# TMS320x280x DSP 引导 ROM

# 参考指南

文献编号：ZHCU005
2004 年 11 月 – 修订 2005 年 8 月

TEXAS INSTRUMENTS

# 内容

# 附图目录

# 附表目录

# 请先阅读

## 关于本手册

此参考指南适用于存储在 TMS320x280x 系列处理器的片上引导 ROM 中的代码和数据， 其中包括 280x 系列中所有基于快闪、基于 ROM 和基于 RAM 的器件。

引导 ROM 在出厂时已设定好引导加载软件。 引导模式信号（通用的 I/O）用于指示 bootloader 软件在上电时要使用哪种模式。 TMS320x280x 器件的引导 ROM 还包含用于 *C28x™ IQMath Library - 虚拟浮点引擎*（文献编号 SPRC087）中 IQ 数学相关算法的标准数学表（例如 SIN/COS 波形）。

本指南描述了 bootloader 的用途和特性， 它还描述了器件的片上引导 ROM 的其它内容，并标识了所有信息在该存储器内的位置。

## 命名惯例

本文档使用以下惯例。

- 表示十六进制数时加一个后缀 h 或一个前缀 0x。 例如，以下数字为十六进制的 40（十进制的 64）：40h 或 0x40。
- 本文档中含有寄存器的图形显示和表格说明。
  - 每个寄存器图形显示为一个分成多个字段的矩形，每个字段分别代表了此寄存器的字段。 每个域用其位名标记，域的起始位和结束位标记在标签的上面，域的读取/写入属性标记在下面， 并用图例解释了用于表示属性的符号。
  - 寄存器图形中的保留位指定一位用于将来器件扩展。

## 德州仪器 （TI） 提供的相关文档

以下文档描述了 280x 器件及相关的支持工具。 www.ti.com 网站上提供了这些文档的副本。 *提示：*请在 www.ti.com 上提供的搜索框中输入文献编号。

数据手册 –

**SPRS230：** – TMS320F2801、TMS320F2806、TMS320F2808、UCD9501 数字信号处理器数据手册包含 F280x 器件的引脚、信号说明以及电子和定时规范。

用户指南 –

**SPRU051：** – TMS320x281x、280x 串行通信接口 (SCI) 参考指南描述了一个通常称为 UART 的 SCI，这是一个两线制异步串行端口。 SCI 模块支持 CPU 与其它异步外设之间的使用标准非归零 (NRZ) 格式的数字通信。

**SPRU059：** – TMS320x281x、280x 串行外设接口 (SPI) 参考指南描述了 SPI，一种高速同步串行输入/输出 (I/O) 端口，它允许按照已编程的位传输速率将具有编程长度的串行位流（1 到 16 位）移入或移出器件。

**SPRU074：** – TMS320x281x、280x 增强型控制器局域网络 (eCAN) 参考指南描述了在电噪音环境下使用已设立的协议与其他控制器进行串行通信的 eCAN。

**SPRU430：** – TMS320C28x DSP CPU 和指令集参考指南描述了 TMS320C28x 定点数字信号处理器 (DSP) 的中央处理器 (CPU) 和汇编语言指令， 它还描述了这些 DSP 上可用的仿真功能。

**SPRU513：** – TMS320C28x 汇编语言工具用户指南描述了用于 TMS320C28x 器件的汇编语言工具（汇编程序和用于开发汇编语言代码的其它工具）、汇编程序指令、宏、常用对象文件格式和符号调试指令。

**SPRU514：** – TMS320C28x 优化 C 编译器用户指南描述了 TMS320C28x™ C/C++ 编译器。 此编译器接受 ANSI 标准 C/C++ 源代码，并为 TMS320C28x 器件生成 TMS320 DSP 汇编语言源代码。

**SPRU566：** – TMS320x281x 和 280x 外设参考指南描述了 28x 数字信号处理器 (DSP) 的外设参考指南。

SPRU608： – TMS320C28x 指令集仿真器技术概述描述了 TMS320C2000 IDE 的 Code Composer Studio 内可用于模拟 C28x™ 内核指令集的仿真器。

SPRU625： – TMS320C28x DSP/BIOS 应用编程接口 (API) 参考指南描述了使用 DSP/BIOS 进行的开发。

SPRU712： – TMS320x280x 系统控制和中断参考指南描述了 280x 数字信号处理器 (DSP) 的各种中断和系统控制特性。

SPRU716： – TMS320x280x 模数转换器 (ADC) 参考指南描述了如何配置和使用片上 ADC 模块，这是一种 12 位管线型 ADC。

SPRU721： – TMS320x280x 内部集成电路 (I2C) 参考指南描述了 TMS320x280x 数字信号处理器 (DSP) 上可用的内部集成电路 ($I^2C$) 模块的特性和操作。

SPRU790： – TMS320x280x 增强型正交编码器脉冲 (eQEP) 参考指南描述了 eQEP 模块，在高性能运动和定位控制系统中，该模块用于与线性或旋转增量编码器连接，以从旋转机器中获取位置、方向和速度信息。 该指南同样也包括模块说明和寄存器。

SPRU791： – TMS320x280x 增强型脉宽调制器 (ePWM) 模块参考指南描述了增强型脉宽调制器的主要应用领域，包括数字电机控制、开关模式电源控制、UPS（不间断电源）和其它形式的电力转换。

SPRU807： – TMS320x280x 增强型捕捉 (eCAP) 模块参考指南描述了增强型捕捉模块。 它包括模块说明和寄存器。

SPRU924： – 高分辨率脉宽调制器 (HRPWM) 描述了脉宽调制器的高分辨率扩展版本 (HRPWM) 的操作。

应用报告 –

SPRAA58： – TMS320x281x 到 TMS320x280x 迁移概述描述了德州仪器 (TI) 的 TMS320x281x 与 TMS320x280x DSP 之间的差异，以便在将应用从 281x 迁移到 280x 的过程中提供帮助。 尽管本文档侧重从 281x 到 280x 的迁移，想要反向迁移（从 280x 到 281x）的用户也会发现本文档非常有用。

SPRA550： – 用于数字电机控制的 3.3V DSP 描述了仅使用 3.3V 电机控制器的方案，并指出对于大多数应用，3.3V 与 5V 之间不存在明显的连接问题， 还对比讨论了片上 3.3V 模数转换器 (ADC) 与 5V ADC。概述了可以降低系统噪声和电磁干扰影响的组件布局和印刷电路板 (PCB) 设计指南。

SPRA820： – TMS320C28x DSP 在线堆栈溢出检测介绍了 TMS320C28x™ DSP 上在线堆栈溢出检测的方法， 提供了包含一些函数的 C 源代码，这些函数用于在 DSP/BIOS™ 和非 DSP/BIOS 应用中执行溢出检测。

SPRA861： – RAMDISK：用户定义的 C I/O 驱动程序示例提供了在任意器件上使用高级 CIO 功能的复杂缓冲技术的简易方法。 本应用报告介绍了用户自定义的器件驱动程序的实施示例。

SPRA873： – 使用 TMS320F2812 DSP 和 DRV592 功率放大器的热电制冷器控制介绍了由德州仪器 (TI) 的 TMS320F2812 数字信号处理器 (DSP) 和 DRV592 功率放大器组成的热电制冷器系统。 DSP 使用集成的 12 位模数转换器读取热敏电阻，并将脉宽调制的波形直接输出到 H 桥接的 DRV592 功率放大器，以实现数字比例积分微分反馈控制器。 全面地描述了试验系统以及软件和软件操作指南。

SPRA876： – TMS320F281x eCAN 的编程示例包含几个编程示例，阐述了如何针对不同的操作模式设置 eCAN 模块，以帮助您实现快速 eCAN 编程。 附加的 SPRA876.zip 文件中包含所有项目和 CANalyzer 配置文件。

SPRA953： – IC 封装热度量描述了传统的热度量和新的热度量，并展望其在关于系统级结温估值中的应用。

SPRA958： – 从 TMS320F281x DSP 上的内部闪存运行应用程序（修订版 B）讨论了正确配置从片上闪存执行应用软件所需的要求。 提供了对 DSP/BIOS™ 和非 DSP/BIOS 项目的要求。 包括示例代码项目。

SPRA963： – TMS320LF24x 和 TMS320F281x 器件的可靠性数据描述了 TMS320LF24x 和 TMS320F281x 器件的可靠性数据。

SPRA989： – F2810、F2811 和 F2812 ADC 校准描述了提高 F2810/F2811/F2812 器件上的 12 位模数转换器 (ADC) 绝对精度的方法。 本应用手册附带一个从 F2812 eZdsp 上的 RAM 执行的示例程序 (ADCcalibration.zip)。

**SPRA991：** — 仿真实现了调试和分析的增强 - 白皮书描述了通过允许开发人员更有效地评估系统替代方案来缩短开发周期的仿真增强。

# 引导 ROM 概述

## 1.1 引导 ROM 存储器映射

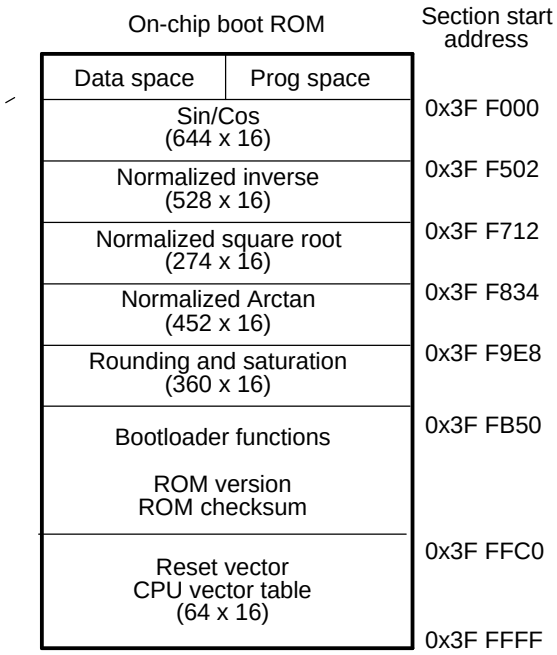
280x 器件的引导 ROM 是一个位于 0x3F F000 - 0x3F FFF 地址处的 4K x 16 只读存储器块。

片上引导 ROM 在出厂时已设定好引导加载例程以及要与 *C28x™ IQMath Library - 虚拟浮点引擎*（文献编号 SPRC087）一起使用的数学表。 第 4 章包含用于以下每一项的代码：

- Bootloader 功能
- 版本号、发布日期和校验和
- 复位向量
- CPU 向量表（仅用于测试）
- IQmath 表

图 1-1 显示了片上引导 ROM 的存储器映射。 该内存块的大小为 4K x 16，位于程序和数据空间的 0x3F F000 - 0x3F FFFF 处。

图 1-1. 片上 ROM 的存储器映射



## 1.2 片上引导 ROM 的 IQMath 表

引导 ROM 存储器保留 3K x 16 个字供 IQMath 表使用。 提供这些数学表是为了帮助提高性能和节省 RAM 空间。

德州仪器™的 *C28x™ IQMath Library - 虚拟浮点引擎*（文献编号 SPRC087）将使用引导 ROM 中包括的这些数学表。 28x IQmath Library 是一个高度优化的高精度数学函数集合，使 C/C++ 编程人员可以将浮点算法无缝地连接到 TMS320C28x 器件上的定点代码中。

这些例程通常用于非常需要最佳执行速度和高精度的计算密集型实时应用中。 使用这些例程所达到的执行速度远远快过使用标准 ANSI C 语言编写的等效代码的执行速度。 另外，TI IQmath Library 通过提供即用型高精度函数，可以显著缩短 DSP 应用的开发时间。 TI 网站上提供了 *C28x™ IQMath Library - 虚拟浮点引擎*（文献编号 SPRC087）的下载。

引导 ROM 中包括下列数学表：

- 正弦/余弦表
  - 表大小： 1282 个字
  - Q 格式： Q30
  - 内容： 1 又 1/4 个周期正弦波 的 32 位示例

  此表对精确正弦波形的生成和 32 位 FFT 很有用； 同时还可用于 16 位数学运算，只需跳过每第二个值即可。
- 归一化反转表
  - 表大小： 528 个字
  - Q 格式： Q29
  - 内容： 32 位归一化反转示例以及 饱和极限

  此表用作牛顿-拉普森 (Newton-Raphson) 反转算法中的初始值估计。 估计越精确，收敛越快，因此周期也更短。
- 归一化平方根表
  - 表大小： 274 个字
  - Q 格式： Q30
  - 内容： 32 位归一化反平方根示例以及 饱和值

  此表用作牛顿-拉普森 (Newton-Raphson) 平方根算法中的初始值估计。 估计越精确，收敛越快，因此周期也更短。
- 归一化反正切表
  - 表大小： 452 个字
  - Q 格式： Q30
  - 内容 最佳拟合的 32 位二阶系数以及 归一化表

  此表用作牛顿-拉普森 (Newton-Raphson) 反正切迭代算法中的初始值估计。 估计越精确，收敛越快，因此周期也更短。
- 舍入和饱和表
  - 表大小： 360 个字
  - Q 格式： Q30
  - 内容： 各 Q 值的 32 位舍入和饱和极限

## 1.3 引导 ROM 的版本及校验和信息

引导 ROM 在地址 0x3F FFBA 处包含自己的版本号。 此版本号的起始值为 1，然后每当修改引导 ROM 代码时，版本号将递增。 下一个地址 0x3F FFBB 处包含此引导代码的发布月份和年份（MM/YY，采用十进制）。 接下来的四个寄存器位置包含引导 ROM 的校验和值。 将 ROM 中的所有地址（校验和位置除外）执行 64 位求和即得到此校验和。

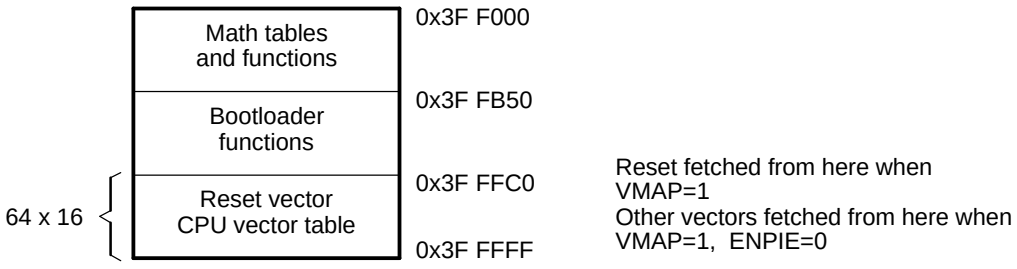
表 1-1. Bootloader 修订及校验和信息

| 地址 | 内容 |
|---|---|
| 0x3F FFB9 | 闪存 API 芯片兼容性检查。 某些版本的闪存 API 将读取此位置，以确保是在兼容的芯片版本上运行。 |
| 0x3F FFBA | 引导 ROM 的版本号 |
| 0x3F FFBB | 发布时间 MM/YY（十进制） |
| 0x3F FFBC | 校验和的最低有效字 |
| 0x3F FFBD | 。 。 。 |
| 0x3F FFBE | 。 。 。 |
| 0x3F FFBF | 校验和的最高有效字 |

## 1.4 CPU 向量表

CPU 向量表位于引导 ROM 寄存器中的 0x3F FFC0 至 0x3F FFFF 地址段。 当 VMAP = 1、ENPIE = 0 时，该向量表在复位后被激活（禁用 PIE 向量表）。

图 1-2. 向量表映射




A VMAP 位在状态寄存器 1 (ST1) 中。 复位时 VMAP 始终为 1。 复位后可通过软件更改该值，但正常操作模式下会保留 VMAP = 1。

B ENPIE 位在 PIECTRL 寄存器中。 复位时此位的默认状态为 0，以禁用外设中断扩展 (PIE) 块。

唯一能从内部引导 ROM 存储器正常处理的向量就是 0x3F FFC0 处的复位向量。 此复位向量在出厂时已设定为指向存储在引导 ROM 中的 InitBoot 函数。 此函数启动引导加载进程。 然后在通用的 I/O (GPIO I/O) 引脚上执行一系列检查操作，以确定将使用哪种引导模式。 本文档的第 2.8 部分中描述了此引导模式选择。

引导 ROM 中的其余向量在正常操作时将不使用。 引导进程完成后，您应当初始化外设中断扩展 (PIE) 向量表并启用 PIE 块。 从此时起，所有向量（复位向量除外）将从 PIE 模块获取，而不是从表 1-2 所示的 CPU 向量表获取。

为执行 TI 芯片调试和测试，引导 ROM 存储器中的向量都指向 M0 SARAM 块中的位置，如表 1-2 所述。 在芯片调试期间，您可以使用分支指令为 M0 中的特定位置编程，以捕获从引导 ROM 获取的任何向量。 正常的器件操作不要求执行此操作。

表 1-2. 向量位置

| 向量 | 在引导 ROM 中的位置 | 内容（即，指向） | 向量 | 在引导 ROM 中的位置 | 内容（即，指向） |
|---|---|---|---|---|---|
| RESET | 0x3F FFC0 | InitBoot (0x3F FB50) | RTOSINT | 0x3F FFE0 | 0x00 0060 |
| INT1 | 0x3F FFC2 | 0x00 0042 | 保留 | 0x3F FFE2 | 0x00 0062 |
| INT2 | 0x3F FFC4 | 0x00 0044 | NMI | 0x3F FFE4 | 0x00 0064 |

表 1-2. 向量位置(接上表)

| 向量 | 在引导 ROM 中的位置 | 内容（即，指向） | 向量 | 在引导 ROM 中的位置 | 内容（即，指向） |
|---|---|---|---|---|---|
| INT3 | 0x3F FFC6 | 0x00 0046 | ILLEGAL | 0x3F FFE6 | 0x00 0066 |
| INT4 | 0x3F FFC8 | 0x00 0048 | USER1 | 0x3F FFE8 | 0x00 0068 |
| INT5 | 0x3F FFCA | 0x00 004A | USER2 | 0x3F FFEA | 0x00 006A |
| INT6 | 0x3F FFCC | 0x00 004C | USER3 | 0x3F FFEC | 0x00 006C |
| INT7 | 0x3F FFCE | 0x00 004E | USER4 | 0x3F FFEE | 0x00 006E |
| INT8 | 0x3F FFD0 | 0x00 0050 | USER5 | 0x3F FFF0 | 0x00 0070 |
| INT9 | 0x3F FFD2 | 0x00 0052 | USER6 | 0x3F FFF2 | 0x00 0072 |
| INT10 | 0x3F FFD4 | 0x00 0054 | USER7 | 0x3F FFF4 | 0x00 0074 |
| INT11 | 0x3F FFD6 | 0x00 0056 | USER8 | 0x3F FFF6 | 0x00 0076 |
| INT12 | 0x3F FFD8 | 0x00 0058 | USER9 | 0x3F FFF8 | 0x00 0078 |
| INT13 | 0x3F FFDA | 0x00 005A | USER10 | 0x3F FFFA | 0x00 007A |
| INT14 | 0x3F FFDC | 0x00 005C | USER11 | 0x3F FFFC | 0x00 007C |
| DLOGINT | 0x3F FFDE | 0x00 005E | USER12 | 0x3F FFFE | 0x00 007E |

# Bootloader 特性

本部分详细描述了引导模式选择进程以及 bootloader 操作的具体细节。

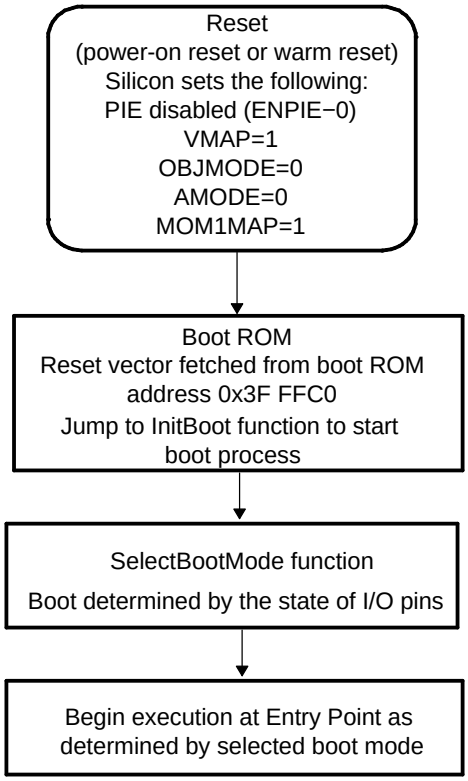
## 2.1 Bootloader 函数操作

Bootloader 是位于片上引导 ROM 中的在复位后执行的程序。

Bootloader 用于在上电后将代码从外部源传输到内存储器中； 从而允许将代码驻留在外部的慢速非易失存储器中，然后传输至高速存储器中执行。

Bootloader 提供了多种不同的下载代码方式以适应不同的系统要求。 Bootloader 使用各种 GPIO 信号确定将使用哪种引导模式。 本文档中的其余部分描述了引导模式选择进程以及每个 bootloader 的具体细节。 图 2-1 显示了基本 bootloader 流程。

图 2-1. Bootloader 流程图




引导 ROM 中的复位向量将程序执行重定向至 InitBoot 函数。 执行器件初始化之后，bootloader 将检查 GPIO 引脚的状态以确定您需要执行哪种引导模式。 这些选项包括：跳转至闪存、跳转至 SARAM、跳转至 OTP 或调用其中一个片上引导加载例程。

完成选择进程后，如果已完成所需的引导加载，处理器将在所选引导模式确定的应用起点继续执行。 如果调用了 bootloader，则由外设加载的输入流确定此应用起点地址。 第 2.9 部分中描述了此数据流。 然而，如果您选择直接引导至闪存、OTP 或 SARAM，这些存储器块中每一个存储器块的应用起点地址均已预定义。

以下部分详细论述了不同的可用引导模式以及用于将数据代码加载至器件的进程。

## 2.2 Bootloader 器件配置

复位时，任何基于 28x™ CPU 的器件都处于 27x™ 对象兼容模式。 在继续执行前，如何将器件置于正确的操作模式下取决于应用程序。

对于 28x 器件，当从内部引导 ROM 执行引导时，此引导 ROM 软件将针对 28x 操作模式配置器件。 您负责所有必需的附加配置。

例如，如果您的应用包括 C2xLP™ 源，在执行从 C2xLP 源生成的代码之前，您首先需要针对 C2xLP 源兼容性配置器件。

表 2-1 中概述了每个操作模式所需的配置。

表 2-1. 器件模式的配置

| | C27x 模式（复位） | 28x 模式 | C2xLP 源<br>兼容模式 |
|---|---|---|---|
| OBJMODE | 0 | 1 | 1 |
| AMODE | 0 | 0 | 1 |
| PAGE0 | 0 | 0 | 0 |
| M0M1MAP(1) | 1 | 1 | 1 |
| 其它设置 | | | SXM = 1，C = 1，SPM = 0 |

(1) 对于 C27x 兼容性，M0M1MAP 通常为 0。 然而对于 280x，M0M1MAP 由于在内部切断连接而处于高电平，因此在这些器件上，始终要在复位时针对 28x 模式配置 M0M1MAP。

