*Application Note*

# *使用嵌入式控制器 (EC) 将补丁捆绑包直接载入 TPS25751 或 TPS26750 中*

*Chris Sterzik, Roy Chou*

**摘要**

TPS25751 和 TPS26750 是可配置的 USB Type-C® 和电力输送 (PD) 控制器，针对支持电源的应用进行了优化。通常，这些 PD 控制器在引导期间使用 I2Cc 从外部 EEPROM 加载二进制映像。如果有嵌入式控制器 (EC)，则可以使用 EC 代替 EEPROM，从而通过 I2Ct 接口将映像加载到 PD。技术参考手册 ( 1 和 2 ) 演示了通过 I2Ct 总线将映像同时写入多个 PD 控制器的流程。本应用手册更详细地描述了此流程，并重点介绍了将映像写入一个 PD 控制器的情况。随示例代码一起提供了通过 I2Ct 使用 I2C 命令从 PTCH 模式切换到 APP 模式的分步说明。

## 内容



## 商标

# 1 简介

在某些应用中，需要嵌入式控制器 (EC) 在应用运行期间（APP 模式）与 PD 控制器交互，或者系统中可能有一个 EC 负责执行日常活动。为了降低成本，可以移除 EEPROM，并将通过二进制映像存储和更新 PD 的功能移至 EC。本应用手册重点介绍如何使用 EC 来加载映像，并使用四字符代码 (4CC) 命令将 PD 控制器从 PTCH 模式移至 APP 模式。在讨论 EC 之前，将简要介绍一下无 EEPROM 的应用环境中的无电电池配置。

# 2 ADCINX 设置

ADCIN1 输入引脚决定 PD 控制器目标地址，而 ADCIN2 输入引脚定义无电电池配置。完整说明请参阅器件数据表；见 3 和 4。无电电池一词源自 PC 笔记本电脑应用场景，当电池无电时，PD 控制器需要在不进行任何交互（或无电源）的情况下启用电源路径，以从 VBUS 为无电电池充电。PD 控制器提供两种不同的无电电池配置[1]：AlwaysEnableSink 和 SafeMode。SafeMode 提供 USB Type-C 灌电流功能，但不会启用灌电流路径。在 SafeMode 下，将实际禁用 PD 状态机，直到二进制映像加载。AlwaysEnableSink 在建立 USB Type-C 连接（端口伙伴必须为供电端）后启用电源路径，并且 USB PD 功能在配置加载前保持禁用。

本应用手册中描述的步骤与无电电池配置无关。EC 对无电电池标志的处理和对系统的运行不属于本文档介绍范畴。

[1] 本讨论中省略了 NegotiateHighVoltage。

# 3 唯一地址接口协议

补丁突发模式 (PBM) 功能同时使用 SMBUS 协议和简单的 I2C 写入。数据表中对 SMBUS 协议进行了介绍（见 3 和 4），适用于所有寄存器访问。SMBUS 寄存器写入示例 和 SMBUS 寄存器读取示例 中列出了 PBM 中使用的寄存器写入和读取示例。在使用 SMBUS 协议发出 PBMs 命令后，I2C 写入操作将指向 PBMs 命令中建立的目标地址。I2C 补丁突发模式写入 展示了用于将映像发送到 PD 控制器的简单 I2C 写入。

**表 3-1. SMBUS 寄存器写入示例**

| 类型 | ACK | 地址 | 读取 | DATA | 说明 |
|---|---|---|---|---|---|
| start | | | | | |
| address | 真 | 0x21 | 否 | | 用于访问寄存器的 PD I2C 地址 |
| 指令 | 真 | | | 0x08 | 寄存器地址 |
| 指令 | 真 | | | 0x04 | （发送到目标的）字节数 |
| 指令 | 真 | | | 0x50 | P |
| 指令 | 真 | | | 0x42 | B |
| 指令 | 真 | | | 0x4D | M |
| 指令 | 真 | | | 0x73 | S |
| stop | | | | | |

**表 3-2. SMBUS 寄存器读取示例**

| 类型 | ACK | 地址 | 读取 | DATA | 说明 |
|---|---|---|---|---|---|
| start | | | | | |
| address | 真 | 0x21 | 否 | | 用于访问寄存器的 PD I2C 地址 |
| 指令 | 真 | | | 0x09 | 寄存器编号 |
| start | | | | | 重复启动以从写入更改为读取 |
| address | 真 | 0x21 | 真 | | 用于访问寄存器的 PD I2C 地址 |
| 指令 | 真 | | | 0x40 | 字节数 [2] |
| 指令 | 真 | | | 0x00 | |
| 指令 | 真 | | | 0x00 | |
| 指令 | 真 | | | 0x00 | |
| 指令 | 真 | | | 0x00 | |
| | 真 | | | 0x30 | PBM 地址 |
| | 否 | | | 0x31 | 超时。控制器对要读取的最后一个字节发出否定应答。 |
| stop | | | | | |

[2] 控制器可以选择读取所有字节或仅读取部分字节。在此示例中，控制器仅收到 6 个字节，对最后一个字节发出了否定应答。在 DATA1 寄存器 0x09 的最终读取操作中，读取了所有 0x40 个字节。

表 3-3. I2C 补丁突发模式写入

| 类型 | ACK | 地址 | 读取 | DATA | 说明 |
|---|---|---|---|---|---|
| start | | | | | |
| address | 真 | 0x30 | 否 | | PBM I2C 地址 |
| 指令 | 真 | | | 0x01 | 映像字节 0 |
| 指令 | 真 | | | 0x00 | 映像字节 1 |
| 指令 | 真 | | | 0xE0 | 映像字节 2 |
| 指令 | 真 | | | 0xAC | 映像字节 3 |
| 字节 4 至 4,093 | | | | | |
| 指令 | 真 | | | | 映像字节 4,094 |
| stop[3] | | | | | |
| start | | | | | |
| address | 真 | 0x30 | 否 | | PBM I2C 地址 |
| 指令 | | | | | 映像字节 4,095 |
| 字节 4,096 至 11,390 | | | | | |
| 指令 | 真 | | | 0x00 | 映像字节 11,391 |
| stop | | | | | |

