size_t	16	16	16
受限数据模型	ptrdiff_t	16	16	16
大型数据模型	数据指针	20	32	16
大型数据模型	size_t	20	32	16
大型数据模型	ptrdiff_t	20	32	16

2.5 复数类型

支持 C99 标准中定义的 _Complex 类型。内部表示法如下所示：

```
struct _Complex
    { float_type real;
      float_type imag; };
```

2.6 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。

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。MSP430 仅使用小端模式。

2.7 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。

C62x、C67x	4 字节
所有其他	8 字节

2.8 位字段

MSP430 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 MSP430 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。MSP430 仅使用小端字节序模式。

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。

MSP430 EABI 将 plain *int* 的声明类型视为 *signed int*。

2.8.1 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。

2.9 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，如果 long 或 unsigned long 可表示所有枚举器，则使用该类型。否则，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

函数调用通过调用专用的 CALL 指令进行，该指令会将返回地址压入函数调用栈并跳转到被调用函数。被调用函数通过执行专用 RET 指令返回，该指令从堆栈中弹出返回地址并跳转到该地址。

MSP430X 在小型代码模型中使用 CALL 指令，在大型代码模型中使用 CALLA 指令。

3.1.1 调用指令

用于执行调用的指令因要调用的函数在调用时间是已知还是未知而异。对于已知函数，使用的指令还取决于使用的架构。

3.1.1.1 间接调用

在编译时，如果待调用的函数是未知的，那么对于所有架构，函数的地址将存储在 CPU 寄存器（“Rn”）中。该指令到达整个地址空间。例如：

```
CALL R5 ; small code model
CALLA R5 ; large code model
```

3.1.1.2 直接调用

如果在编译时已知调用函数，则所有架构都使用直接调用指令。该指令可使用立即、绝对、或符号寻址模式。此处的示例仅显示立即寻址模式。

MSP430 使用具有 16 位寻址模式的 CALL 指令。该寻址模式可到达所有有效代码存储器。

```
CALL #func    ; immediate mode, call func
```

MSP430X 小型代码模型与 MSP430 相同。CALL 指令不会到达整个地址空间。但是，使用小型代码模型时，任何代码都不能位于低 64KB 之外。

MSP430X 大型代码模型使用具有 20 位寻址模式的 CALLA 指令。该寻址模式可到达所有有效代码存储器。

```
CALLA #func    ; immediate mode, call func
```

3.1.2 返回指令

被调用函数通过执行专用 RET 指令返回，该指令从栈中弹出返回地址并分支到此地址。该指令用于具有小型代码模型的 MSP430 和 MSP430X。

如果在 MSP430X 上使用大型代码模型，则将执行 RETA 指令。

如果此函数是中断处理程序函数，则将使用 RETI 指令。

3.1.3 流水线约定

MSP430 流水线受到保护。无需考虑流水线延迟或指令完成（尽管这可能有助于提高代码性能）。

3.1.4 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。

ABI 支持调用导入的弱函数；即可能在其他不同静态链接单元中定义的函数。如果在链接时对弱函数的引用仍然未解析，则链接器会将其地址替换为零。在尝试调用弱函数前，用户负责添加一个检查，以确认地址不为零或 NULL。

3.2 寄存器惯例

所有架构都具有 16 个 CPU 寄存器。它们命名为 R0 到 R15。其中一些寄存器是专用寄存器。MSP430 具有 16 位寄存器。MSP430X 具有 20 位 CPU 寄存器，SR (R2) 除外，它是 16 位。有关寄存器的详细信息，请参阅 *MSP430x2xx 系列用户指南* (SLAU144)。

R0 是程序计数器 (PC)。程序计数器必须始终在一个字（2 字节）边界上保持对齐。

R1 是调用栈指针 (SP)。即使在手工编码的汇编函数中，栈指针也必须始终保持正确对齐（请参阅节 4.5.1）。MSP430 和 MSP430X 都需要对齐到 2 个字节。 栈管理和本地帧结构如节 4.5 所述。

R2 是状态寄存器 (SR)。在一些寻址模式中，如果寄存器是 SR，实际上会解释为常量。SR 表示的常量取决于寻址模式。有关详细信息，请参阅 *MSP430x2xx 系列用户指南* 中的“常量发生器寄存器”一节。

R3 是 的常量生成器寄存器。当使用此寄存器时，它解释为常量。R3 表示的常量取决于寻址模式。

实现不能将这些专用寄存器用于除专门特殊用途以外的任何目的。其余的寄存器为通用型寄存器。

对于 MSP430 和 MSP430X，ABI 将 R4-R10 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。

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。

表 3-1. MSP430 和 MSP430X 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
R0	PC	不适用	程序计数器
R1	SP	是	调用栈指针
R2	SR	不适用	状态寄存器
R3	CG	不适用	常量发生器寄存器
R4		是	
R5		是	
R6		是	
R7		是	
R8		是	函数实参（特殊情况）
R9		是	函数实参（特殊情况）
R10		是	函数实参（特殊情况）
R11		否	函数实参（特殊情况）
R12		否	函数实参，返回值
R13		否	函数实参，返回值
R14		否	函数实参，返回值
R15		否	函数实参，返回值

3.2.1 实参寄存器

其中四个通用寄存器用于将实参传递给函数。对于 MSP430 和 MSP430X，实参寄存器为 R12 至 R15。对于某些特殊情况，R8 至 R11 也用于传递实参。请参阅节 3.3.5。

3.2.2 被调用者保存的寄存器

需要由被调用的函数保留被调用者保存的寄存器，确保其在从函数返回时具有与调用时相同的值。

对于 MSP430 和 MSP430X，R4 到 R10 为被调用者保存的寄存器。但是，当将 R4 或 R5 用作全局寄存器时，调用者和被调用者都不应保存、恢复或以其他方式使用寄存器。全局寄存器用作中断函数的全局状态。对于 EABI，弃用了全局寄存器。

所有其他通用寄存器都是调用者保存的寄存器；也就是说，它们不会在调用中保留，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。

3.3 实参传递

对于 MSP430/430X，一个函数最多可以在寄存器中传递四个实参。

在寄存器中传递的实参数量取决于每个实参的大小和类型。实参按照声明的顺序分配给以下相应列表中的第一个可用寄存器：单寄存器、寄存器对或四倍字寄存器（存在以下特殊例外情况）。

对于 MSP430 和 MSP430X，实参寄存器为：**R12、R13、R14、R15**

CPU 寄存器的大小因架构而异。实参寄存器的使用方式也相应地有所不同。

单个寄存器不能包含多个实参。在寄存器中传递类型小于 int（16 位）的实参时，编译器可能会将其提升为寄存器的大小。TI 编译器会提升此类实参。请注意，如果在堆栈上传递这些实参，则 TI 编译器不会提升这些实参，除非将实参传递给可变实参函数或原型不在范围内（默认实参提升）。当要在寄存器中传递窄值时，调用方负责正确地对其进行符号扩展或零扩展，以填充寄存器宽度。

3.3.1 单个寄存器

类型适合单个 CPU 寄存器的实参在单个 CPU 寄存器中传递。

对于 MSP430 和 MSP430X，最高 16 位的类型在单个寄存器中传递。各种大小的指针类型也在单个寄存器中传递。

对于 MSP430X，当使用大型代码或大型数据存储器模型时，指针类型可以是 20 位，但 CPU 寄存器也是 20 位，因此指针值始终适合单个寄存器。对于非指针值，MSP430X CPU 寄存器被视为只有 16 位。其结果是，无论使用何种存储器模型，用于实现实参传递的寄存器对于 MSP430 和 MSP430X 都是相同的。

MSP430 和 MSP430X 示例：

C 源代码：

```
void func1(int a0, int a1, int a2, int a3);
int a0, a1, a2, a3;
void func2(void)
{
    func1(a0, a1, a2, a3);
}
```

编译后的汇编代码：

```
          MOV.W     &a0,R12
          MOV.W     &a1,R13
          MOV.W     &a2,R14
          MOV.W     &a3,R15
; call instruction here
```

MSP430X 示例：

C 源代码：

```
void func1(int *a0, int *a1, int *a2, int *a3);
int a0, a1, a2, a3;
void func2(void)
{
    func1(&a0, &a1, &a2, &a3);
}
```

在大型代码模型中编译：

```
          MOVX.A    #a0,R12
          MOVX.A    #a1,R13
          MOVX.A    #a2,R14
          MOVX.A    #a3,R15
; call instruction here
```

3.3.2 寄存器对

类型大于单个寄存器但不大于单个寄存器大小的两倍的实参在寄存器对中传递。编号最低的寄存器保存 LSW（最低有效字）。

对于 MSP430 和 MSP430X，寄存器对不需要对齐，因此 R12:R13、R13:R14 和 R14:R15 是有效寄存器对。最高 32 位的类型在寄存器对中传递。这包括“long int”、“float”以及通过值传递的最多 32 位大小的结构体。

MSP430 和 MSP430X 示例：

C 源代码：

```
void func1(int a0, long a1, int a2);
int a0, a2;
long a1;
func2(void)
{
    func1(a0, a1, a2);
}
```

编译后的汇编代码：

```
          MOV.W     &a0,  R12
          MOV.W     &a1+0,R13
          MOV.W     &a1+2,R14
          MOV.W     &a2,  R15
```

3.3.3 拆分对

对于 MSP430 和 MSP430X，可能会将一个 32 位实参在栈和存储器之间拆分。如果实参要在寄存器对中传递，但只有一个寄存器可用（始终为 R15），则编译器将在 R15 和栈上的一个寄存器大小的位置之间拆分该实参。

MSP430 和 MSP430X 示例：

C 源代码：

```
 void func1(int a0, long a1, long a2);
int a0;
 long a1, a2;
 func2(void)
 {
     func1(a0, a1, a2);
 }
```

编译后的汇编代码：

```
            SUB.W     #2,SP
            MOV.W     &a0,  R12
            MOV.W     &a1+0,R13
            MOV.W     &a1+2,R14
            MOV.W     &a2+0,R15
            MOV.W     &a2+2,0(SP)
```

3.3.4 四倍字（四寄存器实参）

对于 MSP430 和 MSP430X，大小大于 32 位且最多 64 位的实参使用四倍字寄存器。例如，R8::R11 是本手册中使用的表示法，表示由寄存器 R8、R9、R10 和 R11 依次组成的四倍字寄存器。编号最低的寄存器保存 LSW。四倍字寄存器必须对齐。因此，只有 R8::R11 和 R12::R15 是有效的四倍字寄存器。四倍字寄存器需要特殊处理。