## 2.3 PLL 硬件乘法器选择

引导 ROM 不更改 PLL 的状态。 请注意，PLL 硬件乘法器不受调试器的复位影响。 因此，对于在 Code Composer Studio™ 复位时初始化的引导，其速度可能不同于将外部复位线 ($\overline{XRS}$) 拉低时执行的引导速度。

## 2.4 看门狗模块

当直接分支到闪存、M0 单存取 RAM (SARAM) 或一次性可编程 (OTP) 存储器时，将不触发看门狗。 如果采用其它引导模式，则在引导之前禁用看门狗，然后重新启用，并在分支到最终目的地址之前将看门狗清零。

## 2.5 内部上拉电阻

每个 GPIO 引脚都有一个可在软件中启用或禁用的内部上拉电阻。 对于引导模式选择代码在确定选择哪种引导模式时将读取的引脚，默认情况下将在复位后为这些引脚启用上拉。 在噪音条件下，仍然建议您在外部配置这三种引导模式选择引脚中的每一个引脚。

单个 bootloader SCI、SPI、eCAN 和并行引导都为用于执行控制和数据传输的引脚启用上拉电阻。 Bootloader 会在退出时保持为这些引脚启用上拉电阻。 例如，SCI-A bootloader 在 SCITXA 和 SCIRXA 引脚上启用上拉电阻。 Bootloader 退出后，如果需要禁用上拉电阻，则由您执行禁用。

## 2.6 PIE 配置

引导模式不启用 PIE。 它将保持默认状态（禁用）。

## 2.7 保留存储器

M1 存储器块的前 80 个字（地址 0x400 - 0x44F）保留供引导加载进程中的堆栈和 .ebss 代码部分使用。 如果代码被引导加载至此区域，则不执行错误检查，执行此检查是为了防止代码破坏引导 ROM 堆栈。

## 2.8 Bootloader 模式

要适应不同的系统要求，引导 ROM 提供了多种不同的引导模式。 此部分描述了不同的引导模式并概要介绍了各自的函数操作。 三个 GPIO 引脚的状态用于确定所需的引导模式，如表 2-2 所示。

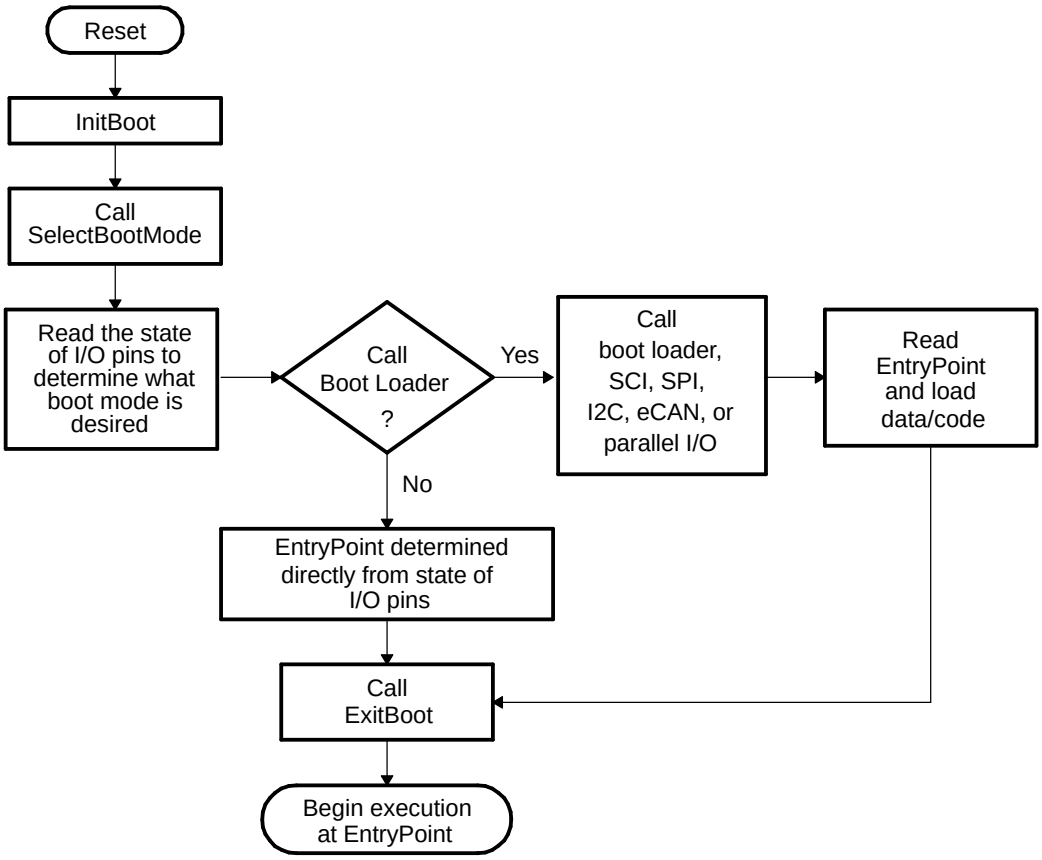
表 2-2. 引导模式选择

| 模式 | 说明 | GPIO18 SPICLKA(1) SCITXB | GPIO29 S1CITXA | GPIO34 |
|---|---|---|---|---|
| 引导至闪存(2) | 跳转至闪存地址 0x3F 7FF6。 在复位以根据需要重定向代码执行之前，必须在此处编写分支指令。 | 1 | 1 | 1 |
| SCI-A 引导 | 从 SCI-A 加载数据流。 | 1 | 1 | 0 |
| SPI-A 引导 | 从 SPI-A 上的外部串行 SPI EEPROM 加载。 | 1 | 0 | 1 |
| $I^2C$ 引导 | 在 $I^2C$ 总线上的 0x50 地址处从外部 EEPROM 加载数据。 | 1 | 0 | 0 |
| eCAN-A 引导 | 调用 CAN_Boot 以从 eCAN-A mailbox 1 加载。 | 0 | 1 | 1 |
| 引导至 M0 SARAM(3) | 跳转至 M0 SARAM 0x00 0000 地址处。 | 0 | 1 | 0 |
| 引导至 OTP(3) | 跳转至 OTP 0x3D 7800 地址处。 | 0 | 0 | 1 |
| 并行 I/O 引导 | 从 GPIO0 - GPIO15 加载数据。 | 0 | 0 | 0 |

(1) 必须格外谨慎，因为切换 SPICLKA 以选择引导模式可能会对外部逻辑产生影响。
(2) 如果采用直接引导至闪存，则假定您之前已经在 0x3F 7FF6 处编写了分支语句，以便根据需要重定向程序流。
(3) 如果采用直接引导至 OTP 或 M0 SARAM，则假定您之前已经编写或加载了从应用起点位置开始的代码。

图 2-2 显示了引导进程的概述。 后续部分将更详细地描述每个步骤。

图 2-2. 引导 ROM 功能概述



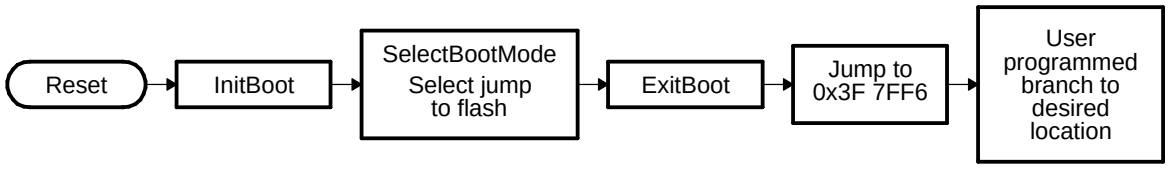

以下引导模式不调用 bootloader。 它们会跳转至存储器中的预定义位置：

- 跳转至闪存中的分支指令

  在此模式下，引导 ROM 软件将针对 28x 操作配置器件，然后直接分支至闪存的 0x3F 7FF6 位置。 此位置刚好在 128 位代码安全模块 (CSM) 密码位置之前。 您需要预先在 0x3F 7FF6 位置处编写分支指令，以将代码执行重定向至定制的 boot-loader 或应用代码。

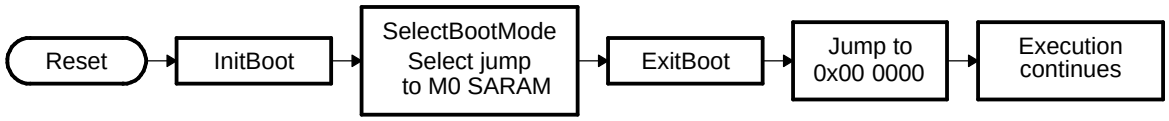
  在仅具有 RAM 的器件上，“引导至闪存”选项将跳转至保留存储器，因此不应当使用此选项。 在仅具有 ROM 的器件上，“引导至闪存”选项将跳转至 ROM 中的 0x3F7FF6 位置。

图 2-3. 跳转至闪存流程图



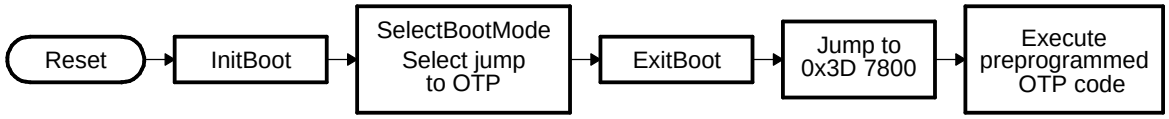
- 跳转至 M0 SARAM
  在此模式下，引导 ROM 软件将针对 28x 操作配置器件，然后直接分支至 0x00 0000，即 M0 SARAM 存储器块中的第一个地址。

图 2-4. 跳转至 M0 SARAM 的流程图




- 跳转至 OTP 存储器
  在此模式下，引导 ROM 软件将针对 28x 操作配置器件，然后直接分支至 0x3D 7800，即 OTP 存储器块中的第一个地址。
  在 ROM 器件上，“引导至 OTP”选项将跳转至 ROM 中的 0x3D 7800 地址。 在 RAM 器件上，“引导至 OTP”选项将跳转至保留存储器，因此不应当使用此选项。

图 2-5. 跳转至 OTP 存储器的流程图




以下引导模式将调用一个用于将数据流从外设加载至存储器的引导加载例程：

- 标准串行引导模式 (SCI-A)
  在此模式下，引导 ROM 通过 SCI-A 端口将要执行的代码加载至片上存储器。
- SPI EEPROM 引导模式 (SPI-A)
  在此模式下，引导 ROM 通过 SPI-A 端口将代码和数据从外部 EEPROM 加载至片上存储器。
- $I^2C$-A 引导模式 ($I^2C$-A)
  在此模式下，引导 ROM 在 $I^2C$-A 总线上的 0x50 地址处将代码和数据从外部 EEPROM 加载至片上存储器。EEPROM 必须遵守采用 16 位基址结构的传统的 $I^2C$ EEPROM 协议。
- eCAN 引导模式 (eCAN-A)
  在此模式下，eCAN-A 外设用于通过 eCAN-A mailbox 1 将数据和代码传输至片上存储器。 传输的是 8 位数据流，其中两个 8 位值在每次通信时都会传输。
- 从 GPIO 端口引导（从 GPIO0-GPIO15 并行引导）
  在此模式下，引导 ROM 使用 GPIO 端口的 A 引脚 GPIO0-GPIO15 从外部源加载代码和数据。 此模式支持 8 位和 16 位数据流。 由于此模式需要用到一些 GPIO 引脚，因此当器件连接至明确用于闪存编程（而不是目标电路板）的平台时，通常采用此模式为闪存编程下载代码。

## 2.9 Bootloader 数据流结构

以下两个表及相关示例显示了流入 bootloader 的数据流的结构。 对于所有 bootloader，此基本结构都相同，并且基于 C54x 十六进制实用程序生成的 C54x 源数据流。 C28x 十六进制实用程序 (hex2000.exe) 已经更新，以支持此结构。 C2000 代码生成工具中附带了 hex2000.exe 实用程序。 数据流结构中的所有值都是十六进制的。

数据流中的第一个 16 位字称为键值。 该键值用于向 bootloader 指示流入数据流的宽度：8 或 16 位。 请注意，并非所有 bootloader 都可以同时接受 8 位和 16 位数据流。 关于有效数据流宽度，请参阅关于各个加载程序的详细信息。 对于 8 位数据流，键值为 0x08AA，对于 16 位数据流，键值为 0x10AA。 如果 bootloader 收到一个无效键值，加载则中止。 在此例中，将使用闪存的应用起点 (0x3F 7FF6)。

接下来的 8 个字用于初始化寄存器值，或者为 bootloader 传递值以增强 bootloader。 如果 bootloader 不使用这些值，则将这些值留作将来使用，并且 bootloader 将只读取这些值，然后丢弃。 当前只有 SPI 和 $I^2C$ bootloader 使用这些字来初始化寄存器。

第十个和第十一个字组成了 22 位应用起点地址。 此地址用于在完成引导加载后初始化 PC。 此地址很可能是 bootloader 所下载的程序的应用起点。

数据流中的第十二个字表示要传输的第一个数据块的大小。 对于 8 位和 16 位数据流格式，该数据块的大小均定义为块中的 16 位字数。 例如，要从 8 位数据流中传输一个包含 20 个 8 位数据值的数据块，该块大小将为 0x000A，表示有 10 个 16 位字。

接下来的两个字向加载程序指示数据块的目的地址。 大小和地址后面的是构成该数据块的 16 位字。

此数据块大小/目的地址模式将重复用于要传输的每个数据块。 一旦传输完所有数据块，就会向加载程序发送一个 0x0000 块大小的信号，告知传输已完成。 此时加载程序将应用起点地址返回至调用例程，后者将清除并退出， 然后在输入数据流内容确定的应用起点地址处继续执行。

表 2-3. 16 位模式的源程序数据流的总体结构

| 字 | 内容 |
|---|---|
| 1 | 10AA（存储器宽度为 16 位的键值） |
| 2 | 寄存器初始化值，或留作将来使用 |
| 3 | 寄存器初始化值，或留作将来使用 |
| 4 | 寄存器初始化值，或留作将来使用 |
| 5 | 寄存器初始化值，或留作将来使用 |
| 6 | 寄存器初始化值，或留作将来使用 |
| 7 | 寄存器初始化值，或留作将来使用 |
| 8 | 寄存器初始化值，或留作将来使用 |
| 9 | 寄存器初始化值，或留作将来使用 |
| 10 | 应用起点 PC[22:16] |
| 11 | 应用起点 PC[15:0] |
| 12 | 要加载的数据的第一个块大小（字数） 如果块大小为 0，则表明源程序结束， 否则后面将跟随另一部分。 |
| 13 | 第一个块的目的地址 Addr[31:16] |
| 14 | 第一个块的目的地址 Addr[15:0] |
| 15 | 源中要加载的第一个块的第一个字 |
| ... | ... |
| ... | ... |
| 。 | 源中要加载的第一个块的最后一个字 |
| 。 | 要加载的第二个块的块大小。 |
| 。 | 第二个块的目的地址 Addr[31:16] |
| 。 | 第二个块的目的地址 Addr[15:0] |
| 。 | 源中要加载的第二个块的第一个字 |
| 。 | … |
| 。 | 源中要加载的第二个块的最后一个字 |
| 。 | 要加载的最后一个块的块大小 |
| 。 | 最后一个块的目的地址 Addr[31:16] |
| 。 | 最后一个块的目的地址 Addr[15:0] |
| 。 | 源中要加载的最后一个块的第一个字 |
| ... | ... |
| ... | ... |
| n | 源中要加载的最后一个块的最后一个字 |
| n+1 | 数据块大小为 0000h - 表示源程序结束 |

*示例 2-1. 数据流结构 16 位*

```
10AA    ; 0x10AA 16-bit key value 0000    ; 8 reserved words
0000
0000
0000
0000
0000
0000
0000
003F    ; 0x003F8000 EntryAddr, starting point after boot load completes
8000
0005    ; 0x0005 - First block consists of 5 16-bit words
003F    ; 0x003F9010 - First block will be loaded starting at 0x3F9010
9010
0001    ; Data loaded = 0x0001 0x0002 0x0003 0x0004 0x0005
0002
0003
0004
0005
0002    ; 0x0002 - 2nd block consists of 2 16-bit words
003F    ; 0x003F8000 - 2nd block will be loaded starting at 0x3F8000
8000
7700    ; Data loaded = 0x7700 0x7625
7625
0000    ; 0x0000 - Size of 0 indicates end of data stream
```

加载完成后，则将已经按照如下所示初始化随后的存储器值：

```
Location     Value
0x3F9010     0x0001
0x3F9011     0x0002
0x3F9012     0x0003
0x3F9013     0x0004
0x3F9014     0x0005
0x3F8000     0x7700
0x3F8001     0x7625
PC Begins execution at 0x3F8000
```

在 8 位模式下，首先发送的是字的最低有效字节 (LSB)，然后是最高有效字节 (MSB)。 对于 32 位值，例如目的地址，首先加载的是最高有效字 (MSW)，然后是最低有效字 (LSW)。 Bootloader 在加载 8 位数据流时会考虑这一点。

表 2-4. 8 位数据流中的 LSB/MSB 加载顺序

| 字节 | | 内容 | |
|---|---|---|---|
| | | 可能是 LSB(每两个字节中的第一个字节) | MSB(每两个字节中的第二个字节) |
| 1 | 2 | LSB：AA（存储器宽度为 8 位的键值） | MSB：08h（存储器的键值宽度 = 8 位） |
| 3 | 4 | LSB：寄存器初始化值，或保留 | MSB：寄存器初始化值，或保留 |
| 5 | 6 | LSB：寄存器初始化值，或保留 | MSB：寄存器初始化值，或保留 |
| 7 | 8 | LSB：寄存器初始化值，或保留 | MSB：寄存器初始化值，或保留 |
| ... | ... | ... | ... |
| ... | ... | ... | ... |
| 17 | 18 | LSB：寄存器初始化值，或保留 | MSB：寄存器初始化值，或保留 |
| 19 | 20 | LSB：应用起点的上半部 PC[23:16] | MSB：应用起点的上半部 PC[31:24] （始终为 0x00） |
| 21 | 22 | LSB：应用起点的下半部 PC[7:0] | MSB：应用起点的下半部 PC[15:8] |
| 23 | 24 | LSB：要加载的第一个数据块的大小（以字数计） 如果块大小为 0，则表明源程序结束， 否则后面将跟随另一个块。例如，块大小为 0x000A，表示块中有 10 个字或 20 个字节。 | MSB：块大小 |
| 25 | 26 | LSB：MSW 目的地址，第一个块 Addr[23:16] | MSB：MSW 目的地址，第一个块 Addr[31:24] |
| 27 号 | 28 | LSB：LSW 目的地址，第一个块 Addr[7:0] | MSB：LSW 目的地址，第一个块 Addr[15:8] |
| 29 | 30 | LSB：要加载的第一个块的第一个字 | MSB：要加载的第一个块的第一个字 |
| ... | ... | ... | ... |
| ... | ... | ... | ... |
| 。 | 。 | LSB：要加载的第一个块的最后一个字 | MSB：要加载的第一个块的最后一个字 |
| 。 | 。 | LSB：第二个块的大小 | MSB：第二个块的大小 |
| 。 | 。 | LSB：MSW 目的地址，第二个块 Addr[23:16] | MSB：MSW 目的地址，第二个块 Addr[31:24] |
| 。 | 。 | LSB：LSW 目的地址，第二个块 Addr[7:0] | MSB：LSW 目的地址，第二个块 Addr[15:8] |
| 。 | 。 | LSB：要加载的第二个块的第一个字 | MSB：要加载的第二个块的第一个字 |
| ... | ... | ... | ... |
| ... | ... | ... | ... |
| 。 | 。 | LSB：第二个块的最后一个字 | MSB：第二个块的最后一个字 |
| 。 | 。 | LSB：最后一个块的大小 | MSB：最后一个块的大小 |
| 。 | 。 | LSB: MSW 目的地址，最后一个块 Addr[23:16] | MSB：MSW 目的地址，最后一个块 Addr[31:24] |
| 。 | 。 | LSB：LSW 目的地址，最后一个块 Addr[7:0] | MSB：LSW 目的地址，最后一个块 Addr[15:8] |
| 。 | 。 | LSB：要加载的最后一个块的第一个字 | MSB：要加载的最后一个块的第一个字 |
| ... | ... | ... | ... |
| ... | ... | ... | ... |
| 。 | 。 | LSB：最后一个块的最后一个字 | MSB：最后一个块的最后一个字 |
| n | n+1 | LSB: 00h | MSB：00h - 表示源结束 |

*示例 2-2. 数据流结构 8 位*

```
AA 08       ; 0x08AA 8-bit key value
00 00 00 00 ; 8 reserved words
00 00 00 00
00 00 00 00
00 00 00 00
3F 00 00 80 ; 0x003F8000 EntryAddr, starting point after boot load completes
05 00       ; 0x0005 - First block consists of 5 16-bit words
3F 00 10 90 ; 0x003F9010 - First block will be loaded starting at 0x3F9010
01 00       ; Data loaded = 0x0001 0x0002 0x0003 0x0004 0x0005
02 00
03 00
04 00
05 00
02 00       ; 0x0002 - 2nd block consists of 2 16-bit words
3F 00 00 80 ; 0x003F8000 - 2nd block will be loaded starting at 0x3F8000
00 77       ; Data loaded = 0x7700 0x7625
25 76
00 00       ; 0x0000 - Size of 0 indicates end of data stream
```