[3] EC 的传输大小限制为 4,095 字节。PD 控制器会自动递增 PBM 地址指针，并且不会通过 I2C 启动或停止进行复位。可以发出 PBM 命令来复位 PBMs 指针。

# 4 从 PTCH 模式切换至 APP 模式

补丁捆绑包是一种二进制映像，其中包括（捆绑了）器件配置和固件更新（补丁）。节 4.2 展示了如何生成补丁捆绑包，也称为低区二进制文件。节 4.1 逐步介绍了补丁突发模式 (PBM) 流程以及从 *PTCH* 到 *APP* 模式的切换结果。PD 控制器的 TRM 中对 PBM 进行了总体介绍，本示例基本符合该介绍，不同的是使用了中断。这些步骤包括如何在 PBM 流程中设置和维护中断。

通过 I2Ct 总线推送补丁捆绑包 中的流程图假设不存在 EEPROM，以 PD 控制器冷启动或硬复位作为起点。

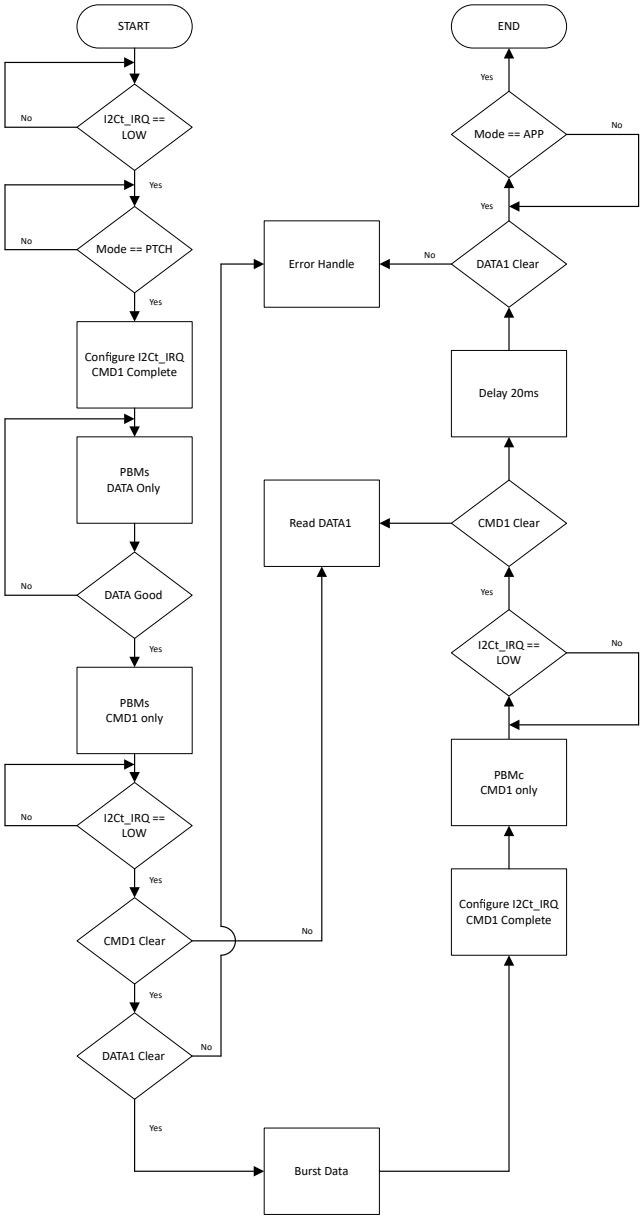


**图 4-1. 通过 I2Ct 总线推送补丁捆绑包**

## 4.1 从 PTCH 模式切换至 APP 模式的步骤

1. **I2Ct_IRQ == 低电平：**

   冷启动（断电重启或 GAID）后，PD 控制器进入 *PTCH* 模式，且仅*修补准备就绪[81]* 中断会自动启用。可以在 *PTCH* 模式下更新中断寄存器。此 PBM 实现方案使用*修补准备就绪*和 CMD1 完成中断。之所以采用中断而非轮询寄存器，主要原因是为了将 PD 控制器 CPU 负载减少至仅与命令相关的活动。

   在 PBM 流程开始时，使用*修补准备就绪*中断指示 PD 控制器已准备就绪。CMD1 完成中断用于提醒 EC：PBMs 和 PBMc 命令已完成。此示例中仍包含对 MODE 寄存器 0x03 的轮询，以应对补丁已加载但 PD 控制器尚未切换至应用模式：*APP* 的情况。

2. **模式 == PTCH：**

   PD 控制器 TRM 中介绍了 *PTCH* 和 *APP* 模式。PD 控制器 EVM 上的 EEPROM 被禁用（SDA 断开），因此 PD 控制器切换至 *PTCH* 模式并保持该模式。在流程开始时不是必须检查 *PTCH* 模式，但为了完整起见，仍包含了这一步。下面给出了命令示例，逻辑分析仪捕获结果如读取 *PTCH* 模式所示。

   [0x21] + ACK（唯一地址/WR/A）

   0x03 + ACK（寄存器编号/A）

   [0x21] + ACK（唯一地址/R/A）

   0x04（字节计数）

   0x50 0x54 0x43 0x48（以 4 个 ASCII 字符表示的 *PTCH*）


**图 4-2. 读取 *PTCH* 模式**

3. **配置 I2Ct_IRQ、CMD1 完成：**

   CMD1 完成中断用于提醒 EC：PBMs 命令已完成。分别通过寄存器 0x16 和 0x18 设置中断屏蔽和清除中断。请参阅 1 和 2

   [0x21] + ACK（唯一地址/WR/A）

   0x16 + ACK（寄存器编号/A）

   0x0B（字节计数）

   0x00 0x00 0x00 0x40 0x00 0x00 0x00 0x00 0x00 0x00 0x01 (MSB)


**图 4-3. 配置中断屏蔽寄存器 0x16**

   [0x21] + ACK（唯一地址/WR/A）

   0x18 + ACK（寄存器编号/A）

   0x0B（字节计数）

   0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF (MSB)