如果剩余足够的实参寄存器（全部四个）来传递 64 位值，则将使用全部四个实参寄存器。否则，64 位值将完全在栈上传递。这可能会留下未使用的实参寄存器（也就是“空洞”）。调用约定将尝试用后续实参回填此空洞，但前提是这些后续实参完全适合寄存器（请参阅下面的示例 2）。也就是说，在任何实参被置于栈上之后，没有任何实参将被置于“拆分对“中，如节 3.3.3 所述。

MSP430 和 MSP430X 示例 1：

C 源代码

```
void func1(long long a0, long long a1);
long long a0, a1;
func2(void)
{
    func1(a0, a1);
}
```

在任何模型中编译：

```
            SUB.W     #8,SP
            MOV.W     &a0+0,R12
            MOV.W     &a0+2,R13
            MOV.W     &a0+4,R14
            MOV.W     &a0+6,R15
            MOV.W     &a1+0,0(SP)
            MOV.W     &a1+2,2(SP)
            MOV.W     &a1+4,4(SP)
            MOV.W     &a1+6,6(SP)
```

MSP430 和 MSP430X 示例 2：

此示例显示了一个不完全适合寄存器的 64 位实参，因此它完全在栈上传递，留下未使用的寄存器，然后用实参 a2、a3 和 a4 回填这些寄存器。

C 源代码

```
void func1(int a0, long long a1, int a2, int a3, int a4);
int a0, a2, a3, a4;
long long a1;
func2(void)
{
    func1(a0, a1, a2, a3, a4);
}
```

在任何模型中编译：

```
            SUB.W     #8,SP
            MOV.W     &a0,R12
            MOV.W     &a1+0,0(SP)
            MOV.W     &a1+2,2(SP)
            MOV.W     &a1+4,4(SP)
            MOV.W     &a1+6,6(SP)
            MOV.W     &a2,R13
            MOV.W     &a3,R14
            MOV.W     &a4,R15
```

MSP430 和 MSP430X 示例 3：

此示例显示了一个不完全适合寄存器的 64 位实参，因此它完全在栈上传递，留下未使用的寄存器，然后将其回填。不过，最后一个实参（32 位类型）不完全适合寄存器，因此它完全在栈上传递。如果 64 位实参未在栈上，则此类型将拆分，一部分在 R15 中传递，一部分在栈上传递。

C 源代码

```
void func1(int a0, long long a1, long a2, long a3);
int a0;
long long a1;
long a2, a3;
func2(void)
{
    func1(a0, a1, a2, a3);
}
```

在任何模型中编译（注意 R15 未使用）：

```
            SUB.W     #12,SP
            MOV.W     &a0,R12
            MOV.W     &a1+0,0(SP)
            MOV.W     &a1+2,2(SP)
            MOV.W     &a1+4,4(SP)
            MOV.W     &a1+6,6(SP)
            MOV.W     &a2+0,R13
            MOV.W     &a2+2,R14
            MOV.W     &a3+0,8(SP)
            MOV.W     &a3+2,10(SP)
```

在 MSP430 和 MSP430X 上，仅当使用四倍字寄存器时才会发生空洞和回填。如果仅使用单个寄存器和寄存器对，实参寄存器按顺序使用，则不会发生回填和空洞。

3.3.5 编译器辅助函数的特殊约定

对于 MSP430 和 MSP430X，为了提高效率，编译器会对采用两个 64 位实参（“long long int”和“double”算术）的某些编译器辅助函数使用特殊的调用约定。

在这种特殊情况下，编译器允许两个四倍字寄存器用于实参传递：R8::R11 和 R12::R15。这是 R8 到 R11 用作实参寄存器的唯一情况。第一个实参在 R8::R11 中传递，第二个实参在 R12::R15 中传递。像往常一样，返回值位于 R12::R15 中。

有关使用修改版约定的辅助函数，请参阅节 6.3。

MSP430 和 MSP430X 示例：

C 源代码

```
long long a1, a2;
long long func(void)
{
    return a1 / a2;
}
```

以小代码、小数据模型编译：

```
func:
        PUSH.W    R10; R10 is caller-saved!
        PUSH.W    R9    ; R9  is caller-saved!
        PUSH.W    R8    ; R8  is caller-saved!
        MOV.W     &a1+0,R8
        MOV.W     &a1+2,R9
        MOV.W     &a1+4,R10
        MOV.W     &a1+6,R11
        MOV.W     &a2+0,R12
        MOV.W     &a2+2,R13
        MOV.W     &a2+4,R14
        MOV.W     &a2+6,R15
        CALL      #__mspabi_divlli
        BR        #__mspabi_func_epilog_3
```

3.3.6 C++ 实参传递

在 C++中，“this”指针作为隐式第一个实参传递给 R12 中的非静态成员函数。（如果非静态成员函数通过引用返回结构，则顺序为“&struct”、“this”。）

3.3.7 传递结构体和联合体

结构体和联合体通过引用传递，如节 3.5 所述。

3.3.8 未在寄存器中传递的实参的栈布局

任何未在寄存器中传递的实参都会以递增的地址（从 0(SP) 开始）放置在栈上。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，结构体的栈对齐都是大于或等于其大小的 2 最小幂。这样一来，即使类型本身未足够严格地对齐，也允许对齐加载实参，而大小为 32 且包含字符数组的结构可能就是这种情况。
- 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。

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。

在栈上传递的实参之间可能会出现对齐“孔”，但不会出现“回填”。

3.3.9 帧指针

MSP430 不使用帧指针。这有效地将单个调用帧限制为 0x7fff 字节，它是任何指令都支持的最小 SP 偏移量。

3.4 返回值

函数返回值根据其类型和大小，位于与通常的第一个实参寄存器相同的寄存器中。

对于 MSP430 和 MSP430X，返回值位于 R12、R12:13 或 R12::R15 中。适于单个 CPU 寄存器的类型的返回值位于 R12 中；这包括指针类型。32 位返回值位于 R12:R13 中。64 位返回值位于 R12::R15 中。LSW 始终位于 R12 中。

结构体和联合体通过引用传递。

3.5 通过引用传递并返回的结构体和联合体

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），被调用者可能需要制作自己的指向对象拷贝。在某些情况下，被调用者不需要制作副本，如被调用者是叶子函数并且它不修改指向对象。

调用者必须传递一个附加实参，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。

此附加实参作为隐式第一个实参在第一个实参寄存器中传递。被调用方通过将对象复制到给定地址来返回对象。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，则可以通过在第一个实参寄存器中传递 &s 来编译赋值 s = f()。

示例

C 源代码：

```
struct S { char big[100]; } g;
struct S accepts_and_returns_struct(struct S s)
{
    s.big[0] = 1;
    return s;
}
void caller(void)
{
    struct S w;
    w.big[0] = 0;
    g = accepts_and_returns_struct(w);
}
```

“降低”C 代码：（高级别 C 代码转换为低级别 C 代码）

```
struct S { char big[100]; } g;
void accepts_and_returns_struct(struct S *dst, struct S *sptr)
{
    struct S s;
    s = *sptr;
    s.big[0] = 1;
    if (dst) *dst = s;
}
void caller(void)
{
    struct S w;
    w.big[0] = 0;
    accepts_and_returns_struct(&g, &w);
}
```

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 6.3。

3.7 已见函数的暂存寄存器

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 6.3 中所述具有特殊约定时，就会出现这种情况。

3.8 _ _mspabi_func_epilog 辅助函数

仅在 MSP430 上，_ _mspabi_func_epilog 辅助函数集可以减小代码量。每个函数均执行典型的 POP 和 RET 函数收尾程序序列。通过将典型的 POP 和 RET 收尾程序序列替换为其中一个函数的分支，可以减小代码量。每个函数均以其恢复的连续寄存器（以 R10 结尾）数量命名。预期的实现为：

```
__mspabi_func_epilog:
      __mspabi_func_epilog_7:   POP    R4
      __mspabi_func_epilog_6:   POP    R5
      __mspabi_func_epilog_5:   POP    R6
      __mspabi_func_epilog_4:   POP    R7
      __mspabi_func_epilog_3:   POP    R8
      __mspabi_func_epilog_2:   POP    R9
      __mspabi_func_epilog_1:   POP    R10
                                RET
```

MSP430X 上不需要这些函数，因为其上使用了多寄存器 POP 指令。

3.9 中断函数

中断函数必须保存所有使用的寄存器，即使是通常视为由被调用方保存的寄存器也不例外，但特殊用途寄存器 PC、SP 和 SR 除外。

中断将 SR 和 PC 寄存器压入栈并分支到中断处理程序。如要从中断函数返回，该函数必须执行特殊指令 RETI，该指令恢复 SR 寄存器并分支到发生中断的 PC。

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。

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 4.3.2。

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）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：

- 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。

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。

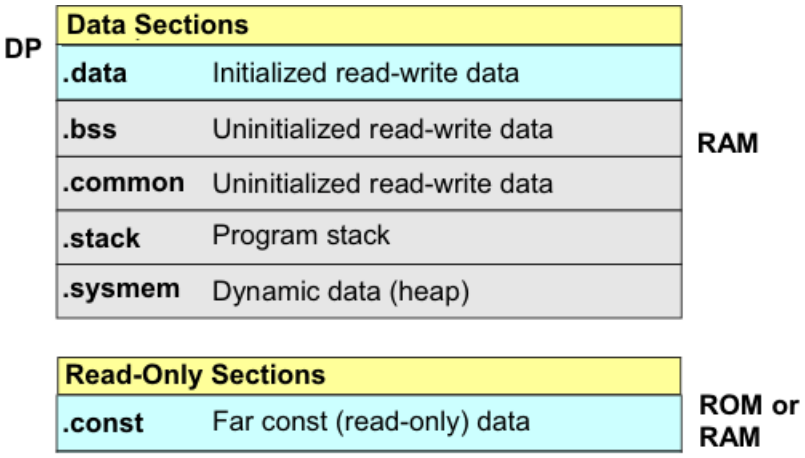


图 4-1. 数据段和数据区段（典型值）

.const 段包含只读常量。.const 段可位于只读存储器中，并且可使用绝对寻址来寻址。

.data 段包含初始化的读写数据。

.bss 段包含未初始化的读写数据。

.common 段包含链接器分配的通用块符号。这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。该段不应用于其他目的。

节 11.3.5 中列出了可由链接器命令文件放置的其他特殊段。

4.2 寻址模式

MSP 系列使用多种汇编代码寻址模式来实现数据和代码存储器模型。这里简要列出了这些模式，如需详细了解这些模式，请参阅 *MSP430x2xx 系列用户指南* (SLAU144) 的“寻址模式”一节。