加载完成后，则将已经按照如下所示初始化随后的存储器值：

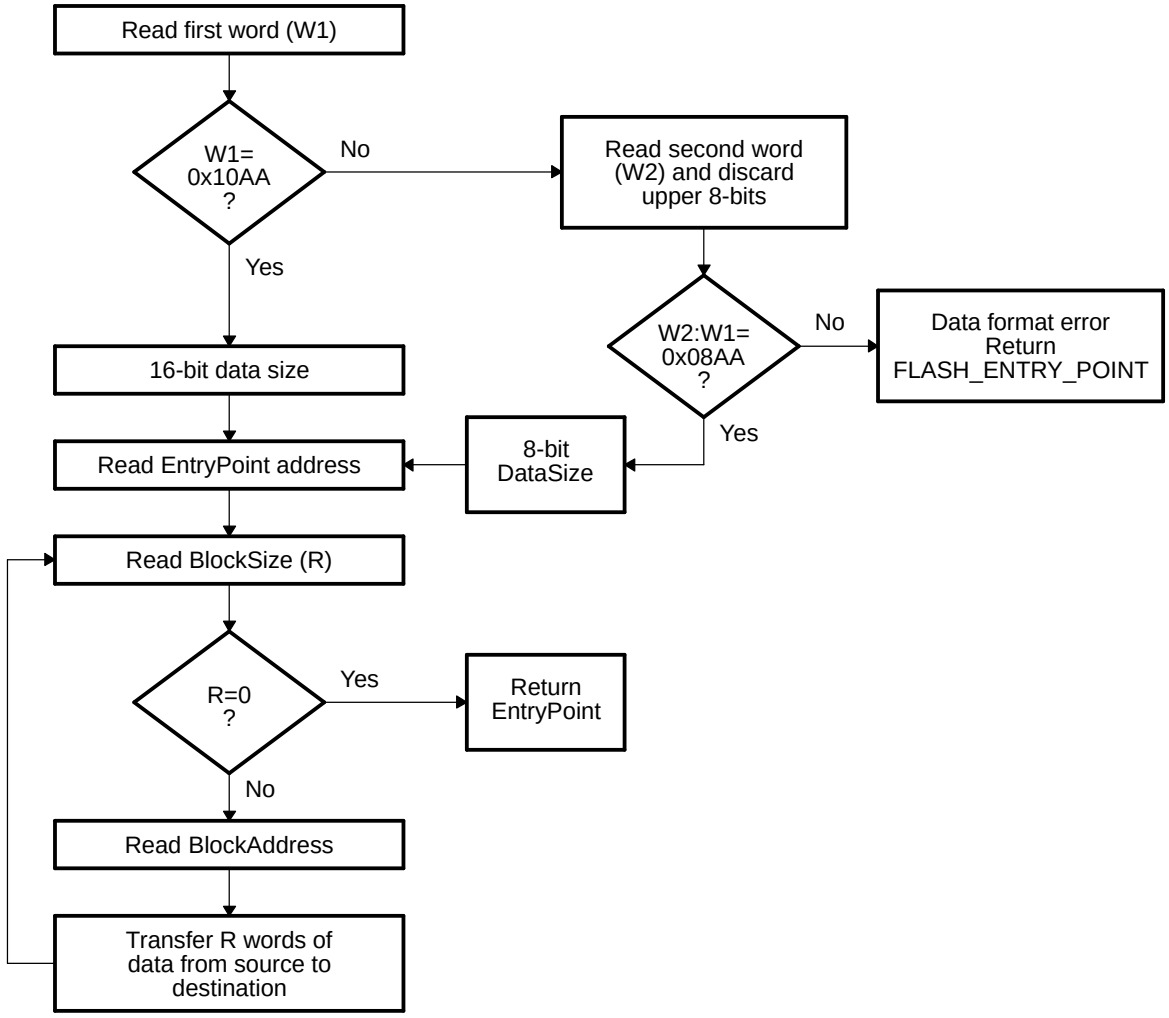
```
Location    Value
0x3F9010    0x0001
0x3F9011    0x0002
0x3F9012    0x0003
0x3F9013    0x0004
0x3F9014    0x0005
0x3F8000    0x7700
0x3F8001    0x7625
PC Begins execution at 0x3F8000
```

## 2.10 基本传输步骤

图 2-6 阐述了 bootloader 用于确定是已选择 8 位还是 16 位数据流、传输该数据以及开始执行程序的基本进程。 当 bootloader 找到根据 GPIO 引脚的状态选定的有效引导模式之后就会开始执行此进程。

加载程序首先将主机发送的第一个值与 16 位键值 0x10AA 进行比较。 如果获取的此值不匹配，加载程序则读取第二个值， 此值将与第一个值组合成一个字， 然后根据 8 位键值 0x08AA 检查该字。 如果加载程序发现此报头与 8 位或 16 位键值均不匹配，或者如果该值对给定的引导模式无效，加载则中止。 在此例中，加载程序将闪存的应用起点地址返回至调用例程。

图 2-6. Bootloader 基本传输步骤




A 8 位和 16 位传输对所有引导模式均无效。 有关任何限制，请参阅特定于具体 bootloader 的信息。

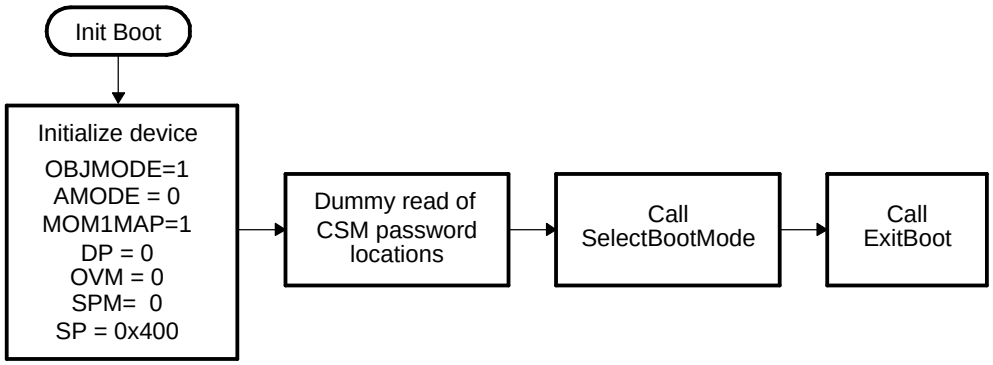
B 在 8 位模式下，首先读取的是 16 位字的 LSB，然后是 MSB。

## 2.11 InitBoot 汇编例程

复位后调用的第一个例程就是 InitBoot 汇编例程。 此例程将针对 C28x 对象模式下的操作初始化器件。 InitBoot 还会虚拟读取代码安全模块 (CSM) 密码位置。 如果 CSM 密码被擦除（均为 0xFFFF），则起到解锁 CSM 的作用。 否则 CSM 仍保持锁定，并且密码位置的虚拟读取将不起作用。 如果您需要引导加载一台新器件，这会很有用。

虚拟读取 CSM 密码位置之后，InitBoot 例程将调用 SelectBootMode 函数。 此函数用于确定某些 GPIO 引脚的状态所需的引导模式类型。 此进程在第 2.12 部分中有介绍。 一旦完成引导，SelectBootMode 函数就会将应用起点地址 (EntryAddr) 传递回 InitBoot 函数。 Bootloader 退出后,代码的执行从 EntryAddr 位置开始。 InitBoot 接着调用 ExitBoot 例程，后者然后将 CPU 寄存器恢复为复位状态，并退出到引导模式确定的 EntryAddr。

图 2-7. InitBoot 汇编函数概述




## 2.12 SelectBootMode 函数

为确定所需的引导模式，SelectBootMode 函数会检查 3 个 GPIO 引脚的状态，如表 2-5 所示。

表 2-5. 引导模式选择

| 模式 | 说明 | GPIO18 SPICLKA (1) SCITXB | GPIO29 SCITXA | GPIO34 |
|---|---|---|---|---|
| 引导至闪存(2) | 跳转至闪存地址 0x3F 7FF6。 在复位以根据需要重定向代码执行之前，必须在此处编写分支指令。 | 1 | 1 | 1 |
| SCI-A 引导 | 从 SCI-A 加载。 | 1 | 1 | 0 |
| SPI-A 引导 | 从 SPI-A 上的外部串行 SPI EEPROM 加载。 | 1 | 0 | 1 |
| $I^2C$-A 引导 | 在 $I^2C$-A 总线上的 0x50 地址处从外部 EEPROM 加载。 | 1 | 0 | 0 |
| eCAN-A 引导 | 调用 CAN_Boot 以从 eCAN-A mailbox 1 加载。 | 0 | 1 | 1 |
| 引导至 M0 SARAM(3) | 跳转至 M0 SARAM 0x00 0000 地址处。 | 0 | 1 | 0 |
| 引导至 OTP(3) | 跳转至 OTP 0x3D 7800 地址处。 | 0 | 0 | 1 |
| 并行 I/O 引导 | 从 GPIO0 - GPIO15 加载。 | 0 | 0 | 0 |

(1) 必须格外谨慎，因为切换 SPICLKA 以选择引导模式可能会对外部逻辑产生影响。
(2) 如果采用直接引导至闪存，则假定您之前已经在 0x3F 7FF6 处编写了分支语句，以便根据需要重定向程序流。
(3) 如果采用直接引导至 OTP 或 M0，则假定您之前已经编写或加载了从应用起点位置开始的代码。

对于要选定的引导模式，必须拉低或拉高与所需引导模式相对应的引脚，直至完成选择进程。 请注意，复位时选择引脚的状态未被锁定； 在SelectBootMode 函数中,几个周期后这些引脚采样会被采样。 复位时将为引导模式选择引脚启用内部上拉电阻。 仍然建议在外部执行引导模式配置，以避免任何噪音对这些引脚产生影响。

SelectBootMode 函数将检查 PLLSTS 寄存器中是否缺少时钟检测位 (MCLKSTS)，以确定 PLL 是否在跛行模式下操作。 如果 PLL 在跛行模式下操作，引导模式选择功能则根据所选的引导模式采取相应的操作：

- 引导至闪存、OTP、SARAM、$I^2C$-A、SPI-A 以及并行 I/O
  这些模式的表现正常。 如果设置了 MCLKSTS 位，用户的软件必须检查是否缺少时钟状态并采取相应的操作。
- SCI-A 引导
  将调用 SCI bootloader。 然而，根据所请求的波特率，此器件可能无法进行自动波特锁定。 在此例中，引导 ROM 软件将在自动波特锁定函数中无限循环。 如果 SCI-A 引导完成，用户的软件必须检查是否缺少时钟状态，并采取相应的操作。
- eCAN-A 引导
  将不调用 eCAN bootloader。 引导 ROM 将无限循环。

注:

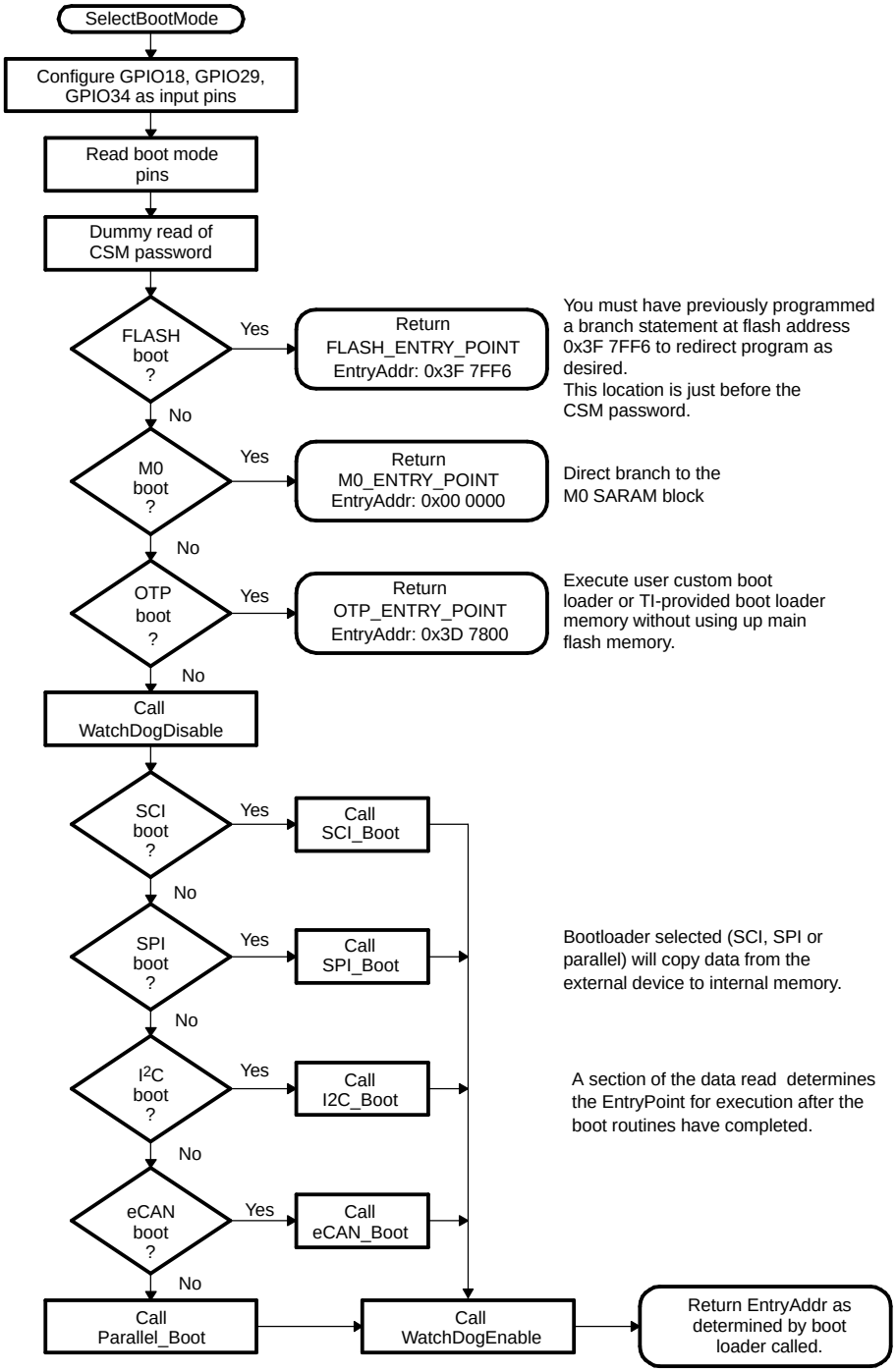
SelectBootMode 例程在调用 SCI、I²C、eCAN、SPI 或并行 bootloader 之前会禁用看门狗。这些 bootloader 不为看门狗提供服务，而且假定看门狗已被禁用。 SelectBootMode 例程在退出之前将重新启用看门狗，并为其复位计时器。

如果不调用 bootloader，则保持不触发看门狗。

选择引导模式时，应当通过轻微下拉或上拉来拉低或拉高引脚，以便 DSP 可以在需要时驱动这些引脚进入新状态。 例如，您希望从 SCI-A 引导，则要拉高的其中一个引脚是 SCITXDA 引脚。 上拉必须是轻微的，以便在 SCI 引导进程开始时，DSP 能够通过 TX 引脚正确发送。 这同样适用于其余的引导模式选择引脚。

在使用 SPICLKA 信号选择引导模式时，务必格外谨慎。 此信号的切换可能会对外部逻辑产生影响，而且产生的影响不容忽略。

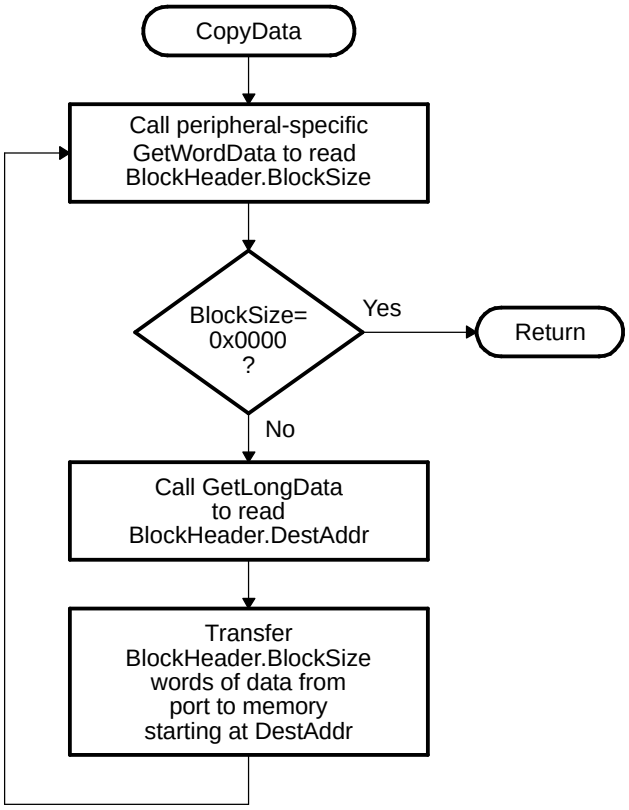
图 2-8. SelectBootMode 函数概述



## 2.13 CopyData 函数

每个 bootloader 都使用相同的函数将数据从端口复制至 DSP SARAM。 此函数就是 CopyData() 函数。 它使用一个指针指向 GetWordData 函数，后者由每个加载程序初始化以正确从该端口读取数据。 例如，在调用 SPI 加载程序时，GetWordData 函数指针被初始化，以指向特定于 SPI 的 SPI_GetWordData 函数， 以便在调用 CopyData() 函数时可访问正确的端口。 CopyData 函数的流程如图 2-9 所示。。

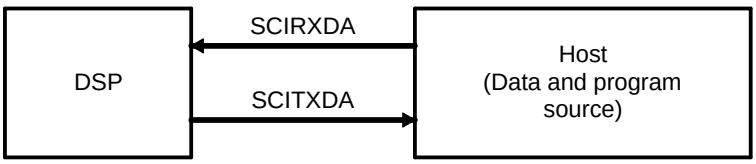
图 2-9. CopyData 函数概述




## 2.14 SCI_Boot 函数

SCI 引导模式可以将代码从 SCI-A 异步传输至内存储器。 此引导模式仅支持传入的 8 位数据流，以及遵循示例 2-2 概述的相同数据流。

图 2-10. SCI bootloader 操作概述




DSP 通过 SCI-A 外设与外部主机器件通信。 SCI 端口的自动波特特性用于锁定与主机通信的波特率。 因此，SCI loader 非常灵活，您可以使用多个不同的波特率与 DSP 通信。

在每个数据传输之后，DSP 会将收到的 8 位字符回波给主机。 通过这种方式，主机可以检查 DSP 是否收到了每个字符。

如果波特率较高，传入数据位的转换率则受收发器和连接器的性能影响。 虽然常规串行通信可以运作良好，但此转换率可能会限制在较高波特率（通常高于 100k 波特）时执行可靠的自动波特检测，并导致自动波特锁定特性失效。 为避免出现这种情况，建议执行以下操作：

1. 使用较低的波特率实现主机与 28x SCI bootloader 之间的波特锁定。
2. 此较低的波特率下加载传入的 28x 应用程序或定制的加载程序。

3. 然后主机与所加载的 28x 应用握手，以将 SCI 波特率寄存器设置为所需的高波特率。

图 2-11. SCI_Boot 函数概述

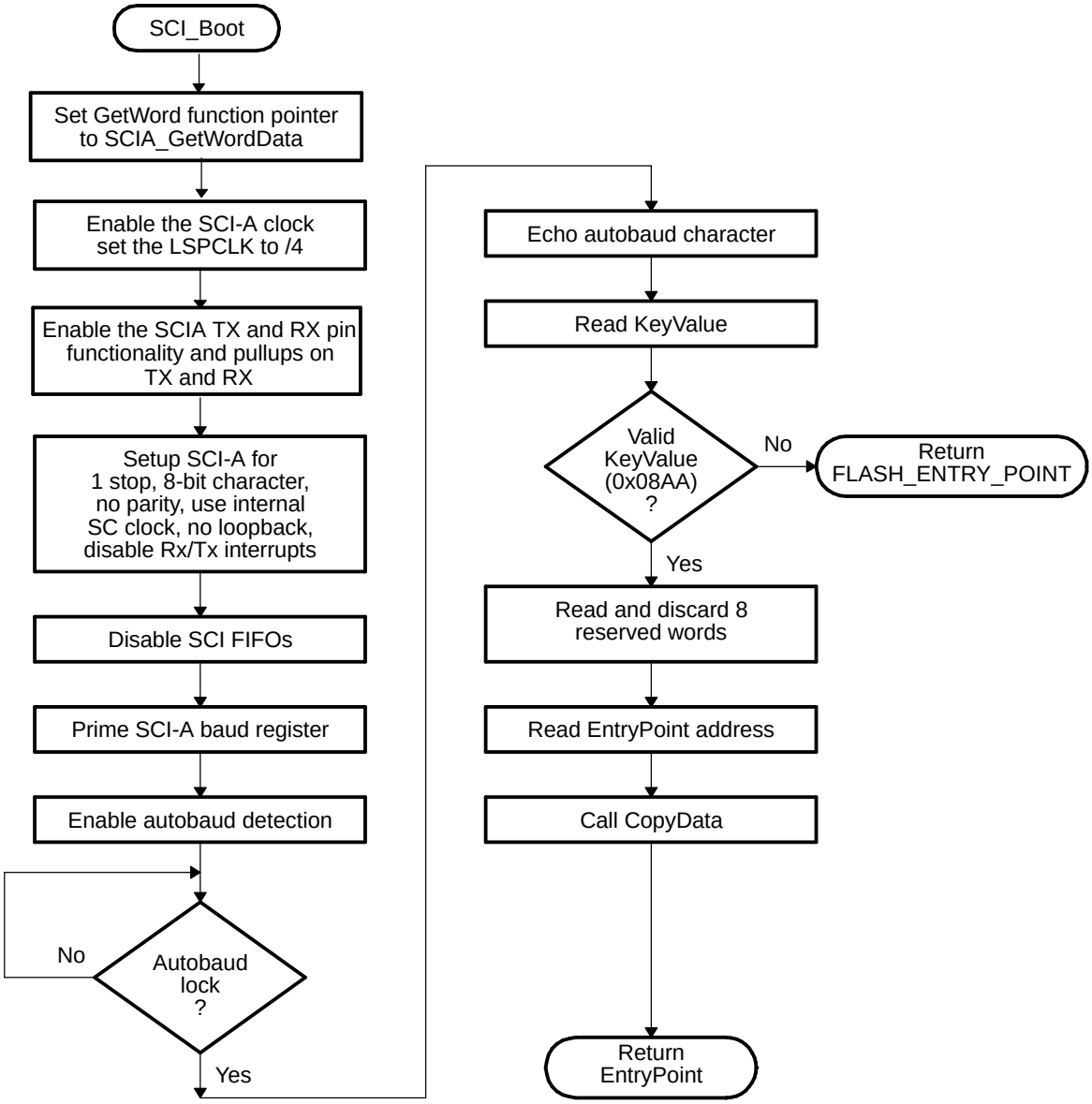
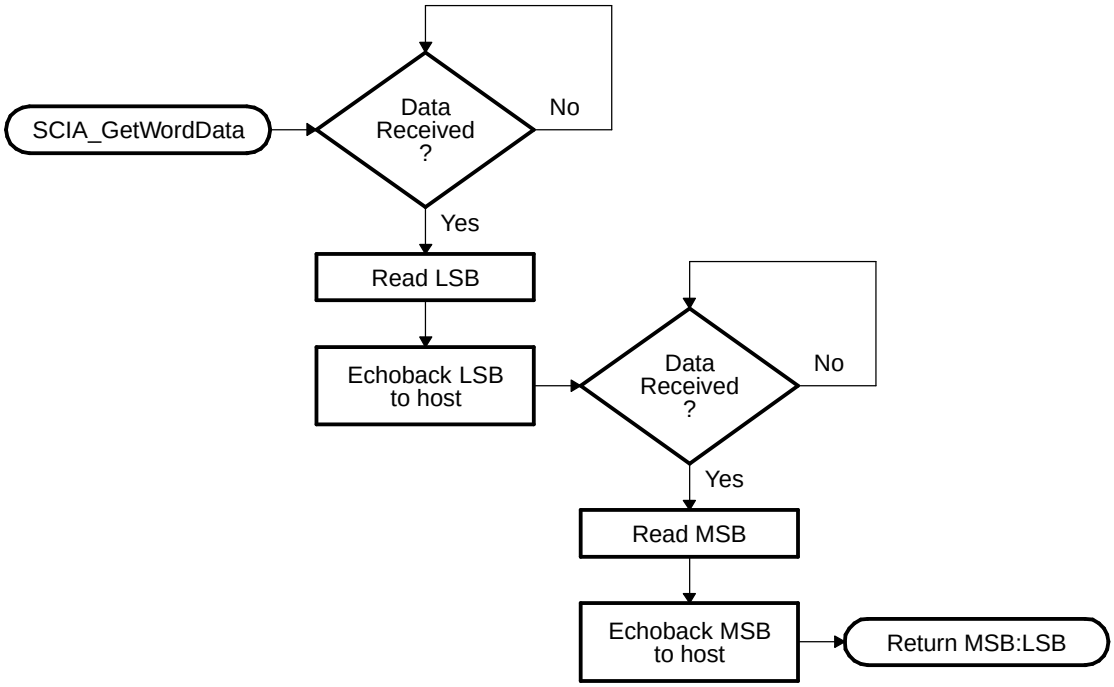