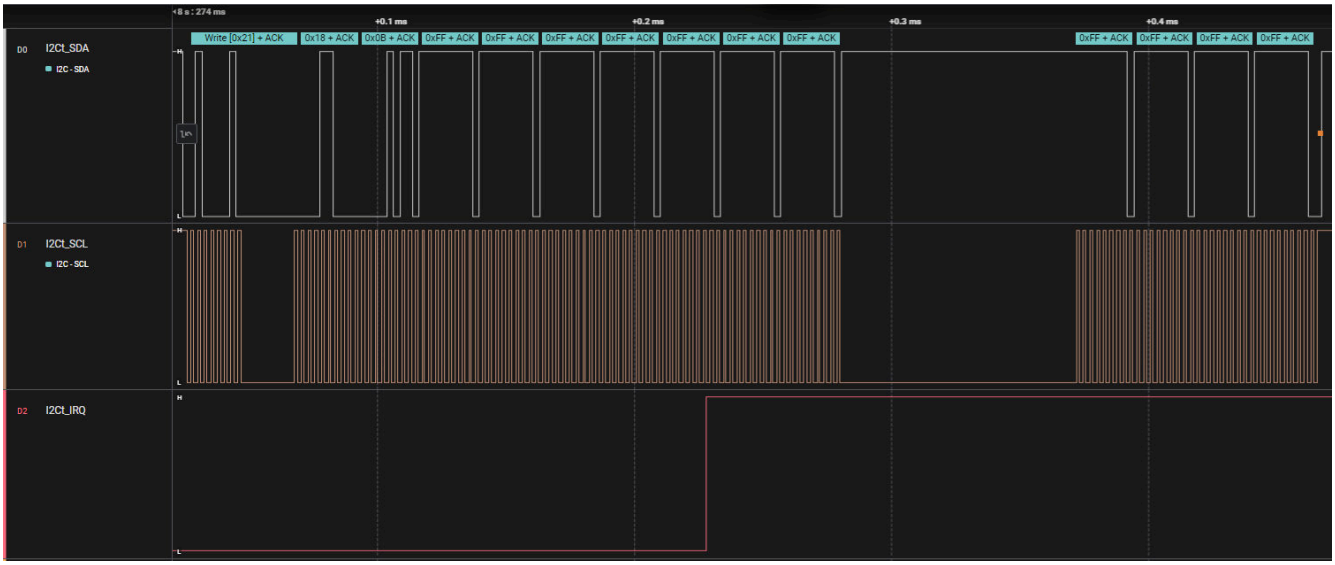


**图 4-4. 中断清除寄存器 0x18**

4. 仅 ***PBMs*** 数据：

TRM 参考资料中定义了 PBMs 命令。对于本示例，表 4-1 中列出了 PBMs 的参数。

**表 4-1. PBMs 配置：DATA1 寄存器**

| 说明 | 值 | 注释 |
|---|---|---|
| 捆绑包大小 | 0x00002C80 | 请参阅 节 5 |
| I2C 突发数据目标地址 | 0x30 | 0x30，参阅参考资料 1。 |
| Timeout | 0x31 | 3.1 秒；参阅参考资料 1 |

[0x21] + ACK（唯一地址/WR/A）

0x09 + ACK（寄存器编号/A）

0x06（字节计数）

0x80 0x2C 0x00 0x00 0x30 0x32（捆绑包大小、I2C 目标地址、超时值）

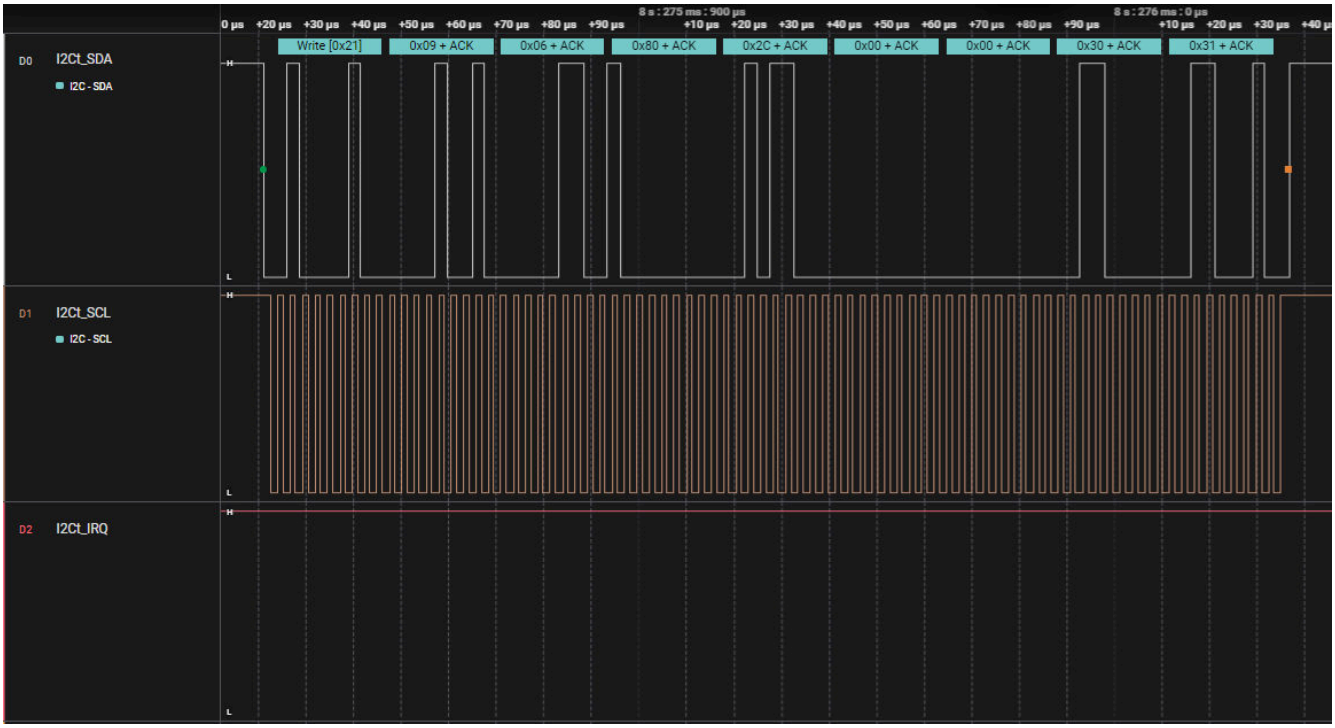


**图 4-5. 用于 PBMs 的 DATA1 寄存器 0x09**

5. **数据正常：**

发送 PBMs 命令需要多次写入 DATA 寄存器。例如，在将 PBMs 命令写入寄存器 0x08 之前，确认 0x09 的值。写入和读取寄存器 0x09 之间存在 500us 延迟。