寻址模式决定了使用的重定位类型。节 11.5 中讨论了重定位。

表 4-1. MSP430 寻址模式

模式名称	汇编示例	重定位类型	注释
寄存器模式	R5	不重定位	

表 4-1. MSP430 寻址模式（续）

模式名称	汇编示例	重定位类型	注释
已索引模式	X(R5)	绝对重定位	
符号模式	ADDR	PC 相对重定位	这实际上是索引模式，但以 PC 作为基址寄存器。
绝对模式(1)	&ADDR	绝对重定位	这实际上是索引模式，但以 SR 作为基址寄存器。以这种方式使用时，SR 被视为 0。
间接寄存器模式	@R5	不重定位	
间接自动递增模式	@R5+	不重定位	
立即模式	#X	绝对重定位	这实际上是间接自动递增模式，但与 PC 寄存器相关。

(1) 在本用户指南中，“绝对模式”这个术语用于表示将确切地址编码在指令中的寻址模式。当重定位用于绝对模式时，它是绝对重定位，但绝对重定位可用于其他寻址模式。

4.3 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）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。

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

4.3.1 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：绝对和 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、 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。

4.3.1.1 绝对寻址

以下指令会将地址“sym”的内容加载到 R5 中。

```
        MOV.W    &sym, R5
```

由于这种寻址模式会对地址进行编码，因此它与位置相关。

4.3.1.2 符号寻址

符号寻址是使用与 PC 寄存器相对的索引寻址来执行。（请参阅节 4.2。）假定数据位于距访问数据的代码的（链接时）恒定偏移处。无论是寻址代码还是数据，寻址机制都是相同的。符号寻址有时称为 PC 相对寻址。

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)。

以下指令会将“sym”的内容加载到 R5 中。

```
        MOV.W    sym, R5
```

4.3.1.3 立即寻址

以下指令使用立即寻址。

```
        MOV.W #sym, R5  ; put the address of sym into R5
        MOV.W #123, R5  ; put the value 123 into R5
```

4.3.2 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 寻址。

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。

4.3.2.1 放置的抽象约定

一些 MSP 系列器件支持多种数据和代码存储器模型。这些模型因允许的对象和指针大小而异。编译器使用不同的指令和重定位来实现这些模型。节 1.9 概述了 MSP 系列支持的代码和数据模型。要了解指针如何因存储器模型而异，请参阅节 2.4。

MSP 系列不区分“near”地址和“far”地址

除了影响指针大小的模型之外，数据模型还会影响 size_t 和 ptrdiff_t 的类型，因而决定允许的最大对象大小。

对于具有小型数据模型的 MSP430 和 MSP430X，所有存储器位置均可通过 16 位地址进行访问。

对于具有受限数据模型和大型数据模型的 MSP430X，大多数地址仍然可以通过 16 位地址访问，并且第 16 位以上的所有位均为零。有些指令（例如 CALL）仅处理 16 位指针，因此这些指令只能在较低的 64KB 存储器空间中使用；它们会忽略 20 位指针的高 4 位。其他指令（例如 CALLA）使用完整的 20 位地址；这些指令可以到达整个存储器空间。

MSP430X 器件没有任何超过 64KB 边界的可写存储器。对于这些器件，即使使用受限数据模型或大型数据模型，也只会将常量数据放置在 64KB 以上。编译器可以利用这一知识生成更高效的代码。这由 --near_data 编译器选项控制。如果指定 --near_data=globals，此选项告诉编译器所有全局读取/写入数据必须位于存储器的前 64K 中。这是默认行为。如果指定了 --near_data=none，编译器不能依赖这个假设来生成更高效的代码。

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。

ABI 建立了变量到段的传统赋值。变量的赋值是初始化类别的函数，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 4-2 列出了传统的段赋值：

表 4-2. 变量到段的传统赋值

	初始化类别		
	未初始化	已初始化	常量
	.bss	.data	.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。

4.3.2.2 寻址的抽象约定

所有 MSP430 变量都位于绝对寻址（与位置相关）的寻址范围内。不支持与位置无关的寻址。

4.3.3 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 14 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。

4.4 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 (R1) 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 指向当前帧中的第一个地址。也就是说，该函数可以读取/写入 0(SP)。

节 4.5 更详细地介绍了堆栈约定和本地帧结构。

4.5 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 指向该函数帧内的最低地址位置。也就是分配 0(SP)，但不分配 -1(SP)。

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 不使用帧指针，因此单个调用帧仅限于 0xffff 字节。

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。但是，异常处理栈展开指令假设没有可用帧指针。

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。

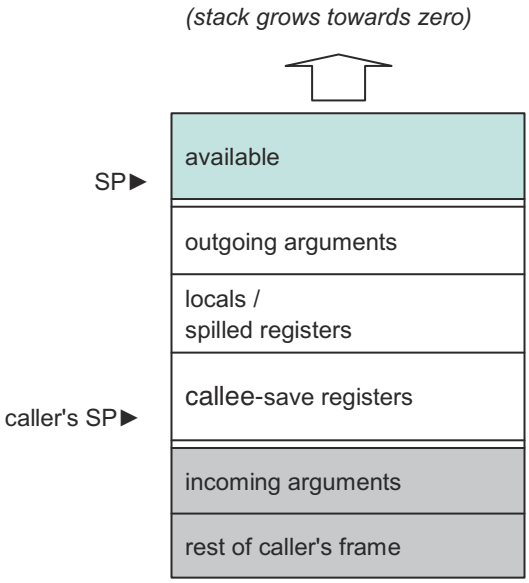


图 4-2. 局部帧布局

在分配帧之前，SP 指向返回地址（用于中断函数的 SR）。

4.5.1 栈对齐

对于 MSP430 和 MSP430X，SP 为 2 字节对齐。

栈必须始终保持正确对齐，包括在帧分配和释放期间。这意味着对 SP 的每次原子调整都必须是所需对齐的倍数。

即使在手工编码的汇编函数（包括中断函数）中，也必须始终保持对齐。手工编码的汇编函数可以调用 C 兼容函数或其调用，并且 C 兼容处理程序可能随时发生中断。

4.5.2 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：

- MSP430X 具有 PUSHM 和 POPM 指令，可使一定数量的连续寄存器入栈或出栈。这些指令比针对每个寄存器使用单独的入栈和出栈指令更高效。这些指令按以下顺序使寄存器入栈，从帧底部（最高地址）编号最大的入栈寄存器开始：**R10 R9 R8 R7 R6 R5 R4**。POPM 当然会以相反顺序使它们出栈。
- **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*（PUSHM 顺序）对位矢量进行编码，因此顺序相同。

当针对 MSP430X 进行编译或支持异常处理时，编译器始终按照 PUSHM 相对顺序保存寄存器，从帧的底部（最低地址）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。

4.6 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。

This page intentionally left blank.

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。指令具有从 16 位到 64 位的可变长度，必须为 16 的精确倍数。

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
- 填充切换表

构成地址的基本方法有三种：绝对寻址、符号寻址和立即寻址。绝对寻址和立即寻址与位置相关。符号（相对于 PC）寻址与位置无关。节 4.2 *MSP430x2xx 系列用户指南* (SLAU144) 的“寻址模式”一节中简要列出并详细介绍了这些模式。

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：

对于具有小型代码模型的 MSP430/MSP430X，以下 CALL 指令使用绝对寻址并调用地址存储在 func 位置的函数。

```
        CALL    &func
```

对于具有大型代码模型的 MSP430X，以下 CALLA 指令等同于之前的 CALL 指令。

```
        CALLA   &func
```

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 符号寻址

符号寻址是使用与 PC 寄存器相对的索引（绝对）寻址来执行。所需地址是有效 PC 和索引寻址模式偏移量之和。此偏移量在汇编代码中不可见。无论是寻址代码还是数据，机制都是相同的。符号寻址有时称为 *PC 相对寻址*。

对于具有小型代码模型的 MSP430/MSP430X，以下 CALL 指令使用符号寻址，并调用地址存储在“sym”位置的函数。

```
        CALL    sym
```

对于具有大型代码模型的 MSP430X，以下 CALLA 指令等同于之前的 CALL 指令。

```
        CALLA   sym
```

5.1.3 立即寻址

这是通常用于实现函数调用的寻址模式。可使用立即寻址来将函数“func”的地址加载到 R5 中。

```
        MOV.W #func, R5    ; immediate
```

对于具有小型代码模型的 MSP430/MSP430X，以下 CALL 指令使用立即寻址并调用从地址“func”开始的函数。

```
        CALL      #func
```

对于具有大型代码模型的 MSP430X，以下 CALLA 指令等同于之前的 CALL 指令。

```
        CALLA     #func
```

5.2 分支

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

对于 MSP430 和 MSP430X 小型代码模型，分支和调用使用具有 16 位偏移量的 BR 和 CALL 指令。

对于 MSP430X 大型代码模型，分支和调用使用具有 20 位偏移量的 BRA 和 CALLA 指令。

5.3 调用

函数调用通过调用专用 CALL 指令进行，该指令将返回地址压入函数调用堆栈并跳转到被调用函数。被调用函数通过执行专用 RET 指令返回，该指令从堆栈中弹出返回地址并跳转到该地址。

5.3.1 直接 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。

5.3.2 Far Call Trampoline

可通过直接调用到达 MSP430 和 MSP430X 的整个地址空间，因此未使用蹦床函数。

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：

```
          CALL  R5                    ; indirect call
```

This page intentionally left blank.