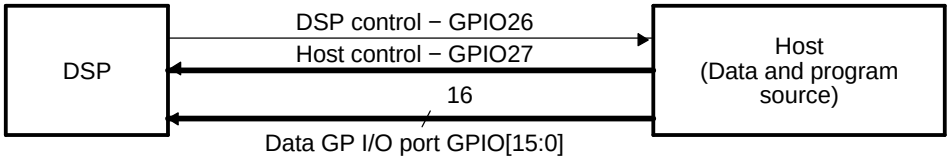
图 2-12. SCI_GetWordData 函数概述




## 2.15 Parallel_Boot 函数 (GPIO)

并行通用 I/O (GPIO) 引导模式可以将代码从 GPIO0-GPIO15 异步传输至内存储器。 每一个值的长度都可以为 16 位或 8 位，并遵循 “ 数据流结构 ” 中概述的相同数据流。

图 2-13. GPIO bootloader 操作概述

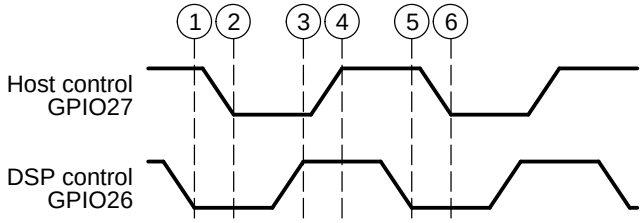



28x 通过轮询/驱动 GPIO27 和 GPIO26 线与外部主机器件通信。 必须使用图 2-14 所示的握手协议才能成功通过 GPIO[15:0] 传输每个字。 此协议相当强大可靠，可允许低速或高速的主机与 DSP 通信。

如果选择了 8 位模式，则会读取 2 个连续的 8 位字以组成一个 16 位字。 首先读取的是最高有效字节 (MSB)，然后是最低有效字节 (LSB)。 在此例中，将从 GPIO[7:0] 较低的 8 条线读取数据，并忽略较高位字节。

DSP 通过拉低 GPIO26 引脚首先向主机发出信号，指明它已做好开始执行数据传输的准备。 主机负载然后通过拉低 GPIO27 引脚来启动数据传输。 此完整协议如下图所示：

图 2-14. 并行 GPIO bootloader 握手协议




1. DSP 通过拉低 GPIO26 引脚指明它已做好开始接收数据的准备。
2. bootloader 一直等待，直至主机将数据放到 GPIO[15:0] 上。 主机通过拉低 GPIO27 引脚向 DSP 发出信

号，指明数据已准备就绪。

3. DSP 读取数据，并通过拉高 GPIO26 引脚向主机发出信号，指明已完成读取。
4. bootloader 一直等待，直至主机通过拉高 GPIO27 引脚对 DSP 做出确认。
5. DSP 再次拉低 GPIO26 引脚，指明它已做好接收更多数据的准备。

每发送一个数据值都重复此进程。

图 2-15 显示了并行 GPIO bootloader 流程的概述。

图 2-15. 并行 GPIO 模式概述

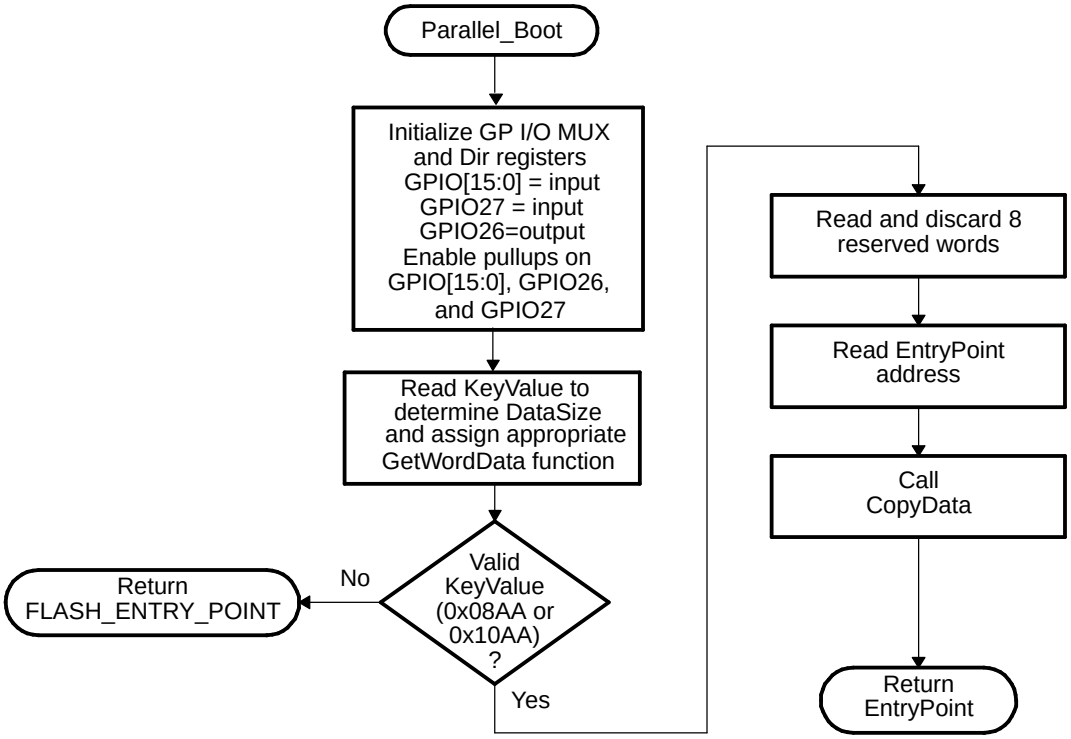



图 2-16 显示了从主机端发出的传输流。 在此模式下，CPU 和主机的操作速度不是非常重要，因为主机将等待 DSP，而 DSP 也会等待主机。 通过此方式，该协议可用于运行速度比 DSP 更快或更慢的主机。

图 2-16. 并行 GPIO 模式 - 主机传输流

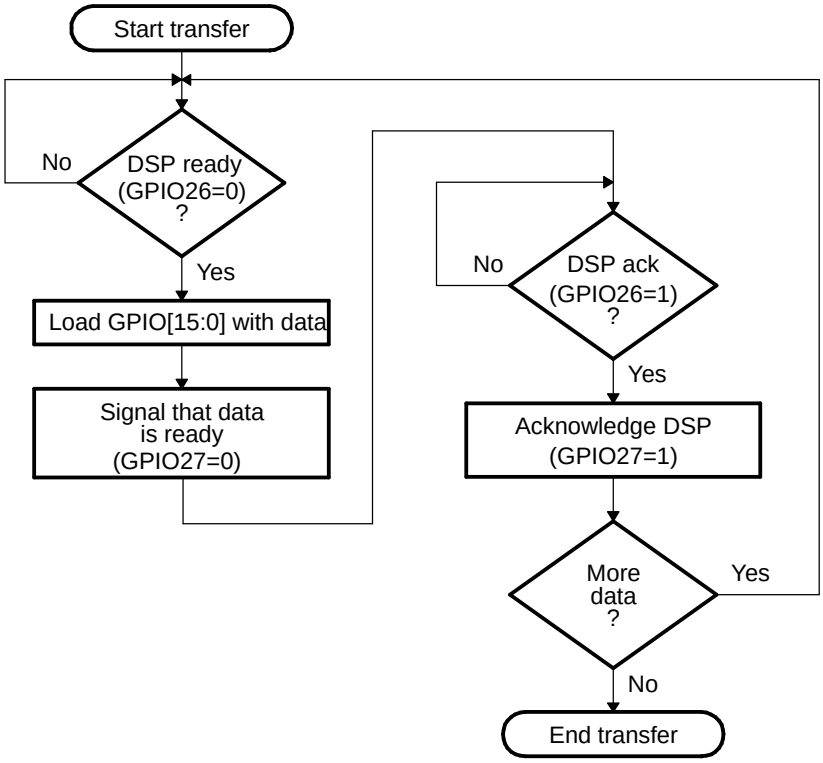



图 2-17 与图 2-18 显示了从并行端口读取数据的单个字时所采用的流程。 加载程序使用图 2-6 所示的方法读取键值，并确定传入的数据流宽度是 8 位还是 16 位。 此并行加载程序会根据传入的数据流的数据大小使用不同的 GetWordData 函数。

- 16 位数据流

  如果是 16 位数据流，将使用 Parallel_GetWordData16bit 函数。 此函数一次可以读取整个 16 位。 此函数的流程如图 2-17 所示。

- 8 位数据流

  如果是 8 位数据流，将使用 Parallel_GetWordData8bit 函数。 此 8 位例程（如图 2-18 所示）将丢弃从端口首次读取的高 8 位，并将低 8 位视为要获取的字的最低有效字节 (LSB)； 接着，该例程将执行第二次读取以获取最高有效字节 (MSB)； 然后将 MSB 和 LSB 组合成要传递回调用例程的单个 16 位值。

图 2-17. 16 位并行 GetWord 函数

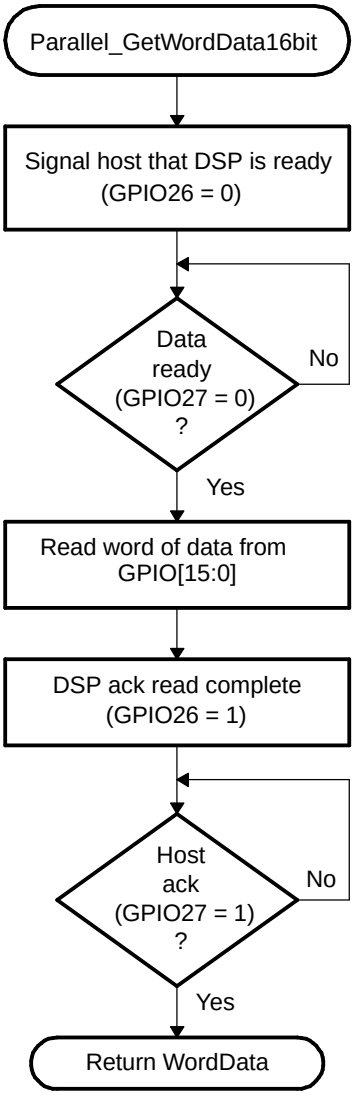
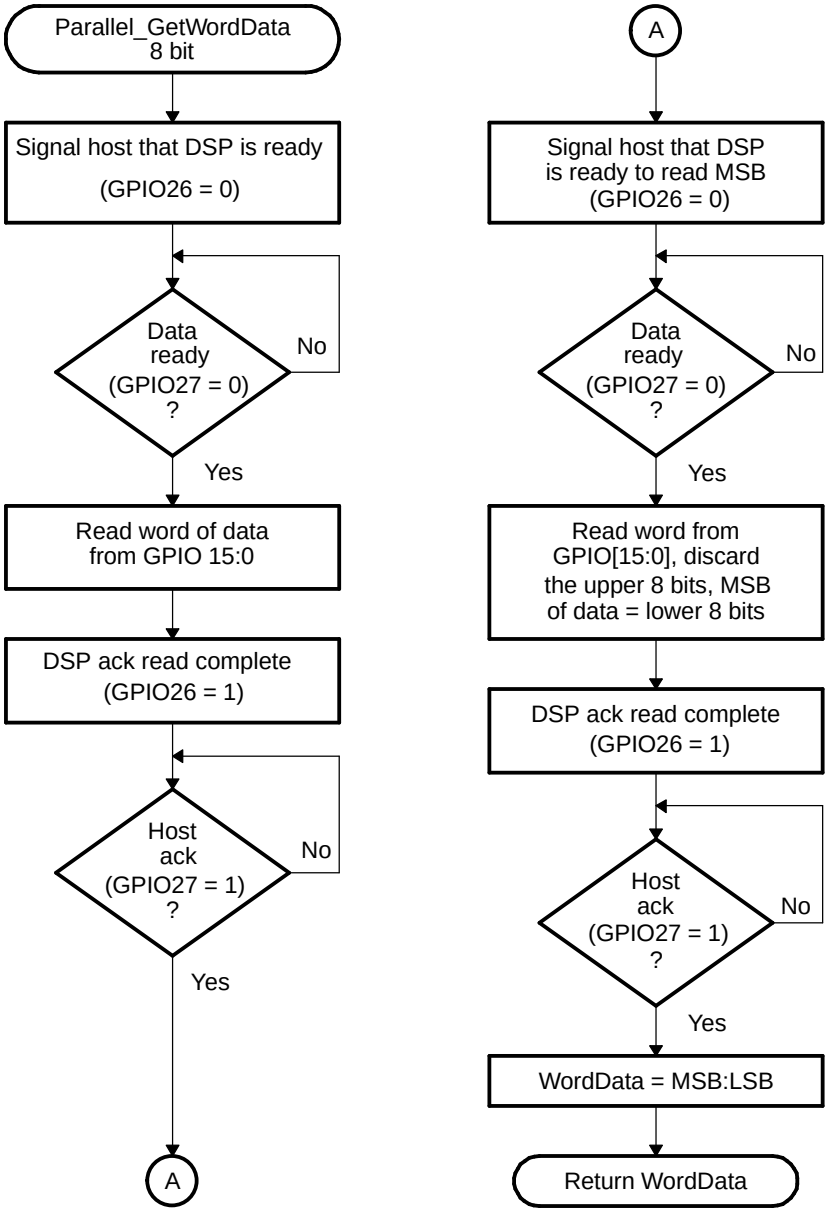


图 2-18. 8 位并行 GetWord 函数



## 2.16 SPI_Boot 函数

如图 2-19 所示，SPI 加载程序预计在 SPI-A 引脚上存在的是一个 8 位宽的与 SPI 兼容的串行 EEPROM 器件。此 SPI bootloader 不支持 16 位数据流。

图 2-19. SPI 加载程序


SPI 引导 ROM 加载程序将初始化要连接至串行 SPI EEPROM 的 SPI 模块。 此类器件包括，但不限于，Xicor X25320 (4Kx8) 和 Xicor X25256 (32Kx8) SPI 串行 SPI EEPROM。

SPI 引导 ROM 加载程序使用最低波特率，用以下设置初始化 SPI：启用 FIFO、8 位字符、内部 SPICLK 主模式和通话模式、时钟相位 = 0、极性 = 0。

如果要从另一器件上的 SPI 端口执行下载，则必须将该器件设置为在从模式下操作并模拟串行 SPI EEPROM；输入 SPI_Boot 函数之后，立即将 SPI 引脚的引脚功能设置为最高优先级并初始化 SPI。 请尽可能以最低速度执行初始化。 一旦初始化 SPI 并读取了键值，您就可以指定波特率或低速外设时钟方面的改动。

表 2-6. SPI 8 位数据流

| 字节 | 内容 |
|---|---|
| 1 | LSB：AA（存储器宽度为 8 位的键值） |
| 2 | MSB：08h（存储器宽度为 8 位的键值） |
| 3 | LSB: LOSPCP |
| 4 | MSB: SPIBRR |
| 5 | LSB：留作将来使用 |
| 6 | MSB：留作将来使用 |
| ... | ... |
| ... | ... |
| 17 | LSB：留作将来使用 |
| 18 | MSB：留作将来使用 |
| 19 | LSB：应用起点的上半部 PC[23:16] (MSW) |
| 20 | MSB：应用起点的上半部 PC[31:24] (MSW) （注意：始终为 0x00） |
| 21 | LSB：应用起点的下半部 PC[7:0] (LSW) |
| 22 | MSB：应用起点的下半部 PC[15:8] (LSW) |
| ... | ... |
| ... | ... |
| ... | 如通用数据流说明中所示的“大小/目的地址/数据”格式的数据块 |
| ... | ... |
| ... | ... |
| n | LSB: 00h |
| n+1 | MSB：00h - 表示源结束 |

从串行 SPI EEPROM 发出的数据传输在“突发”模式下完成。 传输完全以字节模式（8 位/字符的 SPI）执行。 下面列出了此顺序的逐步说明：

1. 初始化 SPI-A 端口
2. 将 GPIO19 (SPISTE) 引脚用作串行 SPI EEPROM 的片选
3. SPI-A 为串行 SPI EEPROM 输出一个读取命令
4. SPI-A 向串行 SPI EEPROM 发送地址 0x0000；即，主机要求 EEPROM 的可下载数据包必须从 EEPROM 中的地

址 0x0000 处开始。

5. 获取的下个字必须与 8 位数据流 (0x08AA) 的键值相匹配。

   此字的最低有效字节是首次读取的那个字节，而最高有效字节是获取的下一个字节。 在 SPI 中所有字传输都为真。

   如果键值不匹配，加载则中止，并将闪存的应用起点 (0x3F 7FF6) 返回至调用例程。

6. 接下来获取的两个字节可用于更改低速外设时钟寄存器 (LOSPCP) 和 SPI 波特率寄存器 (SPIBRR) 的值。读取的第一个字节是 LOSPCP 值，第二个字节是 SPIBRR 值。

   接下来的 7 个字留作将来增强时使用。 SPI bootloader 将读取这 7 个字，然后丢弃。

7. 接下来的 2 个字构成了 32 位应用起点地址，在完成引导加载进程之后，将继续从该地址执行程序； 它通常是通过 SPI 端口下载的程序的应用起点。

8. 通过 SPI 端口将多个代码块和数据块从外部串行 SPI EEPROM 复制到存储器中。 代码块按之前介绍的标准数据流结构组织在一起。 这一操作直至遇到 0x0000 块大小时才完成。 此时将应用起点地址返回至调用例程，后者然后退出 bootloader，并在所指定的地址处继续执行。

图 2-20. 从 EEPROM 发出的数据传输流程

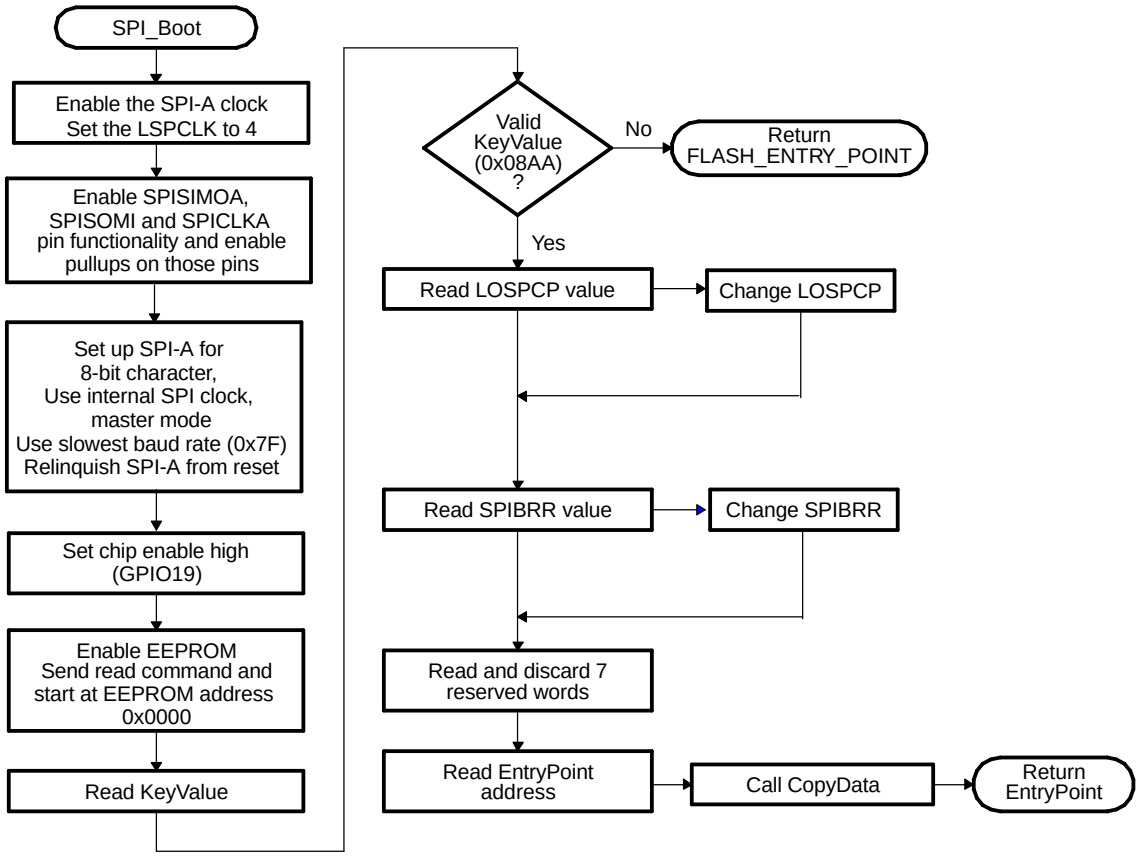
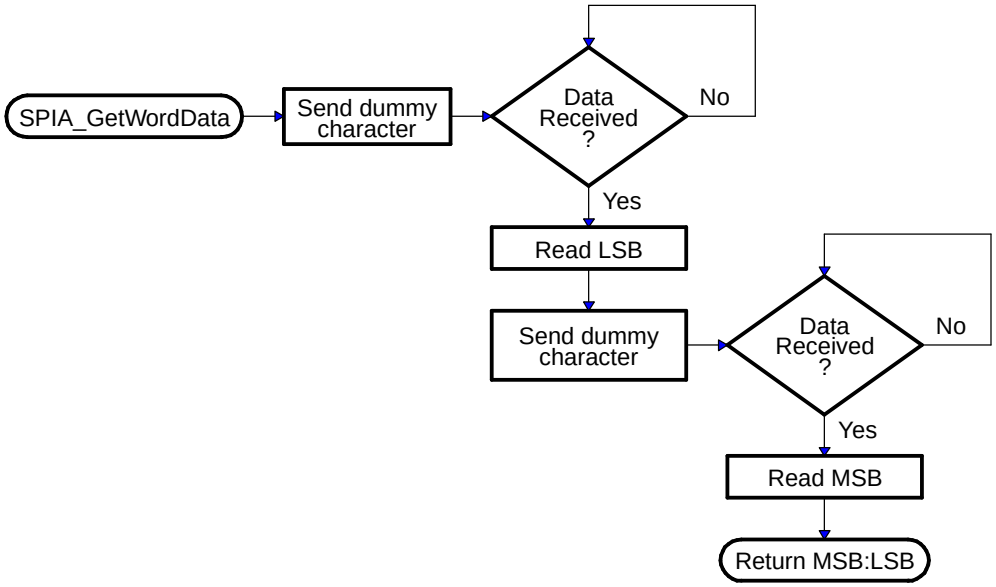