[0x21] + ACK（唯一地址/WR/A）

0x09 + ACK（寄存器编号/A）

[0x21] + ACK（唯一地址/R/A）

0x40（字节计数）

0x00 0x00 0x00 0x00 0x00 0x00（错误，重写 DATA1）

0x80 0x2C 0x00 0x00 0x30 0x32（正确，继续写入 CMD1）

6. **PBMs CMD1：**

   确认 DATA1 后，写入 CMD1 = *PBMs*。I2Ct_IRQ 置为低电平有效，如 图 4-6 所示。

   [0x20] + ACK（唯一地址/WR/A）

   0x08 + ACK（寄存器编号/A）

   0x04（字节计数）

   0x50 0x42 0x4D 0x73（以 4 个 ASCII 字符表示的 *PBMs*）

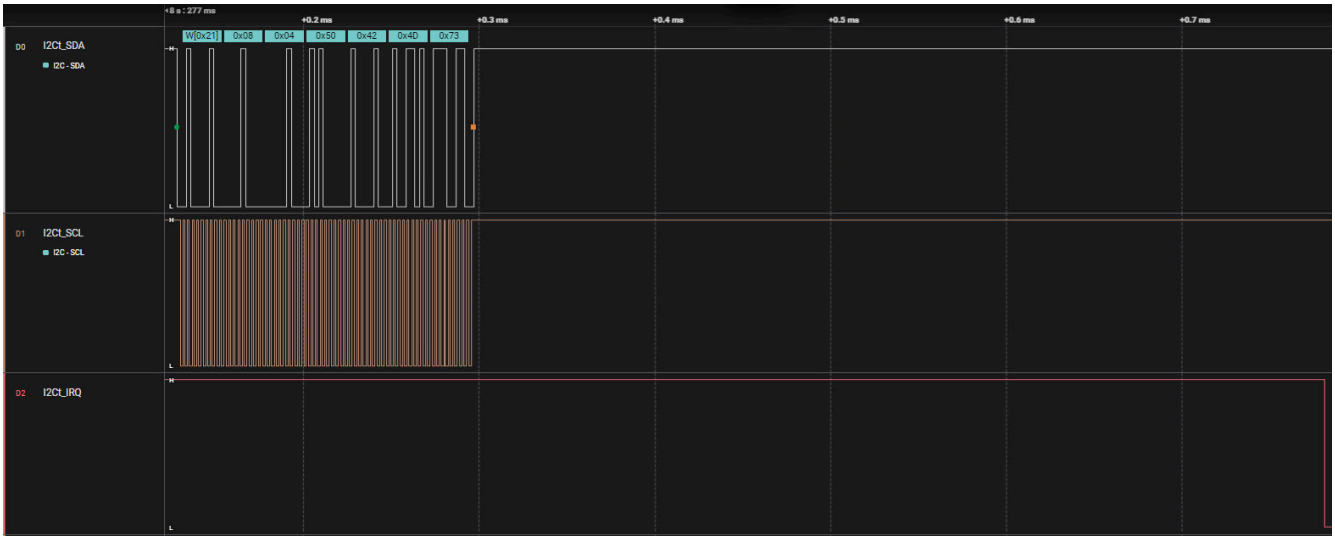


**图 4-6. 将 PBMs 写入 CMD1 寄存器 0x08**

7. **I2Ct_IRQ == 低电平**

   IRQ 信号表示 CMD1 完成事件何时发生，并可读取 CMD1 和 DATA 寄存器以确认 PBMs 命令的结果。预期结果如第 8 和 9 步所示。

8. **CMD1 清除 (PBMs)**

   命令寄存器 0x08 表示在清除寄存器内容时命令已成功完成。为简单起见，示例仅检查第一位以确认内容不是“!CMD”，这表明 PBMs 命令已损坏或 DATA 寄存器加载了非法值 [4]

   [0x21] + ACK（唯一地址/WR/A）

   0x08 + ACK（寄存器编号/A）

   [0x21] + ACK（唯一地址/R/A）

   0x04（字节计数）

   0x00 0x00 0x00 0x00

[4] 如果未为 CMD1 完成使用 IRQ，可以轮询 CMD1 寄存器，直到内容从 [0x50, 0x42, 0x4D, 0x73] 切换为 [0x00, 0x00, 0x00, 0x00] 或 [0x21, 0x43, 0x4D, 0x44]。CMD1[0]=0x21 指示命令未成功完成，CMD1[0]=0x00 指示成功完成。

9. **DATA1 清除 (PBMs)**

   当第一个字节 PatchStartStatus 被清除时，数据寄存器 0x09 指示修补成功。PatchStartStatue 的非零值 0x04、0x05 和 0x06 值分别表示无效的捆绑包大小、目标地址或超时值。具体请参阅 1。

   [0x21] + ACK（唯一地址/WR/A）

   0x09 + ACK（寄存器编号/A）

   [0x21] + ACK（唯一地址/R/A）

   0x40（字节计数）

   0x00 0x00 0x00 0x00 0x30 0x31

10. **突发数据**

    在这一步，不使用 PMBUS 格式，二进制映像的内容直接写入 PBMs 命令中指定的 I2C 突发数据目标地址，见参考表。突发格式受 MCU 架构影响。在本例中，突发大小限制为 4KB，因此会向 PD 发送三个连续突发数据（4,095 字节、4,095 字节和 3,202 字节），每次突发之间延迟 500us。末次突发结束后额外延迟 500 µs 再发送 PBMc 命令。

    [0x30] + ACK（唯一地址/WR/A）

    lowRegion_i2c_array[0], lowRegion_i2c_array[1]..., lowRegion_i2c_array[4094]