章节 6

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。

6.1 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。

MSP430 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。

6.2 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 _ _MSP430 命名。具有此前缀的任何标识符都将保留供 ABI 使用。

辅助函数遵守标准调用约定，节 6.3 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 6-1 中的函数根据 C 语言的转会规则以及节 6.1 指定的浮点行为在各种浮点和 int 格式之间进行转换。

表 6-1. MSP430 浮点和 int 转换

签名	说明
float32 _ _**MSP430_cvtdf**(float64 x);	将双精度浮点型转换为单精度浮点型
float64 _ _**MSP430_cvtfd**(float32 x);	将单精度浮点型转换为双精度浮点型
int16 _ _**MSP430_fixdi**(float64 x);	将双精度浮点型转换为 int
int32 _ _**MSP430_fixdli**(float64 x);	将双精度浮点型转换为 long int
int64 _ _**MSP430_fixdlli**(float64 x);	将双精度浮点型转换为 long long int
uint16 _ _**MSP430_fixdu**(float64 x);	将双精度浮点型转换为 unsigned int
uint32 _ _**MSP430_fixdul**(float64 x);	将双精度浮点型转换为 unsigned long int
uint64 _ _**MSP430_fixdull**(float64 x);	将双精度浮点型转换为 unsigned long long int
int16 _ _**MSP430_fixfi**(float32 x);	将单精度浮点型转换为 int
int32 _ **MSP430_fixfl**i (float32 x) ;	将单精度浮点型转换为 long int
int64 _ _**MSP430_fixflli**(float32 x);	将单精度浮点型转换为 long long int
uint16 _ _**MSP430_fixfu**(float32 x);	将单精度浮点型转换为 unsigned int
uint32 _ _**MSP430_fixful**(float32 x);	将单精度浮点型转换为 unsigned long int
uint64 _ _**MSP430_fixfull**(float32 x);	将单精度浮点型转换为 unsigned long long int
float64 _ _**MSP430_fltid**(int16 x);	将 int 转换为双精度浮点型
float32 _ _**MSP430_fltif**(int16 x);	将 int 转换为单精度浮点型
float64 _ _**MSP430_fltlid**(int32 x);	将 long int 转换为双精度浮点型
float32 _ _**MSP430_fltlif**(int32 x);	将 long int 转换为单精度浮点型
float64 _ _**MSP430_fltud**(uint16 x);	将 unsigned int 转换为双精度浮点型

表 6-1. MSP430 浮点和 int 转换（续）

签名	说明
float32 _ _**MSP430_fltuf**(uint16 x);	将 unsigned int 转换为单精度浮点型
float64 _ _**MSP430_fltuld**(uint32 x);	将 unsigned long int 转换为双精度浮点型
float32 _ _**MSP430_fltulf**(uint32 x);	将 unsigned long int 转换为单精度浮点型

表 6-2 中的函数根据 C 语言的语义和节 6.1 指定的浮点行为执行浮点比较。

如果 x 小于 y，则 MSP430_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。

其余的比较函数当前不受支持，但保留了名称以供将来使用。

表 6-2. MSP430 浮点比较

签名	说明
int16 _ _**MSP430_cmpd**(float64 x, float64 y);	双精度比较
int16 _ _**MSP430_cmpf**(float32 x, float32 y);	单精度比较
int16 _ _**MSP430_eqd**(float64 x, float64 y);	双精度比较：x == y（当前不受支持）
int16 _**MSP430_geq**dfloat64 x、float64 y)；	双精度比较：x >= y（当前不受支持）
int16 _ _**MSP430_gtrd**(float64 x, float64 y);	双精度比较：x > y（当前不受支持）
int16 _ _**MSP430_leqd**(float64 x, float64 y);	双精度比较：x <= y（当前不受支持）
int16 _ _**MSP430_lssd**(float64 x, float64 y);	双精度比较：x < y（当前不受支持）
int16 _ _**MSP430_neqd**(float64 x, float64 y);	双精度比较：x!= y（当前不受支持）

表 6-3 中的函数根据 C 语言的语义和节 6.1 指定的浮点行为执行浮点运算。

表 6-3. MSP430 浮点算术

签名	说明
float64 _ _**MSP430_addd**(float64 x, float64 y);	双精度与双精度相加
float32 _ _**MSP430_addf**(float32 x, float32 y);	单精度与单精度相加
float64 _ _**MSP430_divd**(float64 x, float64 y);	双精度除以双精度
float32 _ _**MSP430_divf**(float32 x, float32 y);	单精度除以单精度
float64 _ _**MSP430_mpyd**(float64 x, float64 y);	双精度与双精度相乘
float32 _ _**MSP430_mpyf**(float32 x, float32 y);	单精度与单精度相乘
float64 _ _**MSP430_subd**(float64 x, float64 y);	从双精度中减去双精度
float32 _ _**MSP430_subf**(float32 x, float32 y);	从单精度中减去单精度
float64 _ _**MSP430_negd**(float64 x);	将双精度数取反
float32 _ _**MSP430_negf**(float32 x);	将单精度数取反

表 6-4 中的整数算术函数根据 C 语言的语义进行运算。

表 6-4. MSP430 整数乘法、除法和余数

签名	说明
	乘法
int16 _ _**MSP430_mpyi**(int16 x, int16 y);	int 与 int 相乘。
int16 _ _**MSP430_mpyi_hw**(int16 x, int16 y);	int 与 int 相乘。使用硬件 MPY16。
int16 _ _**MSP430_mpyi_f5hw**(int16 x, int16 y);	int 与 int 相乘。使用硬件 MPY32（F5xx 器件及更高版本）。
int32 _ _**MSP430_mpyl**(int32 x, int32 y);	long 与 long 相乘。
int32 _ _**MSP430_mpyl_hw**(int32 x, int32 y);	long 与 long 相乘。使用硬件 MPY16。
int32 _ _**MSP430_mpyl_hw32**(int32 x, int32 y);	long 与 long 相乘。使用硬件 MPY32（F4xx 器件）。
int32 _ _**MSP430_mpyl_f5hw**(int32 x, int32 y);	long 与 long 相乘。使用硬件 MPY32（F5xx 器件及更高版本）。
int64 _ _**MSP430_mpyll**(int64 x, int64 y);	long long 与 long long 相乘。
int64 _ _**MSP430_mpyll_hw**(int64 x, int64 y);	long long 与 long long 相乘。使用硬件 MPY16。
int64 _ _**MSP430_mpyll_hw32**(int64 x, int64 y);	long long 与 long long 相乘。使用硬件 MPY32（F4xx 器件）。
int64 _ _**MSP430_mpyll_f5hw**(int64 x, int64 y);	long long 与 long long 相乘。使用硬件 MPY32（F5xx 器件及更高版本）。
int32 _ _**MSP430_mpysl**(int16 x, int16 y);	int 与 int 相乘；结果为 long。
int32 _ _**MSP430_mpysl_hw**(int16 x, int16 y);	int 与 int 相乘；结果为 long。使用硬件 MPY16。
int32 _ _**MSP430_mpysl_f5hw**(int16 x, int16 y);	int 与 int 相乘；结果为 long。使用硬件 MPY32（F5xx 器件及更高版本）。
int64 _ _**MSP430_mpysll**(int32 x, int32 y);	long 与 long 相乘；结果为 long long。
int64 _ _**MSP430_mpysll_hw**(int32 x, int32 y);	long 与 long 相乘；结果为 long long。使用硬件 MPY16。
int64 _ _**MSP430_mpysll_hw32**(int32 x, int32 y);	long 与 long 相乘；结果为 long long。使用硬件 MPY32（F4xx 器件）。
int64 _ _**MSP430_mpysll_f5hw**(int32 x, int32 y);	long 与 long 相乘；结果为 long long。使用硬件 MPY32（F5xx 器件及更高版本）。
uint32 _ _**MSP430_mpyul**(uint16 x, uint16 y);	将 unsigned int 与 unsigned int 相乘；结果为 unsigned long。
uint32 _ _**MSP430_mpyul_hw**(uint16 x, uint16 y);	将 unsigned int 与 unsigned int 相乘；结果为 unsigned long。使用硬件 MPY16。
uint32 _ _**MSP430_mpyul_f5hw**(uint16 x, uint16 y);	将 unsigned int 与 unsigned int 相乘；结果为 unsigned long。使用硬件 MPY32（F5xx 器件及更高版本）。
uint64 _ _**MSP430_mpyull**(uint32 x, uint32 y);	将 unsigned long 与 unsigned long 相乘；结果为 unsigned long long。
uint64 _ _**MSP430_mpyull_hw**(uint32 x, uint32 y);	将 unsigned long 与 unsigned long 相乘；结果为 unsigned long long。使用硬件 MPY16。
uint64 _ _**MSP430_mpyull_hw32**(uint32 x, uint32 y);	将 unsigned long 与 unsigned long 相乘；结果为 unsigned long long。使用硬件 MPY32（F4xx 器件）。
uint64 _ _**MSP430_mpyull_f5hw**(uint32 x, uint32 y);	将 unsigned long 与 unsigned long 相乘；结果为 unsigned long long。使用硬件 MPY32（F5xx 器件及更高版本）。
	除法
int16 _ _**MSP430_divi**(int16 x, int16 y);	int 除以 int。
int32 _ _**MSP430_divli**(int32 x, int32 y);	long 除以 long。
int64 _ _**MSP430_divlli**(int64 x, int64 y);	long long 除以 long long。
uint16 _ _**MSP430_divu**(uint16 x, uint16 y);	unsigned lint 除以 unsigned lint。
uint32 _ _**MSP430_divlu**(uint32 x, uint32 y);	unsigned long 除以 unsigned long。
uint64 _ _**MSP430_divllu**(uint64 x, uint64 y);	unsigned long long 除以 unsigned long long。
	余数（模数）
int16 _ _**MSP430_remi**(int16 x, int16 y);	int 除以 int 的余数 (x mod y)
int32 _ _**MSP430_remli**(int32 x, int32 y);	long 除以 long 的余数 (x mod y)
int64 _ _**MSP430_remlli**(int64x. int64 y);	long long 除以 long long 的余数 (x mod y)

表 6-4. MSP430 整数乘法、除法和余数 （续）

签名	说明
uint16 _ _**MSP430_remu**(uint16 x, uint16 y);	unsigned int 除以 unsigned int 的余数 (x mod y)
uint32 _ _**MSP430_remul**(uint32, uint32);	unsigned long 除以 unsigned long 的余数 (x mod y)
uint64 _ _**MSP430_remull**(uint64, uint64);	unsigned long long 除以 unsigned long long 的余数 (x mod y)