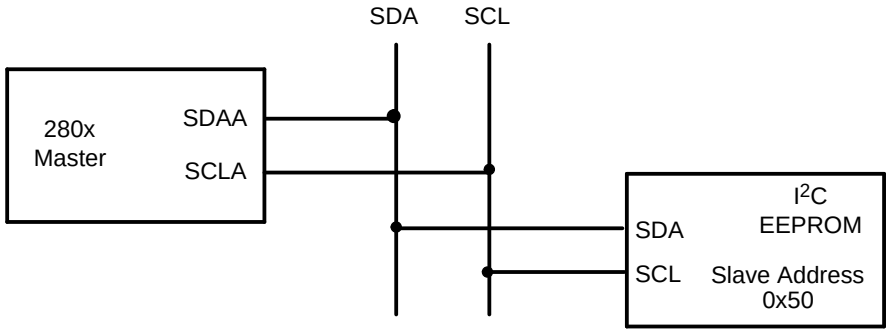
图 2-21. SPIA_GetWordData 函数概述




## 2.17 I²C Boot 函数

如图 2-22 所示，I²C bootloader 预计在 I²C-A 总线上的 0x50 地址处存在的是一个 8 位宽的与 I²C 兼容的 EEPROM 器件。 此 EEPROM 必须遵守采用 16 位基址结构的传统的 I²C EEPROM 协议（如此部分所述）。

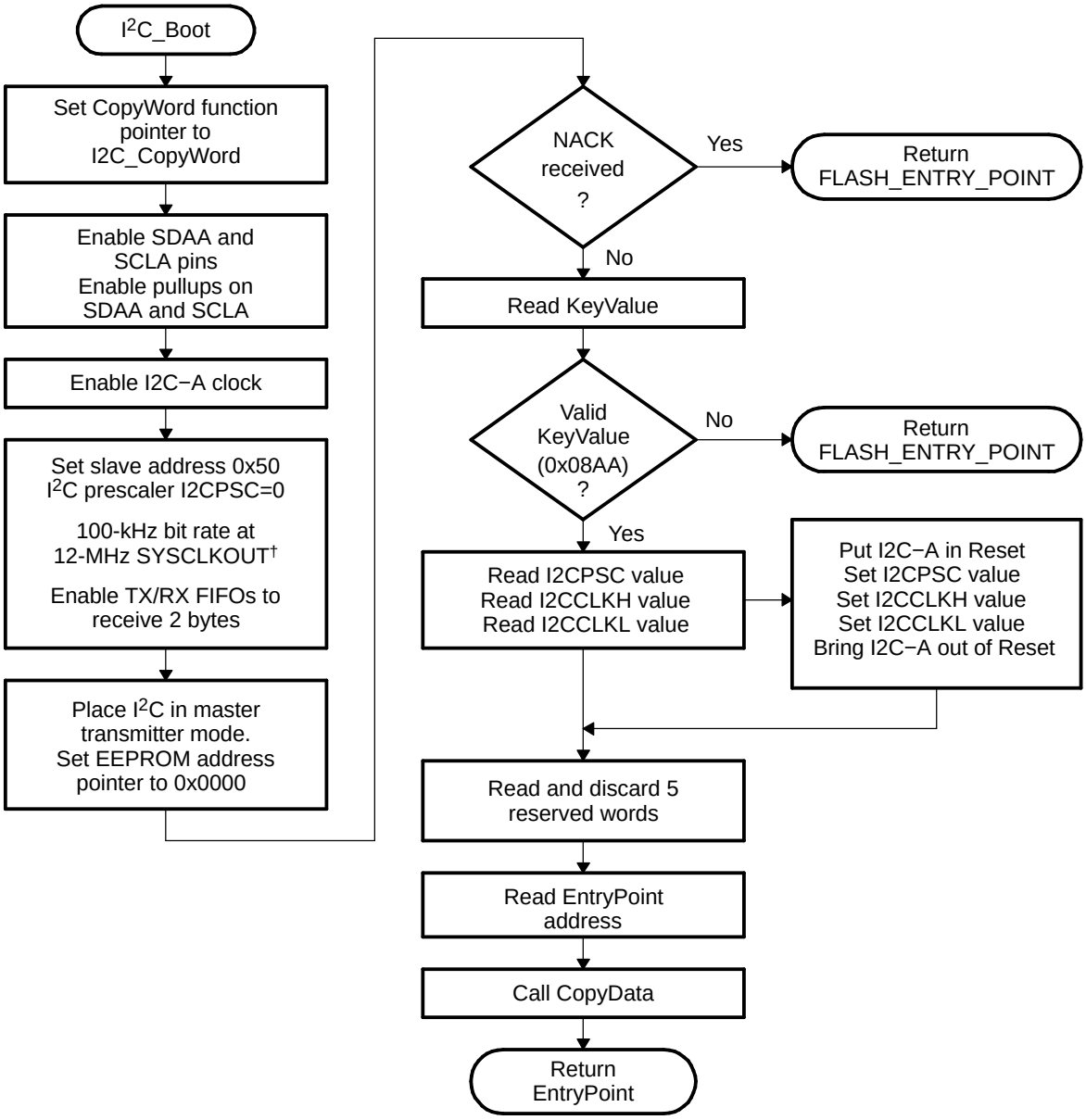
图 2-22. 地址 0x50 处的 EEPROM 器件




如果要从非 EEPROM 器件执行下载，则必须将该器件设置为在从模式下操作并模拟 I²C EEPROM。 输入 I²C Boot 函数之后，立即针对 I²C-A 操作配置 GPIO 引脚并初始化 I²C。 从 I²C 模块引导时，必须满足以下要求：

- 该器件的输入频率必须介于 14MHz 至 24MHz 之间
- EEPROM 必须在从属方地址 0x50 处

图 2-23. I2C_Boot 函数概述




† During device boot, SYSCLKOUT will be the device input frequency divided by two.

要使用 I²C-A bootloader，该器件的输入时钟频率必须介于 14MHz 至 24MHz 之间。 此输入时钟频率将产生默认的 7MHz 至 12MHz 的系统时钟 (SYSCLKOUT)。 默认情况下，bootloader 会将 I2CPSC 预分频值设置为 0，以便不从 SYSCLKOUT 分割出 I²C 时钟， 从而使 I²C 时钟介于 7MHz 至 12MHz 之间，这符合 I²C 外设时钟规范。 可以在从 EEPROM 接收前几个字节后修改 I2CPSC 值，但建议不这样做，因为这会导致 I²C 在所需规范之外操作。

当系统时钟为 12MHz 时，bootloader 将位周期预分频器（I2CCLKH 和 I2CCLKL）配置为以 50% 占空比、100kHz 比特率（标准 I²C 模式）运行 I²C。 可以在从 EEPROM 接收前几个字节之后修改这些寄存器， 从而允许在读取剩余数据时,通信速度提高至 400kHz 比特率（快速 I²C 模式）。

将不检查仲裁、总线忙和从属方信号。 因此，在此初始化阶段将不允许其它主控方控制总线。 如果在 I²C 引导模式期间应用需要另一个主控方，则必须将该主控方配置为延迟发送任何 I²C 消息，直至应用软件发出信号，指明已完成 bootloader 的初始化部分。

只有在发送第一条消息以初始化 EEPROM 基址期间才检查非确认位， 这样可确保 0x50 地址处存在 EEPROM 时才继续。 如果 EEPROM 不在该地址处，代码将跳转至闪存的应用起点。 在数据读取消息的地址阶段 (I2C_Get Word) 将不检查非确认位。 如果在数据读取消息期间收到非确认位，I²C 总线将暂停。 表 2-7 显示了 I²C 使用的 8 位数据流。

表 2-7. I²C 8 位数据流

| 字节 | 内容 |
|---|---|
| 1 | LSB：AA（存储器宽度为 8 位的键值） |
| 2 | MSB：08h（存储器宽度为 8 位的键值） |
| 3 | LSB: I2CPSC[7:0] |
| 4 | 保留 |
| 5 | LSB: I2CCLKH[7:0] |
| 6 | MSB: I2CCLKH[15:8] |
| 7 | LSB: I2CCLKL[7:0] |
| 8 | MSB: I2CCLKL[15:8] |
| ... | ... |
| ... | ... |
| 17 | LSB：留作将来使用 |
| 18 | MSB：留作将来使用 |
| 19 | LSB：应用起点的上半部 PC |
| 20 | MSB：应用起点的上半部 PC[22:16]（注意：始终为 0x00） |
| 21 | LSB：应用起点的下半部 PC[15:8] |
| 22 | MSB：应用起点的下半部 PC[7:0] |
| ... | ... |
| ... | ... |
| | 采用如通用数据流说明中所示的大小/目的地址/数据格式的数据块 |
| ... | ... |
| ... | ... |
| n | LSB: 00h |
| n+1 | MSB：00h - 表示源结束 |

I²C bootloader 所需的 I²C EEPROM 协议如图 2-24 与图 2-25 。 第一次通信（用于将 EEPROM 地址指针指向 0x0000 并从该处读取键值 (0x08AA)）如图 2-24 所示。 所有后续读取如图 2-25 所示，并且一次读取两个字节。

图 2-24. 随机读取

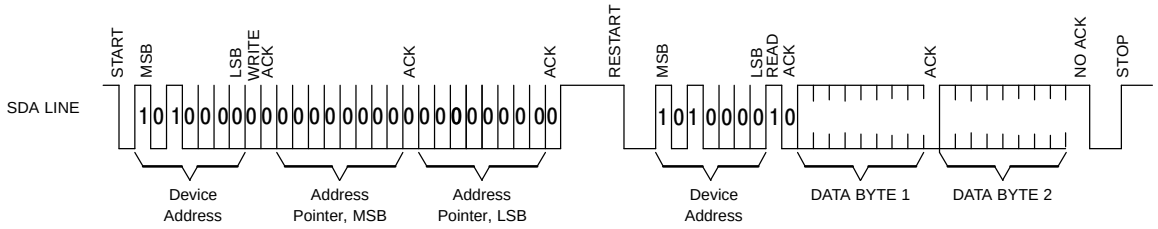
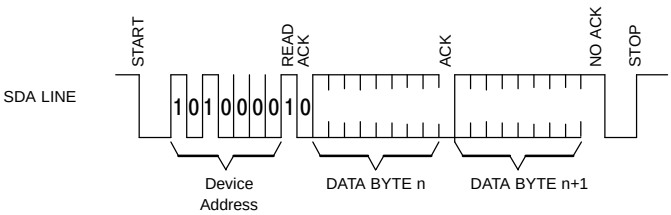


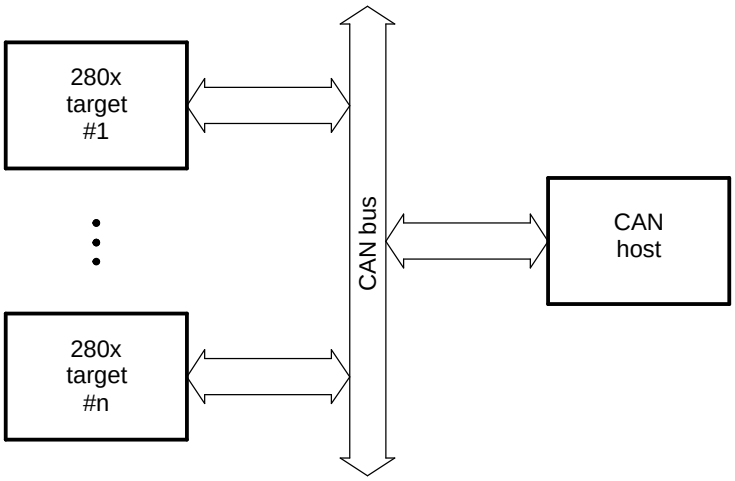

图 2-25. 顺序读取



## 2.18 eCAN Boot 函数

eCAN bootloader 将代码从 eCAN-A 异步传输至内存储器。 主机可以是任意 CAN 节点。 首先使用 11 位标准标识（具有一个等于 0x1 的 MSGID）按每数据帧两个字节完成通信。 如果需要更大的数据吞吐量，主机可以下载一个内核来重新配置 eCAN。

图 2-26. eCAN-A bootloader 操作概述




如表 2-8 所示以对应于不同 XCLKIN 值得到一个有效比特率的方式对位时序寄存器编程。

表 2-8. 对应于不同 XCLKIN 值的比特率值

| XCLKIN | SYSCLKOUT | 比特率 |
|---|---|---|
| 40Mhz | 20Mhz | 1Mbps |
| 20Mhz | 10Mhz | 500kbps |
| 10Mhz | 5Mhz | 250kbps |
| 5Mhz | 2.5Mhz | 125kbps |

所示的 SYSCLKOUT 值是采用默认 PLL 设置时的复位值。 $BRP_{reg}$ 和位时序值已被分别硬编码成 1 和 10。

邮箱 1 使用等于 0x1 的标准 MSGID 进行编程，用于 boot-loader 通信。 CAN 主机应当一次只发送 2 个字节，首先发送 LSB，然后是 MSB。 例如，要将字 0x08AA 发送至 280x，则首先发送 AA，然后是 08。 CAN bootloader 的程序流程与 SCI bootloader 相同。 CAN bootloader 的数据顺序如表 2-9 所示 所示：

表 2-9. eCAN 8 位数据流

| | | |
|---|---|---|
| AA | 08 | 键值：0x08AA |
| 00 | 00 | 8 个保留字流的一部分 |
| 00 | 00 | 8 个保留字流的一部分 |
| 00 | 00 | 8 个保留字流的一部分 |
| 00 | 00 | 8 个保留字流的一部分 |
| 00 | 00 | 8 个保留字流的一部分 |
| 00 | 00 | 8 个保留字流的一部分 |
| 00 | 00 | 8 个保留字流的一部分 |
| 00 | 00 | 8 个保留字流的一部分 |
| bb | aa | 32 位地址的最高有效 (MSW) 部分 (aabb) |

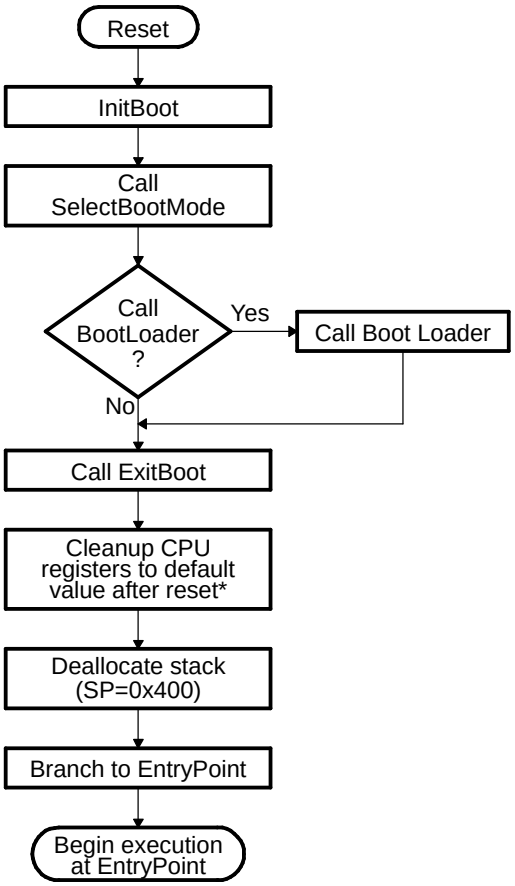
表 2-9. eCAN 8 位数据流(接上表)

| | | |
|---|---|---|
| dd | cc | 32 位地址的最低有效 (LSW) 部分 (ccdd) -<br>最终应用起点地址 = 0xaabbccdd |
| nn | mm | 第一部分的长度 (mmnn) |
| ff | ee | 32 位地址的 MSW 部分 (eeff) |
| hh | hh | 32 位地址的 LSW 部分 (gghh) - 第一部分的起始地址 = 0xeeffgghh |
| xx | xx | 第一部分的第一个字 |
| xx | xx | 第一部分的第二个字 |
| | ... | ... |
| | ... | ... |
| | ... | ... |
| xx | xx | 第一部分的最后一个字 |
| nn | mm | 第二部分的长度 (mmnn) |
| ff | ee | 32 位地址的 MSW 部分 (eeff) |
| hh | gg | 32 位地址的 LSW 部分 (gghh) - 第二部分的起始地址 = 0xeeffgghh |
| xx | xx | 第二部分的第一个字 |
| xx | xx | 第二部分的第二个字 |
| | ... | ... |
| | ... | ... |
| xx | xx | 第二部分的最后一个字<br>（如有必要，可使用更多部分） |
| 00 | 00 | 下一部分的长度为零则表示数据结束。 |

## 2.19 ExitBoot 汇编例程

引导 ROM 包含一个用于在复位时将 CPU 寄存器恢复为默认状态的 ExitBoot 例程。 除一个寄存器外，它将在所有寄存器上执行。 ST1 中的 OBJMODE 位保留设置，以便器件仍保留针对 C28x 操作进行的配置。 下图详细列出了此流程：

图 2-27. ExitBoot 步骤流程




下列 CPU 寄存器将恢复为默认值：

- ACC = 0x0000 0000
- RPC = 0x0000 0000
- P = 0x0000 0000
- XT = 0x0000 0000
- ST0 = 0x0000
- ST1 = 0x0A0B
- XAR0 = XAR7 = 0x0000 0000

完成 ExitBoot 例程并且程序流程重定向至应用起点地址之后，CPU 寄存器将具有下列值：

表 2-10. CPU 寄存器的恢复值

| 寄存器 | 值 | | | 寄存器 | 值 | | |
|---|---|---|---|---|---|---|---|
| ACC | 0x0000 0000 | | | P | 0x0000 0000 | | |
| XT | 0x0000 0000 | | | RPC | 0x00 0000 | | |
| XAR0-XAR7 | 0x0000 0000 | | | DP | 0x0000 | | |
| ST0 | 0x0000 | 15:10 | OVC = 0 | ST1 | 0x0A0B | 15:13 | ARP = 0 |
| | | 9:7 | PM = 0 | | | 12 | XF = 0 |
| | | 6 | V = 0 | | | 11 | M0M1MAP = 1 |
| | | 5 | N = 0 | | | 10 | 保留 |
| | | 4 | Z = 0 | | | 9 | OBJMODE = 1 |
| | | 3 | C = 0 | | | 8 | AMODE = 0 |
| | | 2 | TC = 0 | | | 7 | IDLESTAT = 0 |
| | | 1 | OVM = 0 | | | 6 | EALLOW = 0 |
| | | 0 | SXM = 0 | | | 5 | LOOP = 0 |
| | | | | | | 4 | SPA = 0 |
| | | | | | | 3 | VMAP = 1 |
| | | | | | | 2 | PAGE0 = 0 |
| | | | | | | 1 | DBGM = 1 |
| | | | | | | 0 | INTM = 1 |

# 构建引导表

## 3.1 C2000 十六进制实用程序

要使用 bootloader 的特性，您必须按第 2.9 部分 中所述生成一个数据流和引导表。 28x 代码生成工具所附带的十六进制转换实用工具可以生成所需的数据流，其中包括所需的引导表。 本部分描述了 hex2000 实用程序。 第 3.2 部分中介绍了 hex2000 执行的文件转换示例。

此十六进制实用程序支持为 SCI、SPI、$I^2C$、eCAN 和并行 I/O 加载程序创建所需的引导表。 即，十六进制实用程序会将键值、保留位、应用起点、地址、块起始地址、块长度和终止值等必需信息添加至该文件中。 引导表的内容可能随引导模式以及在运行十六进制转换实用程序时选择的选项而略有不同。 主机所需的实际文件格式（ASCII、二进制、十六进制等）将随具体应用而有所不同，并且可能需要执行一些附加转换。

要构建引导表，请执行以下步骤：

1. 汇编或编译代码。

   将创建对象文件，连接程序然后使用该文件创建一个单一的输出文件。

2. 链接此文件。

   连接程序以常用对象文件格式 (COFF) 将所有对象文件组合成一个单一的输出文件。 连接程序使用指定的连接程序命令文件，将代码部分分配至不同的内存快。 引导表中的每个数据块都与 COFF 文件中已初始化的部分相对应。 十六进制转换实用程序将不转换未初始化的部分。 以下选项可能很有用：

   连接程序的 -m 选项可用于生成映射文件。 此映射文件将显示所有已创建的部分、这些部分在存储器中的位置及其长度。 当要检查此文件以确保已初始化的部分正是您预计的部分时，此映射文件很有用。

   连接程序的 -w 选项同样非常有用。 此选项将告诉您连接程序是否在存储器区域中独自分配了一个部分。 例如，在名为 ramfuncs 的代码中有一个部分。

3. 运行十六进制转换实用程序。

   为所需引导模式选择适当的选项，然后运行十六进制转换实用程序以将连接程序产生的 COFF 文件转换为一个引导表。

有关编译和链接进程的详情，请参阅 *TMS320C28x 汇编语言工具用户指南* (SPRU513) 和 *TMS320C28x 优化 C/C++ 编译器用户指南* (SPRU514)。

表 3-1 概述了 bootloader 可用的十六进制转换实用程序选项。 有关用于生成引导表的 hex2000 操作的详细说明，请参阅 *TMS320C28x 汇编语言工具用户指南* (SPRU513)。 将进行更新以支持 $I^2C$ 引导。 请参阅 Codegen 发布说明以了解最新信息。

表 3-1. Boot-Loader 选项

| 选项 | 说明 |
|---|---|
| -boot | 将所有部分转换成可引导的表单（而不是使用 SECTIONS 指令） |
| -sci8 | 指定 bootloader 表源作为 SCI-A 端口，8 位模式 |
| -spi8 | 指定 bootloader 表源作为 SPI-A 端口，8 位模式 |
| -gpio8 | 指定 bootloader 表源作为 GPIO 端口，8 位模式 |
| -gpio16 | 指定 bootloader 表源作为 GPIO 端口，16 位模式 |
| -bootorg value | 指定 bootloader 表的源地址 |
| -lospcp value | 为 LOSPCP 寄存器指定初始值。 此值仅适用于 spi8 引导表格式，对于所有其它格式，此值被忽略。 如果此值大于 0x7F，则被截断为 0x7F。 |
| -spibrr value | 为 SPIBRR 寄存器指定初始值。 此值仅适用于 spi8 引导表格式，对于所有其它格式，此值被忽略。 如果此值大于 0x7F，则被截断为 0x7F。 |
| -e value | 指定引导加载后开始执行的应用起点。 此值可以是一个地址，也可以是一个全局符号。 此值为可选。 此应用起点可在编译时使用连接程序的 -e 选项定义，以将应用起点分配至一个全局符号。 除非用连接程序的 -e 选项定义，否则 C 程序的应用起点通常为 _c_int00。 |
| -i2c8 | 指定 bootloader 表源为 I²C-A 端口，8 位。 |
| -i2cpsc value | 为 I2CPSC 寄存器指定值。 将在加载所有 I²C 选项之后、从 EEPROM 读取数据之前加载此值并生效。 此值将被截断为最低有效 8 位，并且应当设置为维持 7 - 12MHz 的 I²C 模块时钟。 |
| -i2cclkh value | 为 I2CCLKH 寄存器指定值。 将在加载所有 I²C 选项之后、从 EEPROM 读取数据之前加载此值并生效。 |
| -i2cclkl value | 为 I2CCLKL 寄存器指定值。 将在加载所有 I²C 选项之后、从 EEPROM 读取数据之前加载此值并生效。 |