    [0x30] + ACK（唯一地址/WR/A）

    lowRegion_i2c_array[4095], lowRegion_i2c_array[4096]..., lowRegion_i2c_array[8189]

    [0x30] + ACK（唯一地址/WR/A）

    lowRegion_i2c_array[8190], lowRegion_i2c_array[8191]..., lowRegion_i2c_array[11391]

11. **配置 I2Ct_IRQ、CMD1 完成**

    CMD1 完成中断用于提醒 EC：PBMc 命令已完成。分别通过寄存器 0x16 和 0x18 设置中断屏蔽和清除中断。第 3 步已设置了中断屏蔽。重复中断清除操作，如 图 4-4 和 图 4-7 所示。

12. **仅限 PBMc CMD1**

    PBMc 命令不包括输入数据，因此仅发送该命令。

    [0x21] + ACK（唯一地址/WR/A）

    0x08 + ACK（寄存器编号/A）

    [0x21] + ACK（唯一地址/R/A）

    0x04（字节计数）

    0x50 0x42 0x4D 0x63（以 4 个 ASCII 字符表示的 PBMc）

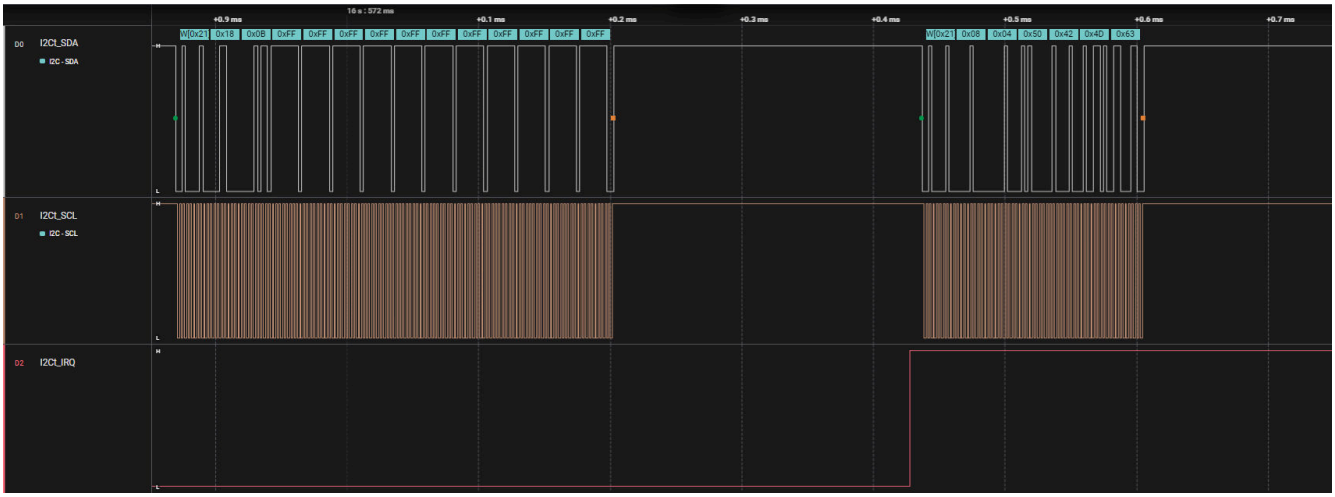


**图 4-7. 清除中断并将 PBMc 写入 CMD1 寄存器**

13. **I2Ct_IRQ == 低电平**

    IRQ 信号表示 CMD1 完成事件何时发生，并可读取 CMD1 寄存器以确认 PBMc 命令的结果。

14. **CMD1 清除 (PBMc)**

    与 PBMs CMD1 清除类似，读取 CMD1 寄存器并确认原始命令已清除且不等于 *!CMD*。


**图 4-8. I2Ct_IRQ 置为有效后读取 CMD1 (PBMc)**

15. **延迟 20ms** [5]

    20ms 延迟允许 PD 控制器加载并应用映像。一旦延迟结束，就会读取 DATA1 和 MODE 寄存器以确认成功。

[5] 20ms 延迟可替换为“补丁已加载”中断。使用该中断代替延迟未能实现时间缩短。因此，本文档中保留了器件 TRM 中描述的 20ms 延迟。请参阅 1 和 2