表 6-5 列出了按位运算符函数。这些函数由 MSP430 和 MSP430X 使用。

循环左移运算不带进位执行。最高有效位被移至最低有效位 (LSB)，所有其他位左移。对于 long long 数据类型，没有循环函数。

逻辑左移运算与算术左移相同；一个零被移入 LSB。

算术右移运算将最高有效位 (MSB) 向右移动并将旧 MSB 复制到新 MSB 中。这会将符号保留在有符号值中。LSB 被丢弃。

逻辑右移运算在 MSB 中插入值 0 并丢弃 LSB。

接受第二个实参的函数中的移位或循环计数不能小于零，即使按 signed int 处理它也不例外。

表 6-5. MSP430/MSP430X 按位运算

签名	说明
	旋转
uint16 _ _**MSP430_rlli**(uint16 x, int16 n);	将 int 的位向左循环移动 n 位，其中 n 最多可以是 16。
uint16 _ _**MSP430_rlli_1**(uint16 x); ... uint16 _ _**MSP430_rlli_15**(uint16 x);	将 int 的位向左循环移动指定的位数。
uint32 _ _**MSP430_rlll**(uint32 x, int16 n);	将 long 的位向左循环移动 n 位，其中 n 最多可以是 32。
	逻辑左移
uint16 _ _**MSP430_slli**(uint16 x, int16 n);	对 int 执行逻辑左移。移动 n 位，其中 n 最多可以是 16。
uint16 _ _**MSP430_slli_1**(uint16 x); ... uint16 _ _**MSP430_slli_15**(uint16 x);	对 int 执行逻辑左移。向左移位指定的位数。
uint32 _ _**MSP430_slll**(uint32 x, int16 n);	对 long 执行逻辑左移。移动 n 位，其中 n 最多可以是 32。
uint32 _ _**MSP430_slll_1**(uint32 x); ... uint32 _ _**MSP430_slll_15**(uint32 x);	对 long 执行逻辑左移。向左移位指定的位数。高于 15 的移位数使用 _ _MSP430_slll。
uint64 _ _**MSP430_sllll**(uint64 x, int16 n);	对 long long 执行逻辑左移。移动 n 位，其中 n 最多可以是 64。
	算术右移
int16 _ _**MSP430_srai**(int16 x, int16 n);	对 int 执行算术右移。移动 n 位，其中 n 最多可以是 16。
int16 _ _**MSP430_srai_1**(int16 x); ... int32 _ _**MSP430_srai_15**(int16 x);	对 int 执行算术右移。移动指定的位数。
int16 _ _**MSP430_sral**(int32 x, int16 n);	对 long 执行算术右移。移动 n 位，其中 n 最多可以是 32。
int32 _ _**MSP430_sral_1**(int32 x); ... int32 _ _**MSP430_sral_15**(int32 x);	对 long 执行算术右移。移动指定的位数。高于 15 的移位数使用 _ _MSP430_sral。
int64 _ _**MSP430_srall**(int64 x, int16 n);	对 long long 执行算术右移。移动 n 位，其中 n 最多可以是 64。
	逻辑右移
uint16 _ _**MSP430_srli**(uint16 x, int16 n);	对 int 执行逻辑右移。移动 n 位，其中 n 最多可以是 16。
uint16 _ _**MSP430_srli_1**(uint16 x); ... uint16 _ _**MSP430_srli_15**(uint16 x);	对 int 执行逻辑右移。向右移动指定的位数。
uint32 _ _**MSP430_srll**(uint32 x, int16 n);	对 long 执行逻辑右移。移动 n 位，其中 n 最多可以是 32。

表 6-5. MSP430/MSP430X 按位运算（续）

签名	说明
uint32 _ _**MSP430_srll_1**(uint32 x); ... uint32 _ _**MSP430_srll_15**(uint32 x);	对 long 执行逻辑右移。向右移动指定的位数。高于 15 的移位数使用 _ _MSP430_srll。
uint64 _ _**MSP430_srlll**(uint64 x, int16 n);	对 long long 执行逻辑右移。移动 n 位，其中 n 最多可以是 64。

表 6-6 中的辅助函数可优化恢复被调用者保存的寄存器。MSP430 使用这些函数，但 MSP430X 不使用这些函数。

表 6-6. MSP430 收尾程序辅助函数

签名	说明
void **__MSP430_epilog_1**(void);	弹出 R10 并返回。
void **__MSP430_epilog_2**(void);	弹出 R10 到 R9 并返回。
void **__MSP430_epilog_3**(void);	弹出 R10 到 R8 并返回。
void **__MSP430_epilog_4**(void);	弹出 R10 到 R7 并返回。
void **__MSP430_epilog_5**(void);	弹出 R10 到 R6 并返回。
void **__MSP430_epilog_6**(void);	弹出 R10 到 R5 并返回。
void **__MSP430_epilog_7**(void);	弹出 R10 到 R4 并返回。

下面的章节将介绍表 6-7 中的其他辅助函数。

表 6-7. MSP430 其他辅助函数

签名	说明
void **_abort_msg**(const char *string);	报告断言失败

_abort_msg

生成函数 _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。

6.3 辅助函数的特殊寄存器约定

以下函数使用 2 个 64 位值：

- __mspabi_mpyll
- __mspabi_divull
- __mspabi_remull
- __mspabi_divlli
- __mspabi_remlli
- __mspabi_srall
- __mspabi_srlll
- __mspabi_sllll
- __mspabi_addd
- __mspabi_subd
- __mspabi_mpyd
- __mspabi_divd
- __mspabi_cmpd

对于 MSP430 和 MSP430X，这些函数具有特殊的调用约定。第一个实参在 R8::R11 中，第二个实参在 R12::R15 中。即：

```
First argument word 0 (LSW): R8
First argument word 1      : R9
First argument word 2      : R10
First argument word 3 (MSW): R11
Second argument word 0 (LSW): R12
Second argument word 1      : R13
Second argument word 2      : R14
Second argument word 3 (MSW): R15
```

此外，对于许多辅助函数，库实现必须注意不要修改某些寄存器。

有关寄存器中存储的实参规模的更多信息，请参阅节 3.3.5。

6.4 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 6-8. 保留的浮点分类辅助函数

签名	说明
int32 _ **_MSP430_isfinite**(float64 x);	如果 x 是一个可表示的值，则为真
int32 _ **_MSP430_isfinitef**(float32 x);	如果 x 是一个可表示的值，则为真
int32 _ **_MSP430_isinf**(float64 x);	如果 x 表示“无穷大”，则为真
int32 _ **_MSP430_isinff**(float32 x);	如果 x 表示“无穷大”，则为真
int32 _ **_MSP430_isnan**(float64 x);	如果 x 表示“非数字”，则为真
int32 _ **_MSP430_isnanf**(float32 x);	如果 x 表示“非数字”，则为真
int32 _ **_MSP430_isnormal**(float64 x);	如果 x 未去规范化，则为真
int32 _ **_MSP430_isnormalf**(float32 x);	如果 x 未去规范化，则为真
int32 _ **_MSP430_fpclassify**(float64 x);	将浮点值分类
int32 _ **_MSP430_fpclassifyf**(float32 x);	将浮点值分类

函数 **_ _MSP430_fpclassify** 用于将浮点数分类。运行如下：

```
int32 __MSP430_fpclassify(float64 x)
{
   if      (isnormal(x)) return 3;
   else if (isinf(x))    return 1;
   else if (isnan(x))    return 2;
   else                  return 4;
}
```

This page intentionally left blank.

章节 7

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 MSP430 ABI。

7.1 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 11.4.4 中列出的任何符号或节 11.1 中列出的任何带有前缀的符号。

7.2 <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 _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 6.2。

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

7.3 <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 MSP 工具集支持 C99 复数并提供此头文件。

7.4 <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）

7.5 <errno.h> 实现

errno 是一个全局 int，定义如下：

```
extern int errno;
```

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 0x21
#define ERANGE 0x22
#define EILSEQ 0x58
#define ENOENT 0x2
#define EFPOS 0x98
```

7.6 <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。

7.7 <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。

7.8 <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。

7.9 <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
```

7.10 <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：

MSP430/430X 顺序：（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;
```

7.11 <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。

7.12 <setjmp.h> 实现

jmp_buf 的类型和大小在 setjmp.h 中定义

在小代码和小数据模型组合下，jmp_buf 的大小和对齐与由 9 个“int”组成的数组（即 16 位 * 9）相同。对于所有其他代码和数据模型组合，jmp_buf 的大小和对齐与由 9 个“long”组成的数组（即 32 位 * 9）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。

7.13 <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
```

7.14 <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。

7.15 <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。

7.16 <stddef.h> 实现

节 2.4 中定义了 stddef.h 中定义的每种类型的大小和对齐。

7.17 <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。

7.18 <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。MSP430 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 long。

7.19 <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。

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。

7.20 <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。

7.21 <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。

7.22 <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。

7.23 <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。。

7.24 <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;
```

This page intentionally left blank.

章节 8

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。MSP430 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。

8.1 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 MSP 系列 ，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。

8.2 导出模板 (GC++ABI 1.4.2)

ABI 当前未指定导出模板。

8.3 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 2.8 中介绍了这些字段。

8.4 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 16 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 16 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;
}
```

8.5 构造函数返回值 (GC++ABI 3.1.5)

MSP430 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 MSP430 ABI 中，它们被声明为返回 void *（对应于 'this'）的函数的指针。

8.6 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 16 位容器的第一个字节中。请参阅节 8.4。

8.7 控制对象构造顺序 (GC++ ABI 3.3.4)

MSP430 ABI 没有指定控制对象构造的机制。

8.8 还原器 API (GC++ABI 3.4)

MSP430 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。

8.9 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。MSP430 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。

8.10 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。MSP430 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。

8.11 回溯表位置 (GC++ABI 5.3)

本文档的章节 9 中介绍了异常处理。

This page intentionally left blank.

章节 9

异常处理

MSP430 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。

9.1 概述

MSP430 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 MSP430 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。

9.2 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_MSP430_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。

9.3 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。

9.3.1 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

9.3.2 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)	

9.3.3 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 9.4 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'

9.4 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 9-1 中列出了 ABI 规定的个性例程。

9.4.1 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。

9.4.2 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)			

9.4.3 个性化例程

MSP430 具有以下 ABI 指定的个性化例程。这些例程与 ARM EABI 的格式相同。下表列出了个性化例程及其 PR 索引。

表 9-1. MSP430 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)	_ _MSP430_unwind_cpp_pr0	最多 3 个一字节指令	16	
0001	PR1 (Lu16)	_ _MSP430_unwind_cpp_pr1	无限个一字节指令	16	
0010	PR2 (Lu32)	_ _MSP430_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

使用紧凑模型 EXTAB 条目时，可重定位文件必须以 R_MSP430_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。

9.5 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 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*。

9.5.1 通用序列

理论上，所有展开序列都采用以下形式：

1. 恢复 SP (SP += 常量)
2. （可选）恢复被调用者保存的寄存器（reg1 := SP[0]；reg2 := SP[-1]，以此类推）
3. 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 将通过递增一个常量来恢复。 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。

步骤 2：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 4.5.2）恢复，从（旧的）SP 指向的位置开始，并移动到较低的地址。

步骤 3：返回

每个展开序列以隐式或显式“RET ”结束，这表示当前帧的展开已完成。

9.5.2 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 ”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 9-2 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 9-2. 堆栈展开指令

编码	指令	说明
0xxx xxxx	POP bitmask (R10, R9, R8, R7, R6, R5, R4) + RET	恢复被调用者保存的寄存器并返回。
11kk kkkk	SP += (kkkkkk << 1) + 2 [0x02-0x80]	递增，增量为一个小常数
1000 0001 kkkk	SP += (ULEB128 << 1) + 0x102 [0x102-max]	递增，增量为大常数
1000 0000 0000 0000	CANTUNWIND	函数不能展开

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。

POP + RET

POP+RET 指令指定一个位掩码，用于表示由此函数逻辑程序保存的寄存器。这些寄存器必须按顺序弹出，从 R4 开始并继续到 R10。完成后，不再有展开指令。如果位掩码中的位均未设置，则这只是一条 RET 指令。

小增量

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

9.6 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			

9.6.1 类型标识符编码

Catch 描述符和 FESPEC 描述符 (节 9.6.5) 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。

9.6.2 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 或 PR1。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 9-1. 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 9-2. 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

9.6.3 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）。

9.6.4 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。

9.6.5 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_MSP430_none 重定位。

9.7 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 .MSP430.exidx 且类型为 SHT_MSP430_UNWIND 的段中。链接器必须将所有输入 .MSP430.exidx 段合并为一个连续的 .MSP430.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 .MSP430.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。

9.8 与非 C++ 代码交互

9.8.1 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。

9.8.2 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。

9.9 与系统功能交互

9.9.1 共享库

异常处理表可以在可执行文件内传播异常。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。

9.9.2 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。

9.9.3 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。

9.10 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。）

$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。

This page intentionally left blank.