## 3.2 示例：为 eCAN 引导加载准备 COFF 文件

此部分显示如何将 COFF 文件转换为适用于基于 CAN 的引导加载的格式。 此示例假设发送数据流的主机可以读取 ASCII 十六进制格式文件。 已经使用名为 GPIO34TOG.out 的示例 COFF 文件执行转换。

使用连接程序的 -m 选项建立项目和链接以生成映射文件。 检查由连接程序生成的 .map 文件。 已从示例映射文件 (GPIO34TOG.map) 复制示例 3-1 所示的信息。 这显示的是代码的部分分配映射。 映射文件包括以下信息：

- 输出部分
  这是使用连接程序命令文件中的 SECTIONS 指令指定的输出部分的名称。
- 起点
  为每个输出部分列出的第一个起点是整个输出部分的起始地址。 随后列出的起点值是输出部分中该部分的起始地址。
- 长度
  为每个输出部分列出的第一个长度是整个输出部分的长度。 随后列出的长度值是与输出部分中的该部分相关的长度。
- 属性/输入部分
  列出的输入文件是整个部分的组成部分，或是与某一输出部分相关的任意值。

有关生成连接程序命令文件和存储器映射的详情，请参阅 *TMS320C28x 汇编语言工具用户指南* (SPRU513)。

示例 3-1 所示的已初始化的所有部分需要加载到 DSP，才能正确执行代码。 在此例中，需要加载 codestart、ramfuncs、.cinit、myreset 和 .text 部分。 其它部分并未初始化，而且将不包括在加载进程中。 映射文件还指明每个部分的大小及起始地址。 例如，此 .text 部分具有 0x155 个字，在 0x3FA000 处开始。

*示例 3-1. GPIO34TOG 映射文件*

```
output                                  attributes/
section   page    origin      length      input sections
--------  ----  ----------  ----------   ----------------
codestart
*          0     00000000    00000002
                 00000000    00000002     DSP280x_CodeStartBranch.obj (codestart)
.pinit     0     00000002    00000000
.switch    0     00000002    00000000     UNINITIALIZED
```

*示例 3-1. GPIO34TOG 映射文件(接上表)*

```
ramfuncs    0     00000002    00000016
                  00000002    00000016     DSP280x_SysCtrl.obj (ramfuncs)
.cinit      0     00000018    00000019
                  00000018    0000000e     rts2800_ml.lib : exit.obj (.cinit)
                  00000026    0000000a                    : _lock.obj (.cinit)
                  00000030    00000001     --HOLE-- [fill = 0]
myreset     0     00000032    00000002
                  00000032    00000002     DSP280x_CodeStartBranch.obj (myreset)
IQmath      0     003fa000    00000000     UNINITIALIZED
.text       0     003fa000    00000155
                  003fa000    00000046     rts2800_ml.lib : boot.obj (.text)
```

要使用 CAN bootloader 加载此代码，主机必须以 bootloader 理解的格式发送数据。 也就是说，数据必须作为数据块发送，并且采用大小、起始地址、然后跟随数据的格式。 块大小为 0 表示数据结束。 HEX2000.exe 实用程序可用于将 COFF 文件转换成一种包括此引导信息的格式。 以下命令语法已用于将此应用转换成一个 ASCII 十六进制格式文件，其中包括 bootloader 所需的所有信息：

*示例 3-2. HEX2000.exe 命令语法*

```
C: HEX2000 GPIO34TOG.OUT -boot -gpio8 -a



Where:
- boot    Convert all sections into bootable form.
- gpio8   Use the GPIO in 8-bit mode data format. The eCAN
          uses the same data format as the GPIO in 8-bit mode.
- a       Select ASCII-Hex as the output format.
```

示例 3-2 所示的命令行将生成一个名为 GPIO34TOG.a00 的 ASCII 十六进制输出文件，其内容在示例 3-3 中说明。 此示例假设主机可以读取 ASCII 十六进制格式文件。 此格式可能不同于您的应用。已加载数据的每个部分都可以连接回示例 3-1 中所述的映射文件中。 加载数据流之后，引导 ROM 将跳转至应用起点地址，此地址已作为该数据流的一部分读取。 在此例中，将在 0x3FA0000 处开始执行。

*示例 3-3. GPIO34TOG 数据流*

```
AA 08                                  ;Keyvalue
00 00 00 00 00 00 00 00                ;8 reserved words
00 00 00 00 00 00 00 00
3F 00 00 A0                            ;Entrypoint 0x003FA000
02 00                                  ;Load 2 words - codestart section
00 00 00 00                            ;Load block starting at 0x000000
7F 00 9A A0                            ;Data block 0x007F, 0xA09A
16 00                                  ;Load 0x0016 words - ramfuncs section
00 00 02 00                            ;Load block starting at 0x000002
22 76 1F 76 2A 00 00 1A 01 00 06 CC F0 ;Data = 0x7522, 0x761F etc...
FF 05 50 06 96 06 CC FF F0 A9 1A 00 05
06 96 04 1A FF 00 05 1A FF 00 1A 76 07
F6 00 77 06 00
55 01                                  ;Load 0x0155 words - .text section
3F 00 00 A0                            ;Load block starting at 0x003FA000
AD 28 00 04 69 FF 1F 56 16 56 1A 56 40 ;Data = 0x28AD, 0x4000 etc...
29 1F 76 00 00 02 29 1B 76 22 76 A9 28
18 00 A8 28 00 00 01 09 1D 61 C0 76 18
00 04 29 0F 6F 00 9B A9 24 01 DF 04 6C
04 29 A8 24 01 DF A6 1E A1 F7 86 24 A7
06 .. ..
.. .. ..
.. .. ..
FC 63 E6 6F
19 00                                  ;Load 0x0019 words - .cinit section
00 00 18 00                            ;Load block starting at 0x000018
FF FF 00 B0 3F 00 00 00 FE FF 02 B0 3F ;Data = 0xFFFF, 0xB000 etc...
00 00 00 00 00 FE FF 04 B0 3F 00 00 00
00 00 FE FF .. .. ..
.. .. ..
3F 00 00 00
02 00                                  ;Load 0x0002 words - myreset section
00 00 32 00                            ;Load block starting at 0x000032
00 00 00 00                            ;Data = 0x0000, 0x0000
00 00                                  ;Block size of 0 - end of data
```

## 4.1 Bootloader 代码修订历史记录

此部分列出了 280x 引导 ROM 软件的修订历史记录。 引导 ROM 包含的版本号可用于识别应将哪个软件版本编程到引导 ROM 中。 此版本号和发布日期信息可以从下列存储器位置读取：

表 4-1. Bootloader 修订信息

| 地址 | 内容 |
|---|---|
| 0x3F FFB9 | 闪存 API 芯片兼容性检查。 某些版本的闪存 API 将使用此位置，以确保是在兼容的芯片版本上运行。 |
| 0x3F FFBA | 引导 ROM 的版本号 |
| 0x3F FFBB | 发布时间 MM/YY（十进制） |
| 0x3F FFBC | 校验和的最低有效字 |
| 0x3F FFBD | ... |
| 0x3F FFBE | ... |
| 0x3F FFBF | 校验和的最高有效字 |

- 版本 3；发布日期：2005 年 4 月
  版本 3 中进行了以下更改：
  - 闪存 API 芯片兼容性位置 (0x3F FFB9) 的内容从 0xFFFF 更改为 0xFFFE。
  - 更新了版本号、发布日期以及校验和存储器位置，以反映此新版本信息。
- 版本：2；发布日期：2005 年 1 月
  版本 2 中进行了以下更改：
  - 更新了版本号、发布日期以及校验和存储器位置，以反映此新版本信息。
  - 更新了 eCAN-A bootloader，以便正确初始化 MSGID1 寄存器的 IDE 和 AME 位。
  - SCI-A、SPI-A、I2C-A 和 eCAN-A 外设的输入配置现在已配置为异步输入（在调用适当的 bootloader 时）。 在上一版本中，这些输入是在与 SYSCLKOUT 同步的默认模式下进行配置的。
  - 引导模式选择例程现在会在 PLLSTS 寄存器中检查缺失时钟检测位 (MCLKSTS)，以确定 PLL 是否在跛行模式下操作。 如果 PLL 在跛行模式下操作，引导模式选择功能则根据所选的引导模式采取操作：
    - 引导至闪存、OTP、SARAM、I2C-A、SPI-A 以及并行 I/O 的表现正常。 如果设置了 MCLKSTS 位，用户的软件必须检查是否缺少时钟状态并采取相应的操作。
    - 将执行 SCI-A 引导，然而根据所请求的波特率，此器件可能无法进行自动波特锁定。 在此例中，引导 ROM 软件将在自动波特锁定函数中无限循环。 如果 SCI-A 引导完成，用户的软件必须检查是否缺少时钟状态，并采取相应的操作。
    - 将不执行引导至 eCAN-A。 引导 ROM 将无限循环。
- 版本：1；发布日期：2004 年 8 月：
  280x 引导 ROM 的初始版本。 此版本具有以下已知问题：
  - eCAN-A bootloader 不初始化 MSGID1 寄存器的 IDE 和 AME 位。 由于这些位可能为 1 或 0，因此可能接收由主机发送的帧，也可能不接收。 通过在运行 e-CAN bootloader 之前手动初始化此寄存器，可以将该 bootloader 用于软件开发。
  - SCI-A、SPI-A、I2C-A 和 eCAN-A 外设的输入配置是在与 SYSCLKOUT 同步的默认模式下进行配置的。 这在下一版本中将更改为异步模式。 bootloader. Thebootloader. The