**图 4-9. CMD1 和 DATA1 寄存器读取之间的延迟**

16. **DATA1 清除 (PBMc)**

在这个实例中，从 DATA1 寄存器读取了 40 个字节。

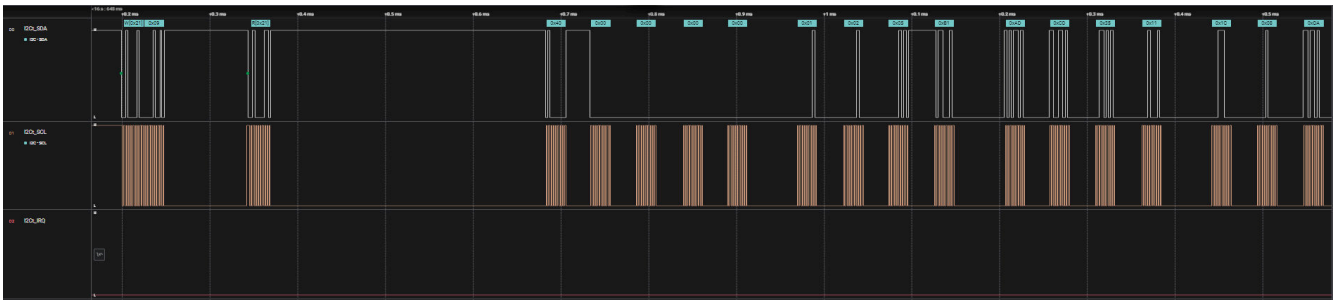

**图 4-10. PBMc 完成后读取 DATA1 寄存器**

17. **模式 == APP**

最后一步是验证 PD 是否已切换为 APP 模式。进入 APP 模式后，PD 控制器现在可以在应用的自定义配置下运行。

[0x21] + ACK（唯一地址/WR/A）

0x03 + ACK（寄存器编号/A）

[0x21] + ACK（唯一地址/R/A）

0x04（字节计数）

0x41 0x50 0x50 0x20（以 4 个 ASCII 字符表示的‘APP’）

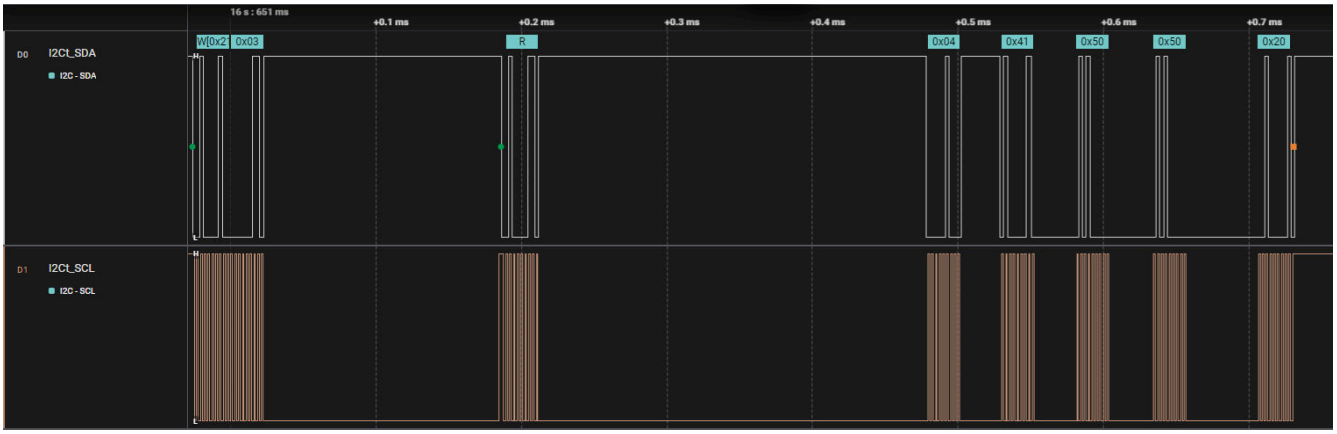


**图 4-11. APP 模式**

## 4.2 生成低区二进制文件的步骤

USBC PD 应用程序自定义工具 (GUI) 利用大量问题来生成自定义 PD 控制器配置。配置完成后，可以生成包含配置和固件更新的补丁捆绑包。完整的闪存二进制文件会生成一个映像，具有 PD 控制器通过 I2Cc 从 EEPROM 读取时所期望的格式。低区二进制文件（见 图 4-12）是一种较小的格式，用于通过 I2Ct 写入 PD 控制器。这种低区二进制文件是在第 10 步发送的内容。

1. 回答 GUI 中的调查问卷。
2. 如有必要，使用 Advanced Configuration 开关配置任何其他设置。
3. 从 Export 下拉菜单中进行选择，以生成低区二进制文件。
4. 选择合适的格式和期望的文件名。

如 图 4-13 所示，GUI 提供了将低区文件导出为二进制文件或 c 文件的选项。本应用手册选择了 c 文件格式，并将其包含在用于构建 EC 代码的器件项目中。可以在 节 5 中找到从 GUI 生成的源文件和相关定义的修改版。