章节 10

DWARF

MSP430 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 MSP430 特定表示部分来扩充该标准。

10.1 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 10-1 定义了从 DWARF3 寄存器编号/名称到 MSP430 寄存器的映射。

表 10-1. MSP 的 DWARF3 寄存器编号

DWARF 名称	MSP430 ISA 寄存器	说明
0 – 15	R0-R15	有关详细信息，请参阅表 3-1。

10.2 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 10-1 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，MSP430 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 (R1)。

展开表可能包括并非所有 MSP430 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。

10.3 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	MSP430 德州仪器 (TI) 的代码生成工具
GNU	GNU 编译器套装 (GCC)

10.4 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 MSP430 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 10-2 为 MSP430 定义了 TI 的供应商特定 DIE 标签。表 10-2 定义了 TI 的供应商特定属性。

表 10-2. 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 10-3. 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_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_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。

This page intentionally left blank.

章节 11

ELF 目标文件 (处理器补充)

MSP430 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 MSP430 处理器特定补充。本文档的 章节 12 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。

11.1 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 11-1 列出了当前注册供应商及其首选的简写名称。

表 11-1. 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。
mspabi，_ _mspabi	适用于 MSP430 EABI 指定的符号的通用命名空间。
MSP430	适用于 MSP430 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 MSP430 的段类型为 SHT_MSP430_EXIDX、C2000 的段类型为 SHT_C2000_EXIDX 等。

11.2 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 MSP430 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 11-2 指定了将用于 MSP430 目标文件的值。

表 11-2. 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
EI_VERSION	EV_CURRENT	1	
EI_ABIVERSION		0	

EI_OSABI 字段应为 ELFOSABI_NONE，除非由特定平台的约定覆盖。MSP 系列的任何平台都不会覆盖 EI_OSABI 字段的默认设置；其值始终为 ELFOSABI_NONE。

e_type

当前无 MSP430 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_MSP430 (105，0x69) 。

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，e_flags 字段无 MSP430 特定标志。

11.3 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。

11.3.1 段索引

MSP430 ABI 不定义任何特殊段索引。

11.3.2 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 11-3 中列出了组合集。

并非所有这些段类型都在 MSP430 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 MSP430 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 11-3. ELF 和 TI 段类型

名称	值	注释
SHT_MSP430_UNWIND	0x70000001	用于栈回溯的回溯函数表
SHT_MSP430_PREEMPTMAP	0x70000002	DLL 动态链接抢占映射（MSP430 不支持）
SHT_MSP430_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_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_MSP430_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 9。

SHT_MSP430_ATTRIBUTES 识别包含对象兼容性属性的段。请参阅章节 13。

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 14。

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 MSP430 上使用。

11.3.3 扩展段标头属性

对于 MSP430，可在 TI 工具链中使用以下处理器特定属性标志：

SHF_MSP_NOINIT 标识包含未初始化变量的段。NOINIT 属性只能应用于 .TI.noinit 和 .TI.persistent 段。例如：

```
    ".TI.noinit"      SHT_NOBITS      SHF_MSP_NOINIT
    ".TI.persistent" SHT_PROGBITS    SHF_MSP_NOINIT
```

链接器不应为这些段创建 .cinit 记录。

11.3.4 子段

MSP430 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 11.3.5 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。

11.3.5 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 11-4 整合了 MSP430 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 11-4. MSP430 特殊段

前缀	类型	属性
	代码段	
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
	数据段	
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.TI.noinit	SHT_NOBITS	SHF_MSP_NOINIT
.TI.persistent	SHT_PROGBITS	SHF_MSP_NOINIT
.const	SHT_PROGBITS	SHF_ALLOC
	异常处理数据段	
.MSP430.exidx	SHT_MSP430_UNWIND	SHF_ALLOC + SHF_LINK_ORDER
.MSP430.extab	SHT_PROGBITS	SHF_ALLOC
	初始化和终止段	
.init_array	SHT_INIT_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
	构建属性	
.MSP430.attributes	SHT_MSP430_ATTRIBUTES	无
	符号调试段	
.debug (1)	SHT_PROGBITS	无
	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
.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 MSP430 EABI 使用的段 (2)	
.comment		
.data1		

表 11-4. MSP430 特殊段（续）

前缀	类型	属性
.dsbt		
.dynamic		
.dynstr		
.dynsym		
.far		
.fardata		
.fardata:const		
.fini		
.fini_array		
.gnu.version		
.gnu.version_d		
.gnu.version_r		
.got		
.hash		
.init		
.interp		
.line		
.neardata		
.plt		
.preinit_array		
.rodata		
.rodata1		
.tbss		
.tdata		
.tdata1		
.TI.tls_init		

(1) 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。
(2) 这些段名称不用于 MSP430 ，后者不支持与这些段相关的功能。但是，此段名称将会保留，不应使用。

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 MSP430 EABI 使用的段**”由 System V ABI 指定，但不在 MSP430 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。

此外，.common 是链接器使用的段名称。这是一个抽象段，不是目标文件中的实际段。此名称是链接器命令文件中用于放置变量的约定。该段不应用于其他目的。

11.3.6 段对齐

包含 MSP430 代码的段必须至少为 16 字节对齐，并填充到 16 字节边界。

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）。

11.4 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，MSP430 ABI 遵循 ELF 规范。

11.4.1 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。

11.4.2 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。

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。

11.4.3 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。

MSP430 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

11.4.4 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 11-1 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 4.6 中所述)

11.4.5 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。

11.5 重定位

MSP430 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 MSP430 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。

节 4.2 简要说明了寻址模式对重定位的影响。

11.5.1 重定位类型

重定位类型在两个表中进行了说明。表 11-5 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 11-6 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 11-5 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 11-6 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。这可能与包含重定位的指令的地址不同。

表 11-5. MSP430 和 MSP430X 重定位类型

名称	值	运算	约束条件
R_MSP430_NONE	0		
R_MSP430_ABS32	1	S + A	
R_MSP430_ABS16	2	S + A	
R_MSP430_ABS8	3	S + A	
R_MSP430_PCR16	4	S + A - PC	
R_MSP430X_PCR20_EXT_SRC	5	S + A - PC	
R_MSP430X_PCR20_EXT_DST	6	S + A - PC	
R_MSP430X_PCR20_EXT_ODST	7	S + A - PC	
R_MSP430X_ABS20_EXT_SRC	8	S + A	
R_MSP430X_ABS20_EXT_DST	9	S + A	
R_MSP430X_ABS20_EXT_ODST	10	S + A	
R_MSP430X_ABS20_ADR_SRC	11	S + A	
R_MSP430X_ABS20_ADR_DST	12	S + A	
R_MSP430X_PCR16	13	S + A - PC	
R_MSP430X_PCR20_CALL	14	S + A - PC	
R_MSP430X_ABS16	15	S + A	
R_MSP430_ABS_HI16	16	S + A	仅限 Rela
R_MSP430_PREL31	17	S + A - PC	

11.5.1.1 绝对重定位

绝对重定位直接对符号的重定位地址进行编码。MSP“索引”、“立即”和“绝对”寻址模式都需要绝对重定位。名称中包含“ABS”的重定位类型全都是绝对重定位类型。

11.5.1.2 PC 相对重定位

PC 相对重定位可将地址编码为有符号的 PC 相对偏移量。MSP“符号”寻址模式需要 PC 相对重定位。名称中包含“PCR”的重定位类型是 PC 相对重定位类型。

在汇编器和链接器中，位移是相对于表 11-6 中定义的重定位的容器地址计算的，而不是指令的起始地址。由于 PC 在读取指令的各个字时会前进，因此在执行寻址模式时由硬件使用的有效 PC 值可能与重定位容器的地址不同。为了补偿这一点，汇编器必须根据差异来调整重定位加数。

11.5.1.3 数据段中的重定位

R_MSP430_ABS16/32 重定位类型直接将符号的重定位地址编码为 16 位或 32 位字段。这些重定位用于重新定位初始化数据段中的地址。未针对 R_MSP430_ABS16/32 指定字段的符号；即，这些重定位类型同时用于有符号和无符号值。它们也用于某些指令重定位，如下所示：

```
        .field X,32         ; R_MSP430_ABS32
        .field X,16         ; R_MSP430_ABS16
```

11.5.1.4 MSP430 指令的重定位

MSP430 指令仅允许 16 位字段。MSP430 指令重定位通常不会检查溢出。MSP430 指令也可以用于 MSP430X，但汇编器使用不同的重定位以便检查溢出。

R_MSP430_ABS16 和 R_MSP430_PCR16 用于 MSP430 指令。两者都不用于 MSP430X 上的指令，但 $LO16 的 R_MSP430_ABS16 除外。将 $LO16 的 R_MSP430_ABS16 用于 MSP430X 是一种特殊情况，旨在成为 32 位立即加载的一半；另一半是 $HI16 (R_MSP430_ABS_HI16)。请参阅节 11.5.1.5 中 R_MSP430_ABS_HI16 的说明。

R_MSP430_ABS16 用于绝对、索引和立即寻址模式，但仅限于 MSP430：

```
        ADD.W #X, R5      ; R_MSP430_ABS16
        ADD.W &X, R5      ; R_MSP430_ABS16
        ADD.W R5, &X      ; R_MSP430_ABS16
        ADD.W K(R4), R5   ; R_MSP430_ABS16
```

R_MSP430_PCR16 用于符号寻址模式，但仅限于 MSP430。重定位容器的地址与有效 PC 匹配，因此无需进行加数调整。

```
        MOV.W X, R5       ; R_MSP430_PCR16
        MOV.W R5, X       ; R_MSP430_PCR16
        MOV.W X, Y        ; R_MSP430_PCR16(X) and R_MSP430_PCR16(Y)
        CALL X            ; R_MSP430_PCR16
```