## 4.2 Bootloader 代码列表

以下代码列表用于 280x Bootloader 3.0 版本。 要确定 bootloader 代码的版本，请检查引导 ROM 中存储器地址 0x3F FFBA 的内容。

```
// TI File $Revision: /main/2 $
// Checkin $Date: January 10, 2005   14:45:35 $
//###########################################################################
//
// FILE:   F280x_Boot.h
//
// TITLE:  F280x Boot ROM Definitions.
//
//###########################################################################
// $TI Release:$
// $Release Date:$
//###########################################################################
#ifndef F280X_BOOT_H
#define F280X_BOOT_H
//---------------------------------------------------------------------------
// Fixed boot entry points:
//
#define FLASH_ENTRY_POINT 0x3F7FF6
#define OTP_ENTRY_POINT   0x3D7800
#define RAM_ENTRY_POINT   0x000000
#define PASSWORD_LOCATION 0x3F7FF8
#define ERROR                    1
#define NO_ERROR                 0
#define EIGHT_BIT                8
#define SIXTEEN_BIT             16
#define EIGHT_BIT_HEADER   0x08AA
#define SIXTEEN_BIT_HEADER 0x10AA
typedef Uint16 (* uint16fptr)();
extern uint16fptr GetWordData;
#endif  // end of F280x_BOOT_H definition
```

```
;; TI File $Revision: /main/6 $
;; Checkin $Date: April 21, 2005   16:00:01 $
;;###########################################################################
;;
;; FILE:    Init_Boot.asm
;;
;; TITLE:   280x Boot Rom Initialization and Exit routines.
;;
;; Functions:
;;
;;     _InitBoot
;;     _ExitBoot
;;
;; Notes:
;;
;;###########################################################################
;; $TI Release:$
;; $Release Date:$
;;###########################################################################
    .def _InitBoot
    .ref _SelectBootMode
    .sect ".Flash"   ; Flash API checks this for
    .word 0xFFFE     ; silicon compatibility

    .sect ".Version"
    .word 0x0003     ; 280x Boot ROM Version 3
    .word 0x0405     ; Month/Year: (4/05 = April 2005)

    .sect ".Checksum"; 64-bit Checksum
    .long 0x6A78A069 ; least significant 32-bits
    .long 0x000003B5 ; most significant 32-bits

    .sect ".InitBoot"
;---------------------------------------------
; _InitBoot
;---------------------------------------------
;---------------------------------------------
; This function performs the initial boot routine
; for the boot ROM.
;
; This module performs the following actions:
;
;     1) Initializes the stack pointer
;     2) Sets the device for C28x operating mode
;     3) Calls the main boot functions
;     4) Calls an exit routine
;---------------------------------------------
_InitBoot:

; Initialize the stack pointer.
__stack:    .usect ".stack",0
    MOV SP, #__stack ; Initialize the stack pointer
; Initialize the device for running in C28x mode.
    C28OBJ       ; Select C28x object mode
    C28ADDR      ; Select C27x/C28x addressing
    C28MAP       ; Set blocks M0/M1 for C28x mode
    CLRC PAGE0   ; Always use stack addressing mode
    MOVW DP,#0   ; Initialize DP to point to the low 64 K
    CLRC OVM
; Set PM shift of 0
    SPM 0
; Decide which boot mode to use
    LCR  _SelectBootMode

; Cleanup and exit.  At this point the EntryAddr
; is located in the ACC register
```

```
    BF  _ExitBoot,UNC
;-----------------------------------------------
; _ExitBoot
;-----------------------------------------------
;-----------------------------------------------
;This module cleans up after the boot loader
;
; 1) Make sure the stack is deallocated.
;    SP = 0x400 after exiting the boot
;    loader
; 2) Push 0 onto the stack so RPC will be
;    0 after using LRETR to jump to the
;    entry point
; 2) Load RPC with the entry point
; 3) Clear all XARn registers
; 4) Clear ACC, P and XT registers
; 5) LRETR - this will also clear the RPC
;    register since 0 was on the stack
;-----------------------------------------------
_ExitBoot:
;-----------------------------------------------
;   Insure that the stack is deallocated
;-----------------------------------------------
    MOV SP,#__stack
;-----------------------------------------------
; Clear the bottom of the stack.  This will end up
; in RPC when you are finished
;-----------------------------------------------
    MOV  *SP++,#0
    MOV  *SP++,#0

;-----------------------------------------------
; Load RPC with the entry point as determined
; by the boot mode.  This address will be returned
; in the ACC register.
;-----------------------------------------------
    PUSH ACC
    POP  RPC

;-----------------------------------------------
; Put registers back in their reset state.
;
; Clear all the XARn, ACC, XT, and P and DP
; registers
;
; NOTE: Leave the device in C28x operating mode
;       (OBJMODE = 1, AMODE = 0)
;-----------------------------------------------
    ZAPA
    MOVL  XT,ACC
    MOVZ  AR0,AL
    MOVZ  AR1,AL
    MOVZ  AR2,AL
    MOVZ  AR3,AL
    MOVZ  AR4,AL
    MOVZ  AR5,AL
    MOVZ  AR6,AL
    MOVZ  AR7,AL
    MOVW  DP, #0
;-----------------------------------------------
;   Restore ST0 and ST1.  Note OBJMODE is
;   the only bit not restored to its reset state.
;   OBJMODE is left set for C28x object operating
;   mode.
;
;  ST0 = 0x0000     ST1 = 0x0A0B
```

```
;  15:10 OVC = 0    15:13      ARP = 0
;   9: 7  PM = 0       12       XF = 0
;      6   V = 0       11  M0M1MAP = 1
;      5   N = 0       10  reserved
;      4   Z = 0        9  OBJMODE = 1
;      3   C = 0        8    AMODE = 0
;      2  TC = 0        7 IDLESTAT = 0
;      1 OVM = 0        6   EALLOW = 0
;      0 SXM = 0        5     LOOP = 0
;                       4      SPA = 0
;                       3     VMAP = 1
;                       2    PAGE0 = 0
;                       1     DBGM = 1
;                       0     INTM = 1
;-----------------------------------------------
    MOV  *SP++,#0
    MOV  *SP++,#0x0A0B
    POP  ST1
    POP  ST0
;-----------------------------------------------
;   Jump to the EntryAddr as defined by the
;   boot mode selected and continue execution
;-----------------------------------------------
    LRETR

;eof ----------
```

```
// TI File $Revision: /main/2 $
// Checkin $Date: January 10, 2005   14:39:40 $
//###########################################################################
//
// FILE:    SelectMode_Boot.c
//
// TITLE:   280x Boot Mode selection routines
//
// Functions:
//
//      Uint32 SelectBootMode(void)
//      inline void SelectMode_GPOISelect(void)
//
// Notes:
//
//###########################################################################
// $TI Release:$
// $Release Date:$
//###########################################################################
#include "DSP280x_Device.h"
#include "280x_Boot.h"
extern Uint32 SCI_Boot(void);
extern Uint32 SPI_Boot(void);
extern Uint32 Parallel_Boot(void);
extern Uint32 I2C_Boot(void);
extern Uint32 CAN_Boot();
//         GPIO18     GPIO29    GPIO34
//         SPICLKA    SCITXDA
//         SCITXB
//Flash       1          1        1
//SCI         1          1        0
//SPI         1          0        1
//I2C         1          0        0
//ECAN        0          1        1
//RAM         0          1        0
//OTP         0          0        1
//I/0         0          0        0
#define FLASH_BOOT    7
#define SCI_BOOT      6
#define SPI_BOOT      5
#define I2C_BOOT      4
#define CAN_BOOT      3
#define RAM_BOOT      2
#define OTP_BOOT      1
#define PARALLEL_BOOT 0
Uint32 SelectBootMode()
{
  Uint32 EntryAddr;
  Uint16 BootMode;

  EALLOW;
  // Set MUX for BOOT Select
  GpioCtrlRegs.GPAMUX2.bit.GPIO18 = 0;
  GpioCtrlRegs.GPAMUX2.bit.GPIO29 = 0;
  GpioCtrlRegs.GPBMUX1.bit.GPIO34 = 0;
  // Set DIR for BOOT Select
  GpioCtrlRegs.GPADIR.bit.GPIO18 = 0;
  GpioCtrlRegs.GPADIR.bit.GPIO29 = 0;
  GpioCtrlRegs.GPBDIR.bit.GPIO34 = 0;
  EDIS;

  // Form BootMode from BOOT select pins
  BootMode  = GpioDataRegs.GPADAT.bit.GPIO18 << 2;
  BootMode |= GpioDataRegs.GPADAT.bit.GPIO29 << 1;
  BootMode |= GpioDataRegs.GPBDAT.bit.GPIO34;
  // Read the password locations - this will unlock the
```

```
    // CSM only if the passwords are erased.  Otherwise it
    // will not have an effect.
    CsmPwl.PSWD0;
    CsmPwl.PSWD1;
    CsmPwl.PSWD2;
    CsmPwl.PSWD3;
    CsmPwl.PSWD4;
    CsmPwl.PSWD5;
    CsmPwl.PSWD6;
    CsmPwl.PSWD7;

    // First check for modes which do not require
    // a boot loader (Flash/RAM/OTP)
    if(BootMode == FLASH_BOOT) return FLASH_ENTRY_POINT;
    if(BootMode == RAM_BOOT)   return RAM_ENTRY_POINT;
    if(BootMode == OTP_BOOT)   return OTP_ENTRY_POINT;

    // Otherwise, disable the watchdog and check for the
    // other boot modes that requre loaders
    EALLOW;
    SysCtrlRegs.WDCR = 0x0068;
    EDIS;
    if(BootMode == SCI_BOOT) EntryAddr = SCI_Boot();
    else if(BootMode == SPI_BOOT) EntryAddr = SPI_Boot();
    else if(BootMode == I2C_BOOT) EntryAddr = I2C_Boot();
    else if(BootMode == CAN_BOOT) EntryAddr = CAN_Boot();
    else if(BootMode == PARALLEL_BOOT) EntryAddr = Parallel_Boot();
    else return FLASH_ENTRY_POINT;
    EALLOW;
    SysCtrlRegs.WDCR = 0x0028; // Enable watchdog module
    SysCtrlRegs.WDKEY = 0x55;  // Clear the WD counter
    SysCtrlRegs.WDKEY = 0xAA;
    EDIS;
     return EntryAddr;
}
```

```
// TI File $Revision: /main/2 $
// Checkin $Date: January 10, 2005   14:39:44 $
//###########################################################################
//
// FILE:    SysCtrl_Boot.c
//
// TITLE:   F2810/12 Boot Rom System Control Routines
//
// Functions:
//
//     void WatchDogDisable(void)
//     void WatchDogEnable(void)
//
// Notes:
//
//###########################################################################
// $TI Release:$
// $Release Date:$
//###########################################################################
#include "DSP280x_Device.h"
//---------------------------------------------------------------
// This module disables the watchdog timer.
//---------------------------------------------------------------

void  WatchDogDisable()
{
   EALLOW;
   SysCtrlRegs.WDCR = 0x0068;               // Disable watchdog module
   EDIS;
}
//---------------------------------------------------------------
// This module enables the watchdog timer.
//---------------------------------------------------------------
void  WatchDogEnable()
{
   EALLOW;
   SysCtrlRegs.WDCR = 0x0028;               // Enable watchdog module
   SysCtrlRegs.WDKEY = 0x55;               // Clear the WD counter
   SysCtrlRegs.WDKEY = 0xAA;
   EDIS;
}
// EOF --------
```

```
// TI File $Revision: /main/2 $
// Checkin $Date: January 10, 2005   14:39:41 $
//###########################################################################
//
// FILE:    Shared_Boot.c
//
// TITLE:   280x Boot loader shared functions
//
// Functions:
//
//     void   CopyData(void)
//     Uint32 GetLongData(void)
//     void ReadReservedFn(void)
//
//###########################################################################
// $TI Release:$
// $Release Date:$
//###########################################################################
#include "DSP280x_Device.h"
#include "280x_Boot.h"
// GetWordData is a pointer to the function that interfaces to the peripheral.
// Each loader assigns this pointer to it's particular GetWordData function.
uint16fptr GetWordData;
// Function prototypes
Uint32 GetLongData();
void   CopyData(void);
void ReadReservedFn(void);
//#####################################################
// void CopyData(void)
//-----------------------------------------------------
// This routine copies multiple blocks of data from the host
// to the specified RAM locations.  There is no error
// checking on any of the destination addresses.
// That is it is assumed all addresses and block size
// values are correct.
//
// Multiple blocks of data are copied until a block
// size of 00 00 is encountered.
//
//-----------------------------------------------------

void CopyData()
{
   struct HEADER {
     Uint16 BlockSize;
     Uint32 DestAddr;
   } BlockHeader;

   Uint16 wordData;
   Uint16 I;

   // Get the size in words of the first block
   BlockHeader.BlockSize = (*GetWordData)();

   // While the block size is > 0 copy the data
   // to the DestAddr.  There is no error checking
   // as it is assumed the DestAddr is a valid
   // memory location

   while(BlockHeader.BlockSize != (Uint16)0x0000)
   {
      BlockHeader.DestAddr = GetLongData();
      for(I = 1; I <= BlockHeader.BlockSize; I++)
      {
          wordData = (*GetWordData)();
          *(Uint16 *)BlockHeader.DestAddr++ = wordData;
```

```
        }

        // Get the size of the next block
        BlockHeader.BlockSize = (*GetWordData)();
    }
    return;
}
//###################################################
// Uint32 GetLongData(void)
//---------------------------------------------------
// This routine fetches a 32-bit value from the peripheral
// input stream.
//---------------------------------------------------
Uint32 GetLongData()
{
    Uint32 longData;
    // Fetch the upper ? of the 32-bit value
    longData = ( (Uint32)(*GetWordData)() << 16);

    // Fetch the lower ? of the 32-bit value
    longData |= (Uint32)(*GetWordData)();
    return longData;
}
//###################################################
// void Read_ReservedFn(void)
//---------------------------------------------
// This function reads 8 reserved words in the header.
// None of these reserved words are used by the
// this boot loader at this time, they may be used in
// future devices for enhancements.  Loaders that use
// these words use their own read function.
//---------------------------------------------

void ReadReservedFn()
{
    Uint16 I;
    // Read and discard the 8 reserved words.
    for(I = 1; I <= 8; I++)
    {
       GetWordData();
    }
    return;
}
```

```
// TI File $Revision: /main/3 $
// Checkin $Date: January 10, 2005   15:57:54 $
//###########################################################################
//
// FILE:    SPI_Boot.c
//
// TITLE:   280x SPI Boot mode routines
//
// Functions:
//
//     Uint32 SPI_Boot(void)
//     inline void SPIA_Init(void)
//     inline void SPIA_Transmit(u16 cmdData)
//     inline void SPIA_ReservedFn(void);
//     Uint32 SPIA_GetWordData(void)
//
// Notes:
//
//###########################################################################
// $TI Release:$
// $Release Date:$
//###########################################################################
#include "DSP280x_Device.h"
#include "280x_Boot.h"
// Private functions
inline void SPIA_Init(void);
inline Uint16 SPIA_Transmit(Uint16 cmdData);
inline void SPIA_ReservedFn(void);
Uint16 SPIA_GetWordData(void);
// External functions
extern void CopyData(void);
Uint32 GetLongData(void);
//#################################################
// Uint32 SPI_Boot(void)
//-----------------------------------------
// This module is the main SPI boot routine.
// It will load code via the SPI-A port.
//
// It will return a entry point address back
// to the ExitBoot routine.
//-----------------------------------------

Uint32 SPI_Boot()
{
   Uint32 EntryAddr;
   // Assign GetWordData to the SPI-A version of the
   // function.  GetWordData is a pointer to a function.
   GetWordData = SPIA_GetWordData;
   // 1. Init SPI-A and set
   //    EEPROM chip enable - low
   SPIA_Init();
   // 2. Enable EEPROM and send EEPROM Read Command
   SPIA_Transmit(0x0300);
   // 3. Send Starting for the EEPROM address 16bit
   //    Sending 0x0000,0000 will work for address and data packets
   SPIA_GetWordData();
   // 4. Check for 0x08AA data header, else go to flash
   if(SPIA_GetWordData() != 0x08AA) return FLASH_ENTRY_POINT;
   // 5.Check for Clock speed change and reserved words
   SPIA_ReservedFn();
   // 6. Get point of entry address after load
   EntryAddr = GetLongData();
   // 7. Receive and copy one or more code sections to  destination addresses
   CopyData();
   // 8. Disable EEPROM chip enable - high
   //    Chip enable - high
```

```
   GpioDataRegs.GPASET.bit.GPIO19 = 1;
   return EntryAddr;
}
//###########################################################
// void SPIA_Init(void)
//-----------------------------------------------
// Initialize the SPI-A port for communications
// with the host.
//-----------------------------------------------

inline void SPIA_Init()
{
    // Enable SPI-A clocks
    EALLOW;
    SysCtrlRegs.PCLKCR0.bit.SPIAENCLK = 1;
    SysCtrlRegs.LOSPCP.all = 0x0002;
    // Enable FIFO reset bit only
    SpiaRegs.SPIFFTX.all=0x8000;
    // 8-bit character
    SpiaRegs.SPICCR.all = 0x0007;
    // Use internal SPICLK master mode and Talk mode
    SpiaRegs.SPICTL.all = 0x000E;
    // Use the slowest baud rate
    SpiaRegs.SPIBRR     = 0x007f;
    // Relinquish SPI-A from reset
    SpiaRegs.SPICCR.all = 0x0087;
    // Enable SPISIMO/SPISOMI/SPICLK pins
    // Enable pull-ups on SPISIMO/SPISOMI/SPICLK/SPISTE pins
    // GpioCtrlRegs.GPAPUD.bit.GPIO16 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO17 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO18 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO19 = 0;
    GpioCtrlRegs.GPAPUD.all &= 0xFFF0FFFF;
    // GpioCtrlRegs.GPAMUX2.bit.GPIO16 = 1;
    // GpioCtrlRegs.GPAMUX2.bit.GPIO17 = 1;
    // GpioCtrlRegs.GPAMUX2.bit.GPIO18 = 1;
    GpioCtrlRegs.GPAMUX2.all |= 0x00000015;
    // SPI-A pins are asynch
    // GpioCtrlRegs.GPAQSEL2.bit.GPIO16 = 3;
    // GpioCtrlRegs.GPAQSEL2.bit.GPIO17 = 3;
    // GpioCtrlRegs.GPAQSEL2.bit.GPIO18 = 3;
    GpioCtrlRegs.GPAQSEL2.all |= 0x0000003F;
    // IOPORT as output pin instead of SPISTE
    GpioCtrlRegs.GPAMUX2.bit.GPIO19 = 0;
    GpioCtrlRegs.GPADIR.bit.GPIO19 = 1;
    // Chip enable - low
    GpioDataRegs.GPACLEAR.bit.GPIO19 = 1;
    EDIS;
    return;
}
//###########################################################
// Uint16 SPIA_Transmit(Uint16 cmdData)
//-----------------------------------------------
// Send a byte/words through SPI transmit channel
//-----------------------------------------------

inline Uint16 SPIA_Transmit(Uint16 cmdData)
{
    Uint16 recvData;
    // Send Read command/dummy word to EEPROM to fetch a byte
    SpiaRegs.SPITXBUF = cmdData;
    while( (SpiaRegs.SPISTS.bit.INT_FLAG) !=1);
    // Clear SPIINT flag and capture received byte
    recvData = SpiaRegs.SPIRXBUF;
    return recvData;
}
```

```
//###################################################
// void SPIA_ReservedFn(void)
//---------------------------------------------------
// This function reads 8 reserved words in the header.
// The first word has parameters for LOSPCP
// and SPIBRR register 0xMSB:LSB, LSB = is a three
// bit field for LOSPCP change MSB = is a 6bit field
// for SPIBRR register update
//
// If either byte is the default value of the register
// then no speed change occurs.  The default values
// are LOSPCP = 0x02 and SPIBRR = 0x7F
// The remaining reserved words are read and discarded
// and then returns to the main routine.
//---------------------------------------------------

inline void SPIA_ReservedFn()
{
    Uint16 speedData;
    Uint16 I;

    // update LOSPCP register
    speedData = SPIA_Transmit((Uint16)0x0000);
    EALLOW;
    SysCtrlRegs.LOSPCP.all = speedData;
    EDIS;
    asm("    RPT #0x0F ||NOP");

    // update SPIBRR register
    speedData = SPIA_Transmit((Uint16)0x0000);
    SpiaRegs.SPIBRR  = speedData;
    asm("    RPT #0x0F ||NOP");
    // Read and discard the next 7 reserved words.
    for(I = 1; I <= 7; I++)
    {
       SPIA_GetWordData();
    }
    return;
}
//###################################################
// Uint16 SPIA_GetWordData(void)
//---------------------------------------------------
// This routine fetches two bytes from the SPI-A
// port and puts them together to form a single
// 16-bit value.  It is assumed that the host is
// sending the data in the form MSB:LSB.
//---------------------------------------------------
Uint16 SPIA_GetWordData()
{
   Uint16 wordData;
   // Fetch the LSB
   wordData =  SPIA_Transmit(0x0000);
   // Fetch the MSB
   wordData |= (SPIA_Transmit(0x0000) << 8);
   return wordData;
}
```

```
// TI File $Revision: /main/3 $
// Checkin $Date: January 10, 2005   15:06:37 $
//###########################################################################
//
// FILE:    SCI_Boot.c
//
// TITLE:   280x SCI Boot mode routines
//
// Functions:
//
//     Uint32 SCI_Boot(void)
//     inline void SCIA_Init(void)
//     inline void SCIA_AutobaudLock(void)
//     Uint32 SCIA_GetWordData(void)
//
// Notes:
//
//###########################################################################
// $TI Release:$
// $Release Date:$
//###########################################################################
#include "DSP280x_Device.h"
#include "280x_Boot.h"
// Private functions
inline void SCIA_Init(void);
inline void SCIA_AutobaudLock(void);
Uint16 SCIA_GetWordData(void);
// External functions
extern void CopyData(void);
Uint32 GetLongData(void);
extern void ReadReservedFn(void);
//#################################################
// Uint32 SCI_Boot(void)
//-----------------------------------------
// This module is the main SCI boot routine.
// It will load code via the SCI-A port.
//
// It will return a entry point address back
// to the InitBoot routine which in turn calls
// the ExitBoot routine.
//-----------------------------------------

Uint32 SCI_Boot()
{
   Uint32 EntryAddr;
   // Assign GetWordData to the SCI-A version of the
   // function.  GetWordData is a pointer to a function.
   GetWordData = SCIA_GetWordData;
   SCIA_Init();
   SCIA_AutobaudLock();
   // If the KeyValue was invalid, abort the load
   // and return the flash entry point.
   if (SCIA_GetWordData() != 0x08AA) return FLASH_ENTRY_POINT;
   ReadReservedFn();
   EntryAddr = GetLongData();
   CopyData();
   return EntryAddr;
}
//#################################################
// void SCIA_Init(void)
//-----------------------------------------------
// Initialize the SCI-A port for communications
// with the host.
//-----------------------------------------------

inline void SCIA_Init()
```

```
{
    // Enable the SCI-A clocks
    EALLOW;
    SysCtrlRegs.PCLKCR0.bit.SCIAENCLK=1;
    SysCtrlRegs.LOSPCP.all = 0x0002;
    SciaRegs.SCIFFTX.all=0x8000;
    // 1 stop bit, No parity, 8-bit character
    // No loopback
    SciaRegs.SCICCR.all = 0x0007;
    // Enable TX, RX, Use internal SCICLK
    SciaRegs.SCICTL1.all = 0x0003;
    // Disable RxErr, Sleep, TX Wake,
    // Disable Rx Interrupt, Tx Interrupt
    SciaRegs.SCICTL2.all = 0x0000;
    // Relinquish SCI-A from reset
    SciaRegs.SCICTL1.all = 0x0023;
    // Enable pull-ups on SCI-A pins
    // GpioCtrlRegs.GPAPUD.bit.GPIO28 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO29 = 0;
    GpioCtrlRegs.GPAPUD.all &= 0xCFFFFFFF;
    // Enable the SCI-A pins
    // GpioCtrlRegs.GPAMUX2.bit.GPIO28 = 1;
    // GpioCtrlRegs.GPAMUX2.bit.GPIO29 = 1;
    GpioCtrlRegs.GPAMUX2.all |= 0x05000000;
    // Input qual for SCI-A RX is asynch
    GpioCtrlRegs.GPAQSEL2.bit.GPIO28 = 3;
    EDIS;
    return;
}
//###########################################################
// void SCIA_AutobaudLock(void)
//-----------------------------------------------
// Perform autobaud lock with the host.
// Note that if autobaud never occurs
// the program will hang in this routine as there
// is no timeout mechanism included.
//-----------------------------------------------

inline void SCIA_AutobaudLock()
{
    Uint16 byteData;
    // Must prime baud register with >= 1
    SciaRegs.SCILBAUD = 1;
    // Prepare for autobaud detection
    // Set the CDC bit to enable autobaud detection
    // and clear the ABD bit
    SciaRegs.SCIFFCT.bit.CDC = 1;
    SciaRegs.SCIFFCT.bit.ABDCLR = 1;
    // Wait until you correctly read an
    // 'A' or 'a' and lock
    while(SciaRegs.SCIFFCT.bit.ABD != 1) {}
    // After autobaud lock, clear the CDC bit
    SciaRegs.SCIFFCT.bit.CDC = 0;
    while(SciaRegs.SCIRXST.bit.RXRDY != 1) { }
    byteData = SciaRegs.SCIRXBUF.bit.RXDT;
    SciaRegs.SCITXBUF = byteData;
    return;
}
//###########################################################
// Uint16 SCIA_GetWordData(void)
//-----------------------------------------------
// This routine fetches two bytes from the SCI-A
// port and puts them together to form a single
// 16-bit value.  It is assumed that the host is
// sending the data in the order LSB followed by MSB.
//-----------------------------------------------
```

```
Uint16 SCIA_GetWordData()
{
   Uint16 wordData;
   Uint16 byteData;

   wordData = 0x0000;
   byteData = 0x0000;

   // Fetch the LSB and verify back to the host
   while(SciaRegs.SCIRXST.bit.RXRDY != 1) { }
   wordData =  (Uint16)SciaRegs.SCIRXBUF.bit.RXDT;
   SciaRegs.SCITXBUF = wordData;
   // Fetch the MSB and verify back to the host
   while(SciaRegs.SCIRXST.bit.RXRDY != 1) { }
   byteData =  (Uint16)SciaRegs.SCIRXBUF.bit.RXDT;
   SciaRegs.SCITXBUF = byteData;

   // form the wordData from the MSB:LSB
   wordData |= (byteData << 8);
   return wordData;
}
// EOF-------
```

```
// TI File $Revision: /main/2 $
// Checkin $Date: January 10, 2005   14:39:37 $
//###########################################################################
//
// FILE:    Parallel_Boot.c
//
// TITLE:   280x Parallel Port I/O boot routines
//
// Functions:
//
//       Uint32 Parallel_Boot(void)
//       inline void Parallel_GPIOSelect(void)
//       inline Uint16 Parallel_CheckKeyVal(void)
//       Uint16 Parallel_GetWordData_8bit()
//       Uint16 Parallel_GetWordData_16bit()
//       void Parallel_WaitHostRdy(void)
//       void Parallel_HostHandshake(void)
// Notes:
//
//###########################################################################
// $TI Release:$
// $Release Date:$
//###########################################################################
#include "DSP280x_Device.h"
#include "280x_Boot.h"
// Private function definitions
inline void Parallel_GPIOSelect(void);
inline Uint16 Parallel_CheckKeyVal(void);
Uint16 Parallel_GetWordData_8bit(void);
Uint16 Parallel_GetWordData_16bit(void);
void Parallel_WaitHostRdy(void);
void Parallel_HostHandshake(void);
// External function definitions
extern void CopyData(void);
extern Uint32 GetLongData(void);
extern void ReadReservedFn(void);
#define HOST_CTRL          GPIO27  // GPIO27 is the host control signal
#define DSP_CTRL           GPIO26  // GPIO26 is the DSP's control signal
#define HOST_DATA_NOT_RDY  GpioDataRegs.GPADAT.bit.HOST_CTRL!=0
#define WAIT_HOST_ACK      GpioDataRegs.GPADAT.bit.HOST_CTRL!=1
// Set (DSP_ACK) or Clear (DSP_RDY) GPIO 17
#define DSP_ACK            GpioDataRegs.GPASET.bit.DSP_CTRL = 1;
#define DSP_RDY            GpioDataRegs.GPACLEAR.bit.DSP_CTRL = 1;
#define DATA               GpioDataRegs.GPADAT.all
//###################################################
// Uint32 Parallel_Boot(void)
//-------------------------------------------
// This module is the main Parallel boot routine.
// It will load code via GP I/O port B.
//
// This boot mode accepts 8-bit or 16-bit data.
// 8-bit data is expected to be the order LSB
// followed by MSB.
//
// This function returns a entry point address back
// to the InitBoot routine which in turn calls
// the ExitBoot routine.
//-------------------------------------------
Uint32 Parallel_Boot()
{
   Uint32 EntryAddr;
   // Setup for Parallel boot
   Parallel_GPIOSelect();
   // Check for the key value.  Based on this the data will
   // be read as 8-bit or 16-bit values.
   if (Parallel_CheckKeyVal() == ERROR) return FLASH_ENTRY_POINT;
```

```
    // Read and discard the reserved words
    ReadReservedFn();
    // Get the entry point address
    EntryAddr = GetLongData();
    // Load the data
    CopyData();

    return EntryAddr;
}
//###########################################################
// void Parallel_GPIOSelect(void)
//-----------------------------------------
// Enable I/O pins for input GPIO 15:0.  Also
// enable the control pins for HOST_CTRL and
// DSP_CTRL.
//-----------------------------------------
inline void Parallel_GPIOSelect()
{
    EALLOW;
    // Enable pull-ups for GPIO Port A 15:0
    // GPIO Port 15:0 are all I/O pins
    // and DSP_CTRL/HOST_CTRL
    // GpioCtrlRegs.GPAPUD.bit.GPIO15 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO14 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO13 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO12 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO11 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO10 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO9 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO8 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO7 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO6 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO5 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO4 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO3 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO2 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO1 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO0 = 0;
    // GpioCtrlRegs.GPAPUD.bit.DSP_CTRL = 0;
    // GpioCtrlRegs.GPAPUD.bit.HOST_CTRL = 0;
    GpioCtrlRegs.GPAPUD.all &= 0xF3FF0000;
    // 0 = I/O pin  1 = Peripheral pin
    GpioCtrlRegs.GPAMUX1.all = 0x0000;
    GpioCtrlRegs.GPAMUX2.bit.DSP_CTRL = 0;
    GpioCtrlRegs.GPAMUX2.bit.HOST_CTRL = 0;

    // HOST_CTRL is an input control
    // from the Host
    // to the DSP Ack/Rdy
    // DSP_CTRL is an output from the DSP Ack/Rdy
    // 0 = input   1 = output
    GpioCtrlRegs.GPADIR.bit.DSP_CTRL = 1;
    GpioCtrlRegs.GPADIR.bit.HOST_CTRL = 0;
    EDIS;
}
//###########################################################
// void Parallel_CheckKeyVal(void)
//-----------------------------------------
// Determine if the data you are loading is in
// 8-bit or 16-bit format.
// If neither, return an error.
//
// Note that if the host never responds then
// the code will be stuck here.  That is there
// is no timeout mechanism.
//-----------------------------------------
```

```
inline Uint16 Parallel_CheckKeyVal()
{
   Uint16 wordData;

   // Fetch a word from the parallel port and compare
   // it to the defined 16-bit header format, if not check
   // for a 8-bit header format.

   wordData = Parallel_GetWordData_16bit();
   if(wordData == SIXTEEN_BIT_HEADER)
   {
   // Assign GetWordData to the parallel 16bit version of the
   // function.  GetWordData is a pointer to a function.
      GetWordData = Parallel_GetWordData_16bit;
      return SIXTEEN_BIT;
   }
   // If not 16-bit mode, check for 8-bit mode
   // Call Parallel_GetWordData with 16-bit mode
   // so you only fetch the MSB of the KeyValue and not
   // two bytes.  You will ignore the upper 8-bits and combine
   // the result with the previous byte to form the
   // header KeyValue.

   wordData = wordData & 0x00FF;
   wordData |= Parallel_GetWordData_16bit() << 8;
   if(wordData == EIGHT_BIT_HEADER)
   {
   // Assign GetWordData to the parallel 8bit version of the
   // function.  GetWordData is a pointer to a function.
      GetWordData = Parallel_GetWordData_8bit;
      return EIGHT_BIT;
   }
   // Didn't find a 16-bit or an 8-bit KeyVal header so return an error.
   else return ERROR;
}
//################################################
// Uint16 Parallel_GetWordData_16bit()
// Uint16 Parallel_GetWordData_8bit()
//------------------------------------------------
// This routine fetches a 16-bit word from the
// GP I/O port.  The 16bit function is used if the
// input 16-bits and the function fetches a
// single word and returns it to the host.
//
// The _8bit function is used if the input stream is
// an 8-bit input stream and the upper 8-bits of the
// GP I/O port are ignored.  In the 8-bit case the
// first fetches the LSB and then the MSB from the
// GPIO port.  These two bytes are then put together to
// form a single 16-bit word that is then passed back
// to the host. Note that in this case, the input stream
// from the host is in the order LSB followed by MSB
//------------------------------------------------
Uint16 Parallel_GetWordData_8bit()
{
   Uint16 wordData;

   // Get LSB.

   Parallel_WaitHostRdy();
   wordData = DATA;
   Parallel_HostHandshake();

   // Fetch the MSB.

   wordData = wordData & 0x00FF;
```

```
   Parallel_WaitHostRdy();
   wordData |= (DATA << 8);
   Parallel_HostHandshake();
   return wordData;
}
Uint16 Parallel_GetWordData_16bit()
{
   Uint16 wordData;

   // Get a word of data.  If you are in
   // 16-bit mode then you are done.

   Parallel_WaitHostRdy();
   wordData = DATA;
   Parallel_HostHandshake();
   return wordData;
}
//###################################################
// void Parallel_WaitHostRdy(void)
//---------------------------------------------------
// This routine tells the host that the DSP is ready to
// receive data.  The DSP then waits for the host to
// signal that data is ready on the GP I/O port.e
//---------------------------------------------------
void Parallel_WaitHostRdy()
{
   DSP_RDY;
   while(HOST_DATA_NOT_RDY) { }
}
//###################################################
// void Parallel_HostHandshake(void)
//---------------------------------------------------
// This routine tells the host that the DSP has received
// the data.  The DSP then waits for the host to acknowledge
// the receipt before continuing.
//---------------------------------------------------
void Parallel_HostHandshake()
{
   DSP_ACK;
   while(WAIT_HOST_ACK) { }
}
// EOF --------
```

```
// TI File $Revision: /main/4 $
// Checkin $Date: January 10, 2005   15:57:47 $
//###########################################################################
//
// FILE:    I2C_Boot.c
//
// TITLE:   280x I2C Boot mode routines
//
// Functions:
//
//     Uint32 I2C_Boot(void)
//     inline void I2C_Init(void)
//     inline Uint16 I2C_CheckKeyVal(void)
//     inline void I2C_ReservedFn(void)
//     Uint16 I2C_GetWord(void)
//
// Notes:
//     The I2C code contained here is specifically streamlined for the F280x
//     bootloader. It can be used to load code via the I2C port into the
//     280x RAM and jump to an entry point within that code.
//
//     Features/Limitations:
//     - The I2C boot loader code is written to communicate with an EEPROM
//       device at address 0x50. The EEPROM must adhere to conventional I2C
//       EEPROM protocol (see the boot rom documentation) with a 16-bit
//       base address architecture (as opposed to 8-bits). The base address
//       of the code should be contained at address 0x0000 in the EEPROM.
//     - The input frequency to the F280x device must be between 14Mhz and
//       24Mhz, creating a 7Mhz to 12Mhz system clock. This is due to a
//       requirement that the I2C clock be between 7Mhz and 12Mhz to meet all
//       of the I2C specification timing requirements. The I2CPSC default value
//       is hardcoded to 0 so that the I2C clock will not be divided down from
//       the system clock. The I2CPSC value can be modified after receiving
//       the first few bytes from the EEPROM (see the boot rom documentation),
//       but it is advisable not to, as this can cause the I2C to operate out
//       of specification with a system clock between 7Mhz and 12Mhz.
//     - The bit period prescalers (I2CCLKH and I2CCLKL) are configured to
//       run the I2C at 50% duty cycle at 100kHz bit rate (standard I2C mode)
//       when the system clock is 12Mhz. These registers can be modified after
//       receiving the first few bytes from the EEPROM (see the boot rom
//       documentation). This allows the communication to be increased up to
//       a 400kHz bit rate (fast I2C mode) during the remaining data reads.
//     - Arbitration, bus busy, and slave signals are not checked. Therefore,
//       no other master is allowed to control the bus during this
//       initialization phase. If the application requires another master
//       during I2C boot mode, that master must be configured to hold off
//       sending any I2C messages until the F280x application software
//       signals that it is past the bootloader portion of initialization.
//     - The non-acknowledgement bit is only checked during the first message
//       sent to initialize the EEPROM base address. This ensures that an
//       EEPROM is present at address 0x50 before continuing on. If an EEPROM
//       is not present, code will jump to the Flash entry point. The
//       non-acknowledgement bit is not checked during the address phase of
//       the data read messages (I2C_GetWord). If a non-acknowledge is
//       received during the data read messages, the I2C bus will hang.
//
//###########################################################################
// $TI Release:$
// $Release Date:$
//###########################################################################
#include "DSP280x_Device.h"     // DSP280x Headerfile Include File
#include "280x_Boot.h"
// Private functions
inline void   I2C_Init(void);
inline Uint16 I2C_CheckKeyVal(void);
inline void   I2C_ReservedFn(void);
```

```
       Uint16 I2C_GetWord(void);
// External functions
extern void   CopyData(void);
extern Uint32 GetLongData(void);
//#################################################
// Uint32 I2C_Boot(void)
//-----------------------------------------
// This module is the main I2C boot routine.
// It will load code via the I2C-A port.
//
// It will return an entry point address back
// to the ExitBoot routine.
//-----------------------------------------

Uint32 I2C_Boot(void)
{
   Uint32 EntryAddr;
   // Assign GetWordData to the I2C-A version of the
   // function.  GetWordData is a pointer to a function.
   GetWordData = I2C_GetWord;
   // Init I2C pins, clock, and registers
   I2C_Init();

   // Check for 0x08AA data header, else go to flash
   if (I2C_CheckKeyVal() == ERROR) { return FLASH_ENTRY_POINT; }

   // Check for clock and prescaler speed changes and reserved words
   I2C_ReservedFn();

   // Get point of entry address after load
   EntryAddr = GetLongData();

   // Receive and copy one or more code sections to  destination addresses
   CopyData();
   return EntryAddr;
}
//#################################################
// void I2C_Init(void)
//-----------------------------------------------
// Initialize the I2C-A port for communications
// with the host.
//-----------------------------------------------
inline void I2C_Init(void)
{
   // Configure I2C pins and turn on I2C clock
   EALLOW;
   GpioCtrlRegs.GPBMUX1.bit.GPIO32 = 1;        // Configure as SDA pin
   GpioCtrlRegs.GPBMUX1.bit.GPIO33 = 1;        // Configure as SCL pin
   GpioCtrlRegs.GPBPUD.bit.GPIO32 = 0;        // Turn SDA pullup on
   GpioCtrlRegs.GPBPUD.bit.GPIO33 = 0;        // Turn SCL pullup on
   GpioCtrlRegs.GPBQSEL1.bit.GPIO32 = 3;    // Asynch
   GpioCtrlRegs.GPBQSEL1.bit.GPIO33 = 3;    // Asynch
   SysCtrlRegs.PCLKCR0.bit.I2CAENCLK = 1;    // Turn I2C module clock on
   EDIS;

   // Initialize I2C in master transmitter mode
   I2caRegs.I2CSAR = 0x0050;        // Slave address - EEPROM control code
   I2caRegs.I2CPSC.all = 0x0;        // I2C clock should be between 7Mhz-12Mhz
   I2caRegs.I2CCLKL = 0x0035;        // Prescalers set for 100kHz bit rate
   I2caRegs.I2CCLKH = 0x0035;        //   at a 12Mhz I2C clock
   I2caRegs.I2CMDR.all = 0x0620;    // Master transmitter
                                      // Take I2C out of reset
                                      // Stop when suspended
   I2caRegs.I2CFFTX.all = 0x6000;    // Enable FIFO mode and TXFIFO
   I2caRegs.I2CFFRX.all = 0x2000;    // Enable RXFIFO
```

```
   return;
}
//###############################################
// Uint16 I2C_CheckKeyVal(void)
//-----------------------------------------------
// This routine sets up the starting address in the
// EEPROM by writing two bytes (0x0000) via the
// I2C-A port to slave address 0x50. Without
// sending a stop bit, the communication is then
// restarted and two bytes are read from the EEPROM.
// If these two bytes read do not equal 0x08AA
// (little endian), an error is returned.
//-----------------------------------------------
inline Uint16 I2C_CheckKeyVal(void)
{
   // To read a word from the EEPROM, an address must be given first in
   // master transmitter mode. Then a restart is performed and data can
   // be read back in master receiver mode.
   I2caRegs.I2CCNT = 0x02;                // Setup how many bytes to send
   I2caRegs.I2CDXR = 0x00;                // Configure fifo data for byte
   I2caRegs.I2CDXR = 0x00;                //   address of 0x0000
   I2caRegs.I2CMDR.all = 0x2620;        // Send data to setup EEPROM address
   while (I2caRegs.I2CSTR.bit.ARDY == 0) // Wait until communication
   {                                     //   complete and registers ready
   }

   if (I2caRegs.I2CSTR.bit.NACK == 1)    // Set stop bit & return error if
   {                                    //    NACK received
      I2caRegs.I2CMDR.bit.STP = 1;
      return ERROR;
   }
   // Check to make sure key value received is correct
   if (I2C_GetWord() != 0x08AA) {return ERROR;}

   return NO_ERROR;
}
//###############################################
// void I2C_ReservedFn(void)
//-----------------------------------------------
// This function reads 8 reserved words in the header.
//   1st word - parameters for I2CPSC register
//   2nd word - parameters for I2CCLKH register
//   3rd word - parameters for I2CCLKL register
//
// The remaining reserved words are read and discarded
// and then program execution returns to the main routine.
//-----------------------------------------------
inline void I2C_ReservedFn(void)
{
   Uint16 I2CPrescaler;
   Uint16 I2cClkHData;
   Uint16 I2cClkLData;
   Uint16 I;

   // Get I2CPSC, I2CCLKH, and I2CCLKL values
   I2CPrescaler = I2C_GetWord();
   I2cClkHData = I2C_GetWord();
   I2cClkLData = I2C_GetWord();
   // Store I2C clock prescalers
   I2caRegs.I2CMDR.bit.IRS = 0;
   I2caRegs.I2CCLKL = I2cClkLData;
   I2caRegs.I2CCLKH = I2cClkHData;
   I2caRegs.I2CPSC.all = I2CPrescaler;
   I2caRegs.I2CMDR.bit.IRS = 1;

   // Read and discard the next 5 reserved words
```

```
   for (I=1; I<=5; I++)
   {
      I2cClkHData = I2C_GetWord();
   }

   return;
}
//###################################################
// Uint16 I2C_GetWord(void)
//-------------------------------------------------
// This routine fetches two bytes from the I2C-A
// port and puts them together little endian style
// to form a single 16-bit value.
//-------------------------------------------------
Uint16 I2C_GetWord(void)
{
   Uint16 LowByte;

   I2caRegs.I2CCNT = 2;                    // Setup how many bytes to expect
   I2caRegs.I2CMDR.all = 0x2C20;        // Send start as master receiver
   // Wait until communication done
   while (I2caRegs.I2CMDR.bit.STP == 1) {}

   // Combine two bytes to one word & return
   LowByte = I2caRegs.I2CDRR;
   return (LowByte | (I2caRegs.I2CDRR<<8));
}
//===========================================================================
// No more.
//===========================================================================
```

```
// TI File $Revision: /main/7 $
// Checkin $Date: January 20, 2005   10:05:26 $
//###########################################################################
//
// FILE:    CAN_Boot.c
//
// TITLE:   280x CAN Boot mode routines
//
// Functions:
//
//     Uint32 CAN_Boot(void)
//     void CAN_Init(void)
//     Uint32 CAN_GetWordData(void)
//
// Notes:
// BRP = 2, Bit time = 10. This would yield the following bit rates with the
// default PLL setting:
// XCLKIN = 40 MHz    SYSCLKOUT = 20 MHz    Bit rate = 1 Mbits/s
// XCLKIN = 20 MHz    SYSCLKOUT = 10 MHz    Bit rate = 500 kbits/s
// XCLKIN = 10 MHz    SYSCLKOUT =  5 MHz    Bit rate = 250 kbits/s
// XCLKIN =  5 MHz    SYSCLKOUT = 2.5MHz    Bit rate = 125 kbits/s
//###########################################################################
// $TI Release:$
// $Release Date:$
//###########################################################################
#include "DSP280x_Device.h"
#include "280x_Boot.h"
// Private functions
void CAN_Init(void);
Uint16 CAN_GetWordData(void);
// External functions
extern void CopyData(void);
extern Uint32 GetLongData(void);
extern void ReadReservedFn(void);
//#################################################
// Uint32 CAN_Boot(void)
//-----------------------------------------
// This module is the main CAN boot routine.
// It will load code via the CAN-A port.
//
// It will return a entry point address back
// to the InitBoot routine which in turn calls
// the ExitBoot routine.
//-----------------------------------------

Uint32 CAN_Boot()
{
   Uint32 EntryAddr;

   // If the missing clock detect bit is set, just
   // loop here.
   if(SysCtrlRegs.PLLSTS.bit.MCLKSTS == 1)
   {
      for(;;);
   }
   // Assign GetWordData to the CAN-A version of the
   // function.  GetWordData is a pointer to a function.
   GetWordData = CAN_GetWordData;
   CAN_Init();

   // If the KeyValue was invalid, abort the load
   // and return the flash entry point.
   if (CAN_GetWordData() != 0x08AA) return FLASH_ENTRY_POINT;
   ReadReservedFn();
   EntryAddr = GetLongData();
   CopyData();
```

```
   return EntryAddr;
}
//###########################################################
// void CAN_Init(void)
//-----------------------------------------------
// Initialize the CAN-A port for communications
// with the host.
//-----------------------------------------------

void CAN_Init()
{
/* Create a shadow register structure for the CAN control registers. This is
 needed, since, only 32-bit access is allowed to these registers. 16-bit access
 to these registers could potentially corrupt the register contents. This is
 especially true while writing to a bit (or group of bits) among bits 16 - 31 */

   struct ECAN_REGS ECanaShadow;
   EALLOW;
/* Enable CAN clock  */
   SysCtrlRegs.PCLKCR0.bit.ECANAENCLK=1;

/* Configure eCAN-A pins using GPIO regs*/
   GpioCtrlRegs.GPAMUX2.bit.GPIO30 = 1; // GPIO30 is CANRXA
   GpioCtrlRegs.GPAMUX2.bit.GPIO31 = 1; // GPIO31 is CANTXA

/* Configure eCAN RX and TX pins for eCAN transmissions using eCAN regs*/
   ECanaRegs.CANTIOC.bit.TXFUNC = 1;
   ECanaRegs.CANRIOC.bit.RXFUNC = 1;

/* Enable internal pullups for the CAN pins  */
   GpioCtrlRegs.GPAPUD.bit.GPIO30 = 0;
   GpioCtrlRegs.GPAPUD.bit.GPIO31 = 0;

/* Asynch Qual */
   GpioCtrlRegs.GPAQSEL2.bit.GPIO30 = 3;

/* Initialize all bits of 'Master Control Field' to zero */
// Some bits of MSGCTRL register come up in an unknown state. For proper operation,
// all bits (including reserved bits) of MSGCTRL must be initialized to zero
    ECanaMboxes.MBOX1.MSGCTRL.all = 0x00000000;
// RMPn, GIFn bits are all zero upon reset and are cleared again
//    as a matter of precaution.
/* Clear all RMPn bits */

   ECanaRegs.CANRMP.all = 0xFFFFFFFF;

/* Clear all interrupt flag bits */

   ECanaRegs.CANGIF0.all = 0xFFFFFFFF;
   ECanaRegs.CANGIF1.all = 0xFFFFFFFF;

/* Configure bit timing parameters for eCANA*/
   ECanaShadow.CANMC.all = ECanaRegs.CANMC.all;
   ECanaShadow.CANMC.bit.CCR = 1 ;            // Set CCR = 1
   ECanaRegs.CANMC.all = ECanaShadow.CANMC.all;
   while(ECanaRegs.CANES.bit.CCE != 1 ) {}   // Wait for CCE bit to be set..
   ECanaShadow.CANBTC.all = 0;
   ECanaShadow.CANBTC.bit.BRPREG = 1;
   ECanaShadow.CANBTC.bit.TSEG2REG = 2;
   ECanaShadow.CANBTC.bit.TSEG1REG = 5;
   ECanaShadow.CANBTC.bit.SAM = 1;
   ECanaRegs.CANBTC.all = ECanaShadow.CANBTC.all;
   ECanaShadow.CANMC.all = ECanaRegs.CANMC.all;
   ECanaShadow.CANMC.bit.CCR = 0 ;            // Set CCR = 0
   ECanaRegs.CANMC.all = ECanaShadow.CANMC.all;
   while(ECanaRegs.CANES.bit.CCE == !0 ) {}   // Wait for CCE bit to be cleared..
```

```
/* Disable all Mailboxes  */

   ECanaRegs.CANME.all = 0;        // Required before writing the MSGIDs

/* Assign MSGID to MBOX1 */
   ECanaMboxes.MBOX1.MSGID.all = 0x00040000;
/* Configure MBOX1 to be a receive MBOX */
   ECanaRegs.CANMD.all = 0x0002;

/* Enable MBOX1 */
   ECanaRegs.CANME.all = 0x0002;
   EDIS;

    return;
}
//###############################################
// Uint16 CAN_GetWordData(void)
//-----------------------------------------------
// This routine fetches two bytes from the CAN-A
// port and puts them together to form a single
// 16-bit value.  It is assumed that the host is
// sending the data in the order LSB followed by MSB.
//-----------------------------------------------
Uint16 CAN_GetWordData()
{
   Uint16 wordData;
   Uint16 byteData;

   wordData = 0x0000;
   byteData = 0x0000;

// Fetch the LSB
   while(ECanaRegs.CANRMP.all == 0) { }
   wordData =  (Uint16) ECanaMboxes.MBOX1.MDL.byte.BYTE0;    // LS byte

   // Fetch the MSB

   byteData =  (Uint16)ECanaMboxes.MBOX1.MDL.byte.BYTE1;    // MS byte

   // form the wordData from the MSB:LSB
   wordData |= (byteData << 8);
/* Clear all RMPn bits */

    ECanaRegs.CANRMP.all = 0xFFFFFFFF;

   return wordData;
}
/*
Data frames with a Standard MSGID of 0x1 should be transmitted to the ECAN-A bootloader.
This data will be received in Mailbox1, whose MSGID is 0x1. No message filtering is employed.
Transmit only 2 bytes at a time, LSB first and MSB next. For example, to transmit
the word 0x08AA to the 280x, transmit AA first, followed by 08. Following is the
order in which data should be transmitted:
AA 08    -    Keyvalue
00 00    -    Part of 8 reserved words stream
00 00    -    Part of 8 reserved words stream
00 00    -    Part of 8 reserved words stream
00 00    -    Part of 8 reserved words stream
00 00    -    Part of 8 reserved words stream
00 00    -    Part of 8 reserved words stream
00 00    -    Part of 8 reserved words stream
00 00    -    Part of 8 reserved words stream
bb aa    -    MS part of 32-bit address (aabb)
dd cc    -    LS part of 32-bit address (ccdd) - Final Entry-point address = 0xaabbccdd
nn mm    -    Length of first section (mm nn)
```

```
ff ee    -    MS part of 32-bit address (eeff)
hh gg    -    LS part of 32-bit address (gghh) - Entry-point address of first section = 0xeeffgghh
xx xx    -   First word of first section
xx xx    -    Second word......
...
...
...
xxx        -     Last word of first section
nn mm    -    Length of second section (mm nn)
ff ee    -    MS part of 32-bit address (eeff)
hh gg    -    LS part of 32-bit address (gghh) - Entry-point address of second section = 0xeeffgghh
xx xx    -   First word of second section
xx xx    -    Second word......
...
...
...
xxx        -     Last word of second section
(more sections, if need be)
00 00    -     Section length of zero for next section indicates end of data.
*/
/*
Notes:
------
Summary of changes in ver 2.0, as compared to 1.0
1. Changed the statement
         ECanaMboxes.MBOX0.MSGCTRL.all = 0x00000000;
   to
   ECanaMboxes.MBOX1.MSGCTRL.all = 0x00000000;
   since it is MBOX1 that is used, not MBOX0.

2. Made BRP = 1. BRP was 0 in rev 1.0 . BT is now 10 to maintain
   the SYSCLKOUT-bitrate relationship.

3. Changed the statement
         ECanaMboxes.MBOX1.MSGID.bit.STDMSGID = 1;
   to
         ECanaMboxes.MBOX1.MSGID.all = 0x00040000;
   since IDE,AME bits are not initialized in the previous version.
4. Employed Shadow writes to CANBTC register

*/
// EOF-------
```

```
/*
// TI File $Revision: /main/5 $
// Checkin $Date: April 21, 2005   15:59:42 $
//###########################################################################
//
// FILE:    F280x_boot_rom_lnk.cmd
//
// TITLE:   F280x boot rom linker command file
//
//
//###########################################################################
// $TI Release:$
// $Release Date:$
//###########################################################################
*/
MEMORY
{
PAGE 0 :
         TABLES    : origin = 0x3FF000, length = 0x000b50
         BOOT      : origin = 0x3FFB50, length = 0x000386
         RSVD1     : origin = 0x3FFED6, length = 0x0000E3
         FLASH_API : origin = 0x3FFFB9, length = 0x000001
         VERSION   : origin = 0x3FFFBA, length = 0x000002
         CHECKSUM  : origin = 0x3FFFBC, length = 0x000004
         VECS      : origin = 0x3FFFC0, length = 0x000040
PAGE 1 :
         EBSS      : origin = 0x400, length = 0x002
         STACK     : origin = 0x402, length = 0x200
}

SECTIONS
{
         IQmathTables : load = TABLES,  PAGE = 0
         .InitBoot    : load = BOOT,    PAGE = 0
         .text        : load = BOOT,    PAGE = 0
         .BootVecs    : load = VECS,    PAGE = 0
         .Checksum    : load = CHECKSUM, PAGE = 0
         .Version     : load = VERSION, PAGE = 0
         .stack       : load = STACK,   PAGE = 1
         .ebss        : load = EBSS,    PAGE = 1
         rsvd1        : load = RSVD1,   PAGE = 0
}
```