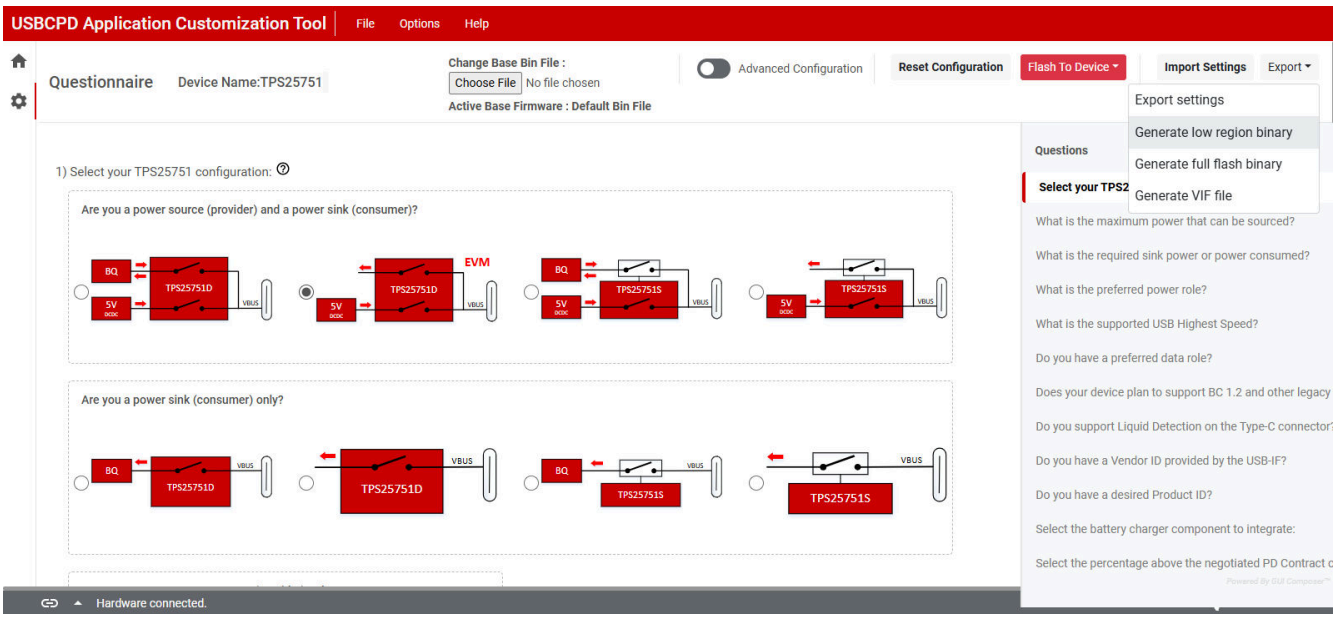


**图 4-12. 从 USBCPD 应用程序自定义工具生成映像**

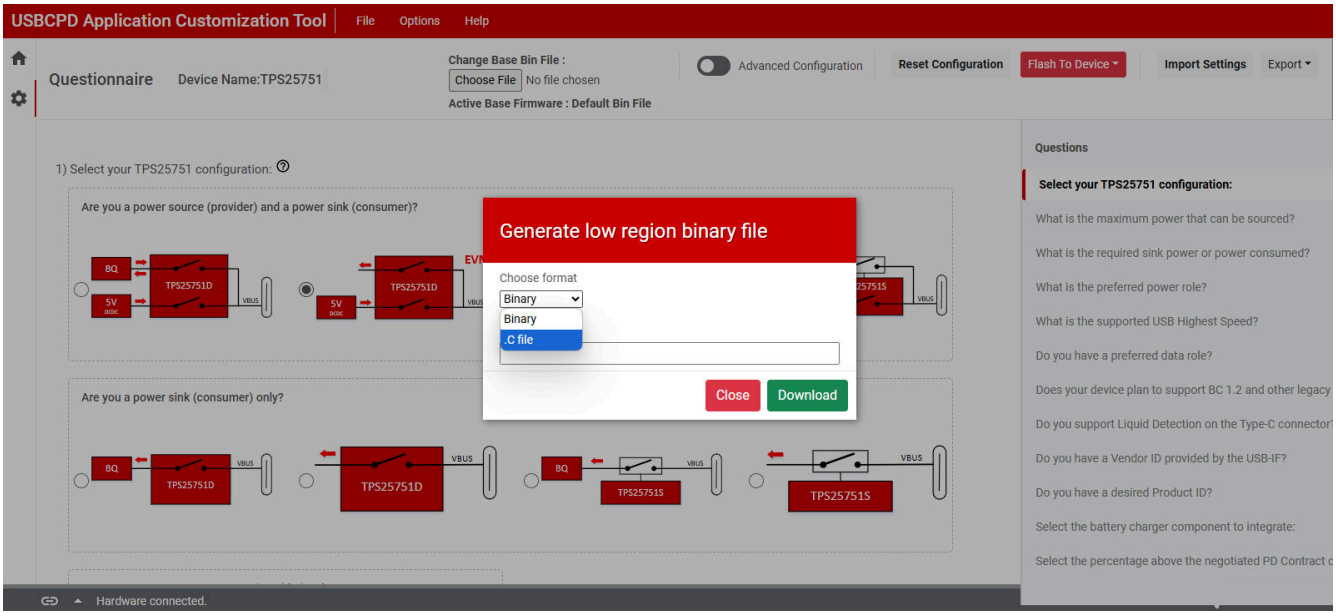


**图 4-13. 选择映像输出类型**

# 5 示例代码

对从 GUI 生成的映像文件进行了轻微修改，如以下代码所示。

```
#include "ti_msp_dl_config.h"
#include "lrb.h"

const uint8_t tps25751_lowRegion_i2c_array[SIZEOFLRB] = {
0x01, 0x00, 0xe0, 0xac, 0xfe, 0xff, 0xff, 0xff, 0x80, 0x06, 0x00, 0x00, 0x00, 0x26, 0x00, 0x00,
0x71, 0x3e, 0x9c, 0xb8, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
...
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
```

文件 *lrb.h* 如以下代码所示，用于提供上下文。

```
#define SIZEOFLRB  0x2C80   // 11392
/* Maximum size of TX packet, MCU 0xFFF */
#define I2C_TX_MAX_PACKET_SIZE
/* SIZEOFLRB/I2C_TX_MAX_PACKET_SIZE +1 */
#define LRB_NUMBER_OF_PACKETS  (0x0003)
/* SIZEOFLRB % I2C_TX_MAX_PACKET_SIZE */
#define LRB_REMAINDER          (0x0C82)
extern const uint8_t tps25751_lowRegion_i2c_array[SIZEOFLRB];
```

以下示例代码基于 MSPM0C110x 驱动程序库 [6] 和节 4.1 中的步骤。

```
    /*
     *  1. Wait for Ready for Patch interrupt
     */
    debounce = 1;
    /* Skip glitch at boot */
    while(debounce)
    {
        while(DL_GPIO_readPins(
                GPIO_SWITCHES_PORT, GPIO_SWITCHES_USER_SWITCH_1_PIN));
        delay_cycles(DELAY_500us);
        if(!DL_GPIO_readPins(
                GPIO_SWITCHES_PORT, GPIO_SWITCHES_USER_SWITCH_1_PIN))
        {
             debounce = 0;
        }
    }
    /*
     *  2. Mode == PTCH
     */
    readRegister(MODE_REGISTER, MODE_BYTE_CNT, gRxPacket);
    if(gRxPacket[1] != 0x50)
    {
        delay_cycles(DELAY_10ms);
        readRegister(MODE_REGISTER, MODE_BYTE_CNT, gRxPacket);
    }
    /*
     *  3a. Configure I2Ct_IRQ for CMD1 Complete
     */                                                  */
    gTxPacket = (uint8_t*)maskIntReg;
    writeRegister(INT_BYTE_CNT, (uint8_t*)gTxPacket);
    waitingForPD = 1;
    while(waitingForPD)
    {
        /*
         *  3b. Clear Interrupts
         */
        gTxPacket = (uint8_t*)clrIntReg;
        writeRegister(INT_BYTE_CNT, (uint8_t*)gTxPacket);
        //  Wait for interrupt to clear
        while(DL_GPIO_readPins(GPIO_SWITCHES_PORT, GPIO_SWITCHES_USER_SWITCH_1_PIN)==0);
        waitingForGoodData = 1;
        while(waitingForGoodData)
```