R_MSP430X_ABS16 和 R_MSP430X_PCR16 (均仅限于 MSP430X) 用于为 MSP430X 汇编的 MSP430 指令。对于 MSP430 从中使用 R_MSP430_ABS16 的指令，MSP430X 将使用 R_MSP430X_ABS16。R_MSP430_ABS16 与 R_MSP430X_ABS16 相同，区别是后者检查溢出。R_MSP430_PCR16 与 R_MSP430X_PCR16 类似。

R_MSP430X_ABS16 用于绝对、索引和立即寻址模式，但仅限于 MSP430X：

```
        ADD.W #X, R5      ; R_MSP430X_ABS16
        ADD.W &X, R5      ; R_MSP430X_ABS16
        ADD.W R5, &X      ; R_MSP430X_ABS16
        ADD.W K(R4), R5   ; R_MSP430X_ABS16
```

R_MSP430X_PCR16 用于符号寻址模式，但仅限于 MSP430X：

```
        MOV.W X, R5       ; R_MSP430X_PCR16
        MOV.W R5, X       ; R_MSP430X_PCR16
        MOV.W X, Y        ; R_MSP430X_PCR16(X) and R_MSP430X_PCR16(Y)
        CALL X            ; R_MSP430X_PCR16
```

R_MSP430X_ABS16 和 R_MSP430X_PCR16 也用于大多数 MSP430X 专用“地址指令”（MOVA、ADDA、SUBA 和 CMPA，但不包括 CALLA），因为这些指令在“索引”寻址模式下仅允许 16 位立即数。（还有其他指令允许在索引寻址模式下使用更大的立即数。）

```
        MOVA K(R4), R5    ; R_MSP430X_ABS16 (MSP430X only)
        MOVA X, R5        ; R_MSP430X_PCR16
```

11.5.1.5 MSP430X 指令的重定位

有关在 MSP430X 上汇编的 MSP430 本机指令，请参阅节 11.5.1.4。

MSP430X 的指令编码具有与 MSP430 不同的立即数字段，需要不同的重定位。

“ABS20”和“PCR20”重定位用于“MSP430X 扩展指令”。这些指令需要额外的操作码前缀字来编码 20 位寻址模式 (MOVX、CMPX、ADDX 等)。

“ABS20”重定位也用于“MSP430X 地址指令”。这些指令允许一种 20 位寻址模式，无需额外的操作码前缀字 (ADDA、MOVA、CMPA、SUBA、CALLA)。

R_MSP430X_ABS20_EXT_SRC、R_MSP430X_ABS20_EXT_DST 和 R_MSP430X_ABS20_EXT_ODST 重定位用于绝对、索引和立即寻址模式，但仅限于 MSP430X。它们是不同的，因为它们在指令编码中编码不同的字段。SRC 重定位源操作数；DST 和 ODST 重定位目标操作数。如果源操作数的寻址模式需要额外的字进行编码，包括常量生成器无法处理的 20 位地址和 20 位立即数常量，则使用 ODST 而非 DST。

```
        MOVX &X, R5        ; R_MSP430X_ABS20_EXT_SRC
        MOVX #X, R5        ; R_MSP430X_ABS20_EXT_SRC
        MOVX R5, &Y        ; R_MSP430X_ABS20_EXT_DST
        MOVX #0xabcde, &Y  ; R_MSP430X_ABS20_EXT_ODST (Y)
        MOVX &X, &Y        ; R_MSP430X_ABS20_EXT_SRC (X) and R_MSP430X_ABS20_EXT_ODST (Y)
        MOVX #X, &Y        ; R_MSP430X_ABS20_EXT_SRC (X) and R_MSP430X_ABS20_EXT_ODST (Y)
        ADDX K(R4), R5     ; R_MSP430X_ABS20_EXT_SRC (K)
```

R_MSP430X_PCR20_EXT_SRC、R_MSP430X_PCR20_EXT_DST 和 R_MSP430X_PCR20_EXT_ODST 与 EXT20 重定位类型类似，区别是它们用于符号寻址模式。

```
        MOVX X, R5         ; R_MSP430X_PCR20_EXT_SRC
        MOVX R5, Y         ; R_MSP430X_PCR20_EXT_DST
        MOVX #0xabcde, Y   ; R_MSP430X_PCR20_EXT_ODST (Y)
        MOVX X, Y          ; R_MSP430X_PCR20_EXT_SRC (X) and R_MSP430X_PCR20_EXT_ODST (Y)
```

R_MSP430X_PCR20_CALL 用于具有符号寻址模式的 CALLA：

```
        CALLA  Y           ; R_MSP430X_PCR20_CALL
```

对于 R_MSP430X_PCR20_EXT_SRC、R_MSP430X_PCR20_EXT_DST、R_MSP430X_PCR20_EXT_ODST 和 R_MSP430X_PCR20_CALL，有效 PC 与重定位容器的地址不匹配。为了补偿这一点，汇编器必须根据差异来调整重定位加数。必须通过对 SRC 和 DST 添加 -4、对 ODST 添加 -6、对 CALL 添加 -2 来调整加数。

R_MSP430X_ABS20_ADR_SRC 和 R_MSP430X_ABS20_ADR_DST 用于绝对、索引和立即寻址模式，但仅限于 MSP430X。它们是不同的，因为它们在指令编码中编码不同的字段。SRC 重定位源操作数；DST 重定位目标操作数。CALLA 使用 DST。

```
        MOVA &X, R5       ; R_MSP430X_ABS20_ADR_SRC
        MOVA R5, &X       ; R_MSP430X_ABS20_ADR_DST
        MOVA #X, R5       ; R_MSP430X_ABS20_ADR_SRC
        CALLA #X          ; R_MSP430X_ABS20_ADR_DST
        CALLA &X          ; R_MSP430X_ABS20_ADR_DST
```

R_MSP430_ABS_HI16 用于加载 32 位链接器符号值。链接器符号值大于 20 位指针，因此不能使用单个指令加载整个值。此值必须拆分为两个重定位。此重定位类型表示值的 MSW。它用于与代表 LSW 的 R_MSP430_ABS16 成对使用。此重定位类型是合法的，但在 MSP430 上未使用。

示例：TI 编译器将从以下 C 代码生成此类重定位对：

```
    extern char X;
    unsigned long fn()
    {
        return _symval(&X);
    }
     MOV #$LO16(X), R12; R_MSP430_ABS16
     MOV #$HI16(X), R13; R_MSP430_ABS_HI16
```

11.5.1.6 其他重定位类型

R_MSP430_NONE 重定位类型不执行任何操作。它用于创建一段对另一段的引用，以确保如果引用段链接进来，被引用的段也会链接进来。

R_MSP430_PREL31 用于对异常处理表中的代码地址进行编码。请参阅节 9.2。

R_MSP430_EHTYPE 用于对异常处理表中的 typeinfo 地址进行编码。请参阅节 9.6.1。

11.5.2 重定位操作

表 11-6 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。表示法 [x,y]+[z,w] 指示重定位占用不连续的位范围，应将其连接在一起以形成字段。当加数列中使用“F”时，表示此字段已具有地址空间的准确大小。
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。
r_addend	加数必须存储在 RELA 字段中，并且不能存储在重定位容器中。

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。

表 11-6. MSP430 重定位操作

重定位名称	符号	容器大小 (CS)	字段 [O, FS] (F)	加数 (A)	结果 (R)	溢出检查	编码值 (EV)
R_MSP430_NONE	无	32	[0,32]	无	无	否	无
R_MSP430_ABS32	任一	32	[0,32]	F	S + A	否	R
R_MSP430_ABS16	任一	16	[0,16]	SE(F)	S + A	否	R
R_MSP430_ABS8	任一	8	[0,8]	SE(F)	S + A	是	R
R_MSP430_PCR16	有符号	16	[0,16]	SE(F)	S + A - P	否	R
R_MSP430X_PCR20_EXT_SRC	有符号	48	[7,4]+[32,16]	SE(F)	S + A - P	是	R
R_MSP430X_PCR20_EXT_DST	有符号	48	[0,4]+[32,16]	SE(F)	S + A - P	是	R
R_MSP430X_PCR20_EXT_ODST	有符号	64	[0,4]+[48,16]	SE(F)	S + A - P	是	R
R_MSP430X_ABS20_EXT_SRC	无符号	48	[7,4]+[32,16]	ZE(F)	S + A	是	R
R_MSP430X_ABS20_EXT_DST	无符号	48	[0,4]+[32,16]	ZE(F)	S + A	是	R
R_MSP430X_ABS20_EXT_ODST	无符号	64	[0,4]+[48,16]	ZE(F)	S + A	是	R
R_MSP430X_ABS20_ADR_SRC	无符号	32	[8,4]+[16,16]	ZE(F)	S + A	是	R
R_MSP430X_ABS20_ADR_DST	无符号	32	[0,4]+[16,16]	ZE(F)	S + A	是	R
R_MSP430X_PCR16	有符号	16	[0,16]	SE(F)	S + A - P	是	R
R_MSP430X_PCR20_CALL	有符号	32	[0,4]+[16,16]	SE(F)	S + A - P	是	R
R_MSP430X_ABS16	无符号	16	[0,16]	SE(F)	S + A	是	R
R_MSP430_ABS_HI16	无	16	[0,16]	r_addend	S + A	否	R >> 16
R_MSP430_PREL31	有符号	32	[0,31]	SE(F)	S + A - P	否	R >> 1

*请参阅节 11.5.1.4。

11.5.3 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：

- 对弱函数的引用应使用立即寻址模式来实现。
- 在绝对重定位类型 (R_MSP430_abs*) 中使用时，引用解析为零。

所有其他情况均不符合 ABI。

章节 12

ELF 程序加载和链接（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。

系统可根据其具体要求来使用机制的子集。

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 MSP430 的系统。

12.1 程序标头

程序标头包含以下字段。

p_type

MSP430 没有为程序标头中的 p_type 字段定义特定于处理器的段类型。

p_vaddr，p_paddr

MSP430 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

没有为 MSP430 定义特定于处理器的段标志。

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 MSP430 中，未指定 p_align 的含义和设置。

12.1.1 基址

MSP430 不支持与位置无关的代码，如 System V ABI 标准第 5 章的“基址”一节所述。

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。

12.1.2 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，MSP430 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。

12.1.3 线程局部存储

ABI 当前未为线程局部存储指定标准机制。

12.2 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 MSP430 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 12-1、表 12-2 和表 12-3 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。

表 12-1. 从 ELF 可执行文件创建过程映像的步骤

步骤	
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 12.1。
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.	Marshall 命令行实参和环境变量。此步骤特定于平台。

表 12-2. 初始化执行环境的步骤

步骤	
5.	设置 SP。SP (R1) 应设置为符号 _ _TI_STACK_END 的值，在 8 字节边界上正确对齐。
6.	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 14 说明了在 TI 工具中实现的一种此类机制。
7.	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。指向初始化函数的指针存储在表中。此表由一对全局符号分隔：_ _TI_INITARRAY_Base 和 _ _TI_INITARRAY_Limit。
8.	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 12-3. 终止步骤

步骤	
9.	执行 atexit 调用。以与注册相反的顺序调用由 atexit 注册的函数。

This page intentionally left blank.