# 修订历史记录

本文档从 SPRU722A 修订为 SPRU722B。 附录部分仅列出在最新版本中所做的修订。 修订范围限定于如表 A-1。

表 A-1. 版本 B 的更改

| 位置 | 添加、删除、修改 |
|---|---|
| 第 2.7 部分 | 更正了保留存储器的地址。 |
| 第 2.12 部分 | 将 eCAN-A 引导下面的第二段更改为一个注意事项并增加了一句话 |
| 第 4.2 部分 | 添加了 SelectBootMode 函数的代码列表。 |

# 重要声明

德州仪器 (TI) 及其下属子公司有权在不事先通知的情况下，随时对所提供的产品和服务进行更正、修改、增强、改进或其它更改，并有权随时中止提供任何产品和服务。 客户在下订单前应获取最新的相关信息，并验证这些信息是否完整且是最新的。 所有产品的销售都遵循在订单确认时所提供的 TI 销售条款与条件。

TI 保证其所销售的硬件产品的性能符合 TI 标准保修的适用规范。 仅在 TI 保修的范围内，且 TI 认为有必要时才会使用测试或其它质量控制技术。 除非政府做出了硬性规定，否则没有必要对每种产品的所有参数进行测试。

TI 对应用帮助或客户产品设计不承担任何义务。 客户应对其使用 TI 组件的产品和应用自行负责。 为尽量减小与客户产品和应用相关的风险，客户应提供充分的设计与操作安全措施。

TI 不对任何 TI 专利权、版权、屏蔽作品权或其它与使用了 TI 产品或服务的组合设备、机器、流程相关的 TI 知识产权中授予的直接或隐含权限作出任何保证或解释。 TI 所发布的与第三方产品或服务有关的信息，不能构成从 TI 获得使用这些产品或服务的许可、授权、或认可。 使用此类信息可能需要获得第三方的专利权或其它知识产权方面的许可，或是 TI 的专利权或其它知识产权方面的许可。

对于 TI 的数据手册或数据表，仅在没有对内容进行任何篡改且带有相关授权、条件、限制和声明的情况下才允许进行复制。 在复制信息的过程中对内容的篡改属于非法的、欺诈性商业行为。 TI 对此类篡改过的文件不承担任何责任。

在转售 TI 产品或服务时，如果存在对产品或服务参数的虚假陈述，则会失去相关 TI 产品或服务的明示或暗示授权，且这是非法的、欺诈性商业行为。 TI 对此类虚假陈述不承担任何责任。

可访问以下 URL 地址以获取有关其它 TI 产品和应用解决方案的信息：

| | |
|---|---|
| **产品** | |
| 放大器 | http://www.ti.com.cn/amplifiers |
| 数据转换器 | http://www.ti.com.cn/dataconverters |
| DSP | http://www.ti.com.cn/dsp |
| 接口 | http://www.ti.com.cn/interface |
| 逻辑 | http://www.ti.com.cn/logic |
| 电源管理 | http://www.ti.com.cn/power |
| 微控制器 | http://www.ti.com.cn/microcontrollers |
| **应用** | |
| 音频 | http://www.ti.com.cn/audio |
| 汽车 | http://www.ti.com.cn/automotive |
| 宽带 | http://www.ti.com.cn/broadband |
| 数字控制 | http://www.ti.com.cn/control |
| 光纤网络 | http://www.ti.com.cn/opticalnetwork |
| 安全 | http://www.ti.com.cn/security |
| 电话 | http://www.ti.com.cn/telecom |
| 视频与成像 | http://www.ti.com.cn/video |
| 无线 | http://www.ti.com.cn/wireless |

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265