[6] MSPM0 SDK (2.05.00.05)/Examples/Development Tools/LP_MSPM0C1104 LaunchPad/DriverLib/i2c_controller_rw_multiple_fifo_interrupts

```
        {
            /*
             *  4. Send PBMs Data
             */
            gTxPacket = (uint8_t*)PBMsData;
            writeRegister(DATA_BYTE_CNT, gTxPacket);
            delay_cycles(DELAY_500us);
            /*
             * 5. Data Good
             */
            readRegister(DATA_REG, (DATA_BYTE_CNT-1), gRxPacket);
            if(gRxPacket[1] == 0x80)
            {
                waitingForGoodData = 0;
            }
        }
        /*
         *  6. Send PBMs to CMD1
         */
        gTxPacket = (uint8_t*)PBMsCmd;
        writeRegister(CMD_BYTE_CNT, gTxPacket);
        /*
         *  7. Wait for I2Ct_IRQ
         */
        while(DL_GPIO_readPins(
                    GPIO_SWITCHES_PORT, GPIO_SWITCHES_USER_SWITCH_1_PIN));
        /*
         *  8. CMD1 Clear
         */
        readRegister(CMD_REG, (CMD_BYTE_CNT-1), gRxPacket);
        /*
         *  9. DATA1 Clear
         */
        readRegister(DATA_REG, (DATA_BYTE_CNT-1), gRxPacket);
        if(gRxPacket[1] == 0x0)
        {
            waitingForPD = 0;
        }
    }
    /*
     *  10. Burst Data
     */
    ii = LRB_NUMBER_OF_PACKETS;
    while(ii)
    {
        if(ii == 1)
        {
            gTxLen = LRB_REMAINDER;
        }
        else
        {
            gTxLen = I2C_TX_MAX_PACKET_SIZE;
        }
        /* The FIFO is 8-bytes deep, and this function returns number of bytes written to FIFO */
        gTxPacket = (uint8_t*)&tps25751_lowRegion_i2c_array[(LRB_NUMBER_OF_PACKETS-
ii)*I2C_TX_MAX_PACKET_SIZE];
        ii = ii-1;
        gTxCount = DL_I2C_fillControllerTXFIFO(I2C_INST, gTxPacket, gTxLen);
        DL_I2C_enableInterrupt(
                I2C_INST, DL_I2C_INTERRUPT_CONTROLLER_TXFIFO_TRIGGER);
        gI2cControllerStatus = I2C_STATUS_TX_STARTED;
        while (!(
            DL_I2C_getControllerStatus(I2C_INST) & DL_I2C_CONTROLLER_STATUS_IDLE))
            ;
        DL_I2C_startControllerTransfer(
            I2C_INST, I2C_TGT_BURST_ADDRESS, DL_I2C_CONTROLLER_DIRECTION_TX, gTxLen);
        while ((gI2cControllerStatus != I2C_STATUS_TX_COMPLETE) &&
            (gI2cControllerStatus != I2C_STATUS_ERROR)) {
            __WFE();
        }
        while (DL_I2C_getControllerStatus(I2C_INST) &
            DL_I2C_CONTROLLER_STATUS_BUSY_BUS)
            ;
        /* Trap if there was an error */
        if (DL_I2C_getControllerStatus(I2C_INST) &
            DL_I2C_CONTROLLER_STATUS_ERROR) {
            __BKPT(0);
        }
```

```
        while(!(
            DL_I2C_getControllerStatus(I2C_INST) & DL_I2C_CONTROLLER_STATUS_IDLE))
            ;
        delay_cycles(DELAY_500us);
    }
    /*
     *  11. Clear Interrupts
     */
    gTxPacket = (uint8_t*)clrIntReg;
    writeRegister(INT_BYTE_CNT, (uint8_t*)gTxPacket);
    //  Wait for interrupt to clear
    while(DL_GPIO_readPins(
        GPIO_SWITCHES_PORT, GPIO_SWITCHES_USER_SWITCH_1_PIN)==0)
        ;
    /*
     *  12. Send PBMc to CMD1 Register
     */
    gTxPacket = (uint8_t*)PBMcCmd;
    writeRegister(CMD_BYTE_CNT,gTxPacket);
    /*
     *  13. I2Ct_IRQ == Low, CMD1 complete
     */
    while(DL_GPIO_readPins(
        GPIO_SWITCHES_PORT, GPIO_SWITCHES_USER_SWITCH_1_PIN));
    /*
     *  14. Read CMD1
     */
    readRegister(CMD_REG, (CMD_BYTE_CNT-1), gRxPacket);
    /*
     *  15. Delay 20ms
     */
    delay_cycles(DELAY_20ms);
    /*
     *  16. Read Data1
     */
    readRegister(DATA_REG, 0x28, gRxPacket);
    /*
     *  17. MODE === APP
     */
    waitingForPD = 1;
    while(waitingForPD)
    {
        readRegister(MODE_REGISTER, MODE_BYTE_CNT, gRxPacket);
        if(gRxPacket[1] == 0x41)
        {
            waitingForPD = 0;
        }
        else
        {
            delay_cycles(DELAY_10ms);
        }
    }
```

## 6 参考资料

1. 德州仪器 (TI)，*TPS25751 技术参考手册*，技术参考手册。
2. 德州仪器 (TI)，*TPS26750 技术参考手册*，技术参考手册。
3. 德州仪器 (TI)，*具有集成式电源开关的 TPS25751 USB Type-C® 和 USB PD 控制器*，数据表。
4. 德州仪器 (TI)，*针对电源应用进行了优化且具有集成式电源开关的 TPS26750 USB Type-C® 和 USB PD 控制器*，数据表。
5. 德州仪器 (TI)，*MSPM0 软件开发套件 (SDK)*，软件
6. 德州仪器 (PTCH)，*从 PTCH 到 APP*，E2E™ 设计支持论坛。
7. 德州仪器 (TI)，*Salae 代码*，E2E™ 设计支持论坛。
8. 德州仪器 (TI)，*屏蔽*，E2E™ 设计支持论坛。

# 7 修订历史记录

| Changes from Revision * (July 2024) to Revision A (June 2025) | Page |
|---|---|
| • 向文档添加了 TPS26750 | 1 |
| • 更新了整个文档中的表格、图和交叉参考的编号格式 | 1 |
| • 删除了对 TPS25751 的具体引用，采用通用描述 | 2 |
| • 删除了对 TPS25751 的具体引用 | 2 |
| • 将范围更改为补丁突发模式 | 3 |

# 重要通知和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、某特定用途方面的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他功能安全、信息安全、监管或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的相关应用。 严禁以其他方式对这些资源进行复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。您应全额赔偿因在这些资源的使用中对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，TI 对此概不负责。

TI 提供的产品受 TI 的销售条款或 ti.com 上其他适用条款/TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

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