章节 13

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。 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。

13.1 MSP430 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 MSP430。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。

MSP430 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。

13.2 MSP430 构建属性标签

OFBA_MSPABI_Tag_ISA (=4), ULEB128

此标签指定可以执行文件中所编码指令的 MSP430 ISA。定义的值如下：

0	ISA 未指定
1	MSP430
2	MSP430X

若要链接，构建中的所有目标文件都必须具有相同的 ISA 标签。

OFBA_MSPABI_Tag_Code_Model, (=6), ULEB128

0	无
1	小型代码模型
2	大型代码模型

此标签指定所使用的代码模型。

若要链接，构建中的所有目标文件均必须具有相同的代码模型。小型代码模型要求使用小型数据模型。MSP430 仅支持小型代码模型。

OFBA_MSPABI_Tag_Data_Model, (=8), ULEB128

0	无
1	小型数据模型
2	大型数据模型
3	受限大型数据模型

此标签指定所使用的数据模型。小型代码模型要求使用小型数据模型。MSP430 仅支持小型数据模型。

若要链接，构建中的所有目标文件均必须具有相同的数据模型。

OFBA_MSPABI_Tag_enum_size, (=10), ULEB128

0	无
1	小（字符型/短整型）
2	整数（默认）
3	不用考虑

此标签指定用于枚举类型的最小容器大小。若要链接，构建中的所有目标文件均必须指定兼容的枚举大小属性。

- **无** - 没有为目标文件指定枚举大小。编译器认为此值与所有值兼容。如果在多个目标文件中引用，用户必须确保各个枚举类型具有一致的大小。
- **小** - 使用可保存枚举类型的最小容器。此值仅与自身兼容。
- **整数** - 使用的最小容器为整数类型。此值仅与自身兼容。这是默认设置。
- **不用考虑** - 目标文件不会跨编译单元引用任何枚举类型。此值与所有值兼容。

表 13-1 总结了由 ABI 定义的构建属性标签。

表 13-1. MSP430 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	
OFBA_MSPABI_Tag_ISA	4	ULEB128	不能在目标文件之间混合使用 ISA、代码模型和数据模型标签。
OFBA_MSPABI_Tag_Code_Model	6	ULEB128	不能在目标文件之间混合使用 ISA、代码模型和数据模型标签。
OFBA_MSPABI_Tag_Data_Model	8	ULEB128	不能在目标文件之间混合使用 ISA、代码模型和数据模型标签。
OFBA_MSPABI_Tag_enum_size	10	ULEB128	请参阅属性说明。

This page intentionally left blank.

章节 14

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。

14.1 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。COPY_RECORD 结构的格式取决于所使用的代码和数据模型。

对于小数据模型和小型代码模型：

```
typedef struct
{
   void  * load_addr; /* 16-bit pointer */
   void  * run_addr;  /* 16-bit pointer */
   uint16  size;
} COPY_RECORD;
```

对于小数据模型和大型代码模型：

```
typedef struct
{
   uint32  load_addr; /* 32-bit storage for data or code pointer */
   uint32  run_addr;  /* 32-bit storage for data or code pointer */
   uint32  size;
} COPY_RECORD;
```

对于大（或受限）数据模型和大型代码模型：

```
typedef struct
{
   void  * load_addr; /* 20-bit pointer */
   void  * run_addr;  /* 20-bit pointer */
   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：

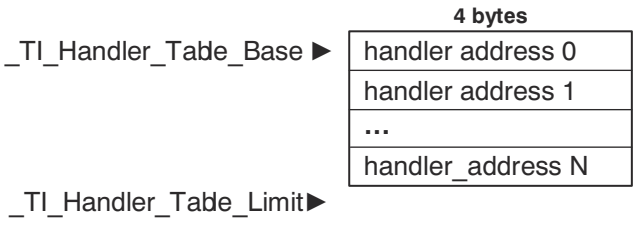


图 14-1. 处理程序表格式

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);
      }
   }
}
```

14.2 压缩的数据格式

理论上，压缩的源数据具有以下格式：

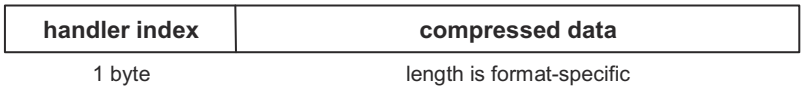


图 14-2. 压缩的源数据格式

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)。

14.2.1 RLE

8 位索引之后的数据使用行程编码 (RLE) 格式进行压缩。MSP430 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。

14.2.2 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。

14.3 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 14 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 14-3 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。

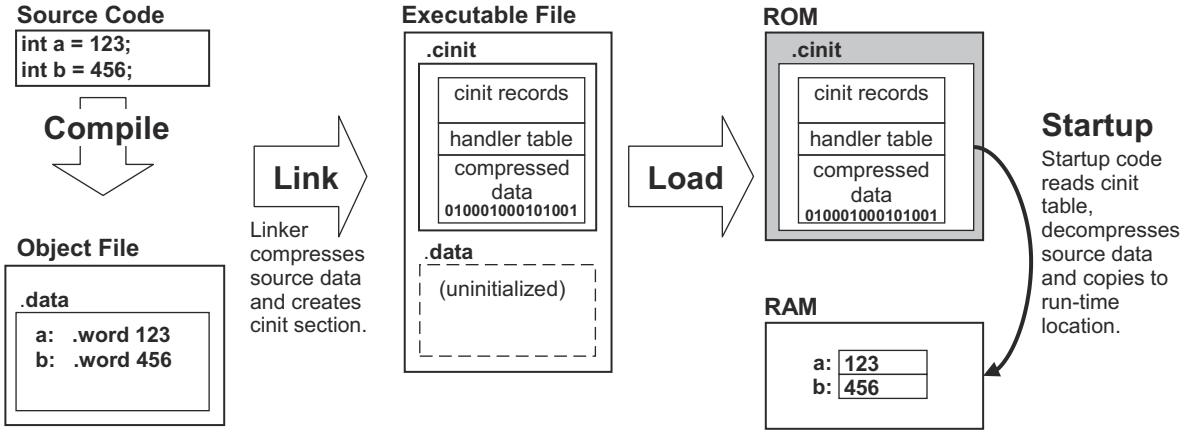


图 14-3. 通过 cinit 进行基于 ROM 的变量初始化

.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 14.1 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 14-4 是展示 .cinit 段的原理图。

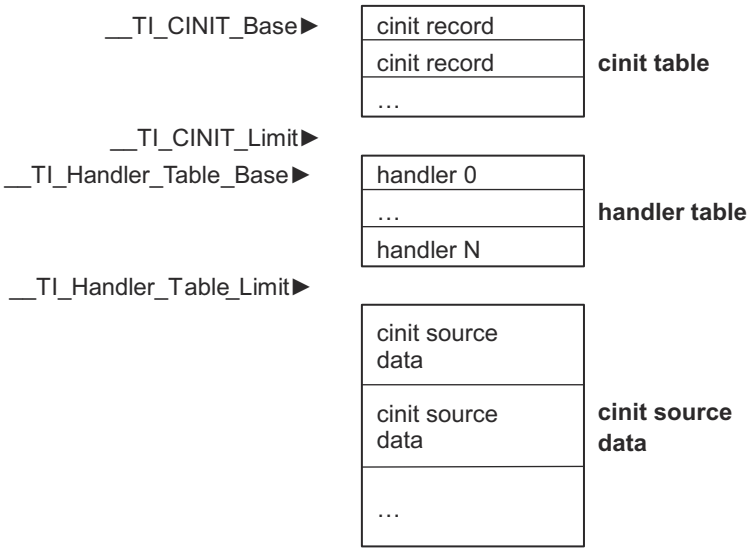


图 14-4. .cinit 段

.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。

CINT_RECORD 结构体的格式取决于所使用的代码和数据模型。

对于小数据模型和小型代码模型：

```
typedef struct  {
   void  * source_data; /* 16-bit pointer */
   void  * dest;        /* 16-bit pointer */
} CINIT_RECORD;
```

对于小数据模型和大型代码模型：

```
typedef struct  {
   uint32  source_data; /* 32-bit storage for data or code pointer */
   uint32  dest;        /* 32-bit storage for data or code pointer */
} CINIT_RECORD;
```

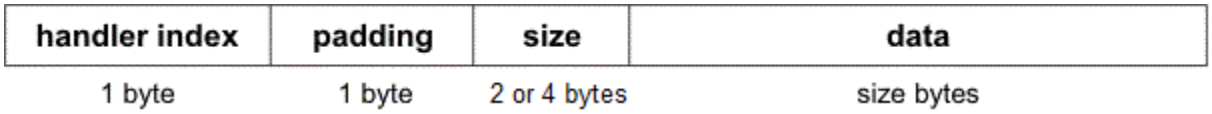
对于大（或受限）数据模型和大型代码模型：

```
typedef struct  {
   void  * source_data; /* 20-bit pointer */
   void  * dest;        /* 20-bit pointer */
} 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。

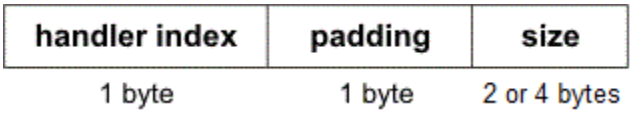
源数据与复制表压缩源数据格式相同（参阅节 14.1），并且处理程序的接口相同。除了 RLE 和 LZSS 格式，cinit 记录还定义了另外两种格式：未压缩和零初始化。

- 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 2 字节边界上对齐。“大小”字段的大小取决于存储器模型。对于小代码和小数据模型，大小为 2 个字节。在所有其他存储器模式组合中，大小为 4 个字节。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 2 字节边界上对齐。“大小”字段的大小取决于存储器模型。对于小代码和小数据模型，大小为 2 个字节。在所有其他存储器模式组合中，大小为 4 个字节。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 15

修订历史记录

表 15-1 列出了自本文档早期版本发布以来所做的更改。

表 15-1. 修订历史记录

	位置	添加/修改/删除
SLAA534A	节 3.3.7、节 3.3.8、节 3.4 和节 3.5，	无论结构体和联合体的大小如何，它们始终通过引用来传递和返回。

重要声明和免责声明

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 产品的应用。严禁对这些资源进行其他复制或展示。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