Subsystem Design

USB 转 I2C 桥接器

1 说明

此子系统演示了如何将 MSPM0 器件实现为 USB 通信器件类抽象控制模型 (USB CDC-ACM) 转 I2C 桥接器。当连接至 USB 主机时，该器件会枚举为虚拟 COM 端口。在 Windows®、macOS® 或 Linux® 上，无需安装自定义驱动程序。

桥接器将 USB 主机数据作为写入事务转发至 I2C 总线，并将 I²C 从器件读取数据返回到 USB 主机。I²C 从器件通过专用 GPIO 引脚（数据就绪信号）指示数据可用性，该引脚会触发 I²C 读取事务。此示例使用开源 TinyUSB 库作为 USB 栈。此示例支持两个目标平台：MSPM0G5187（LaunchPad™、LQFP-64）和 MSPM0C5116（LaunchPad、LQFP-48）。这两个平台运行相同的桥接应用代码，但具有针对各自器件的 SysConfig 外设分配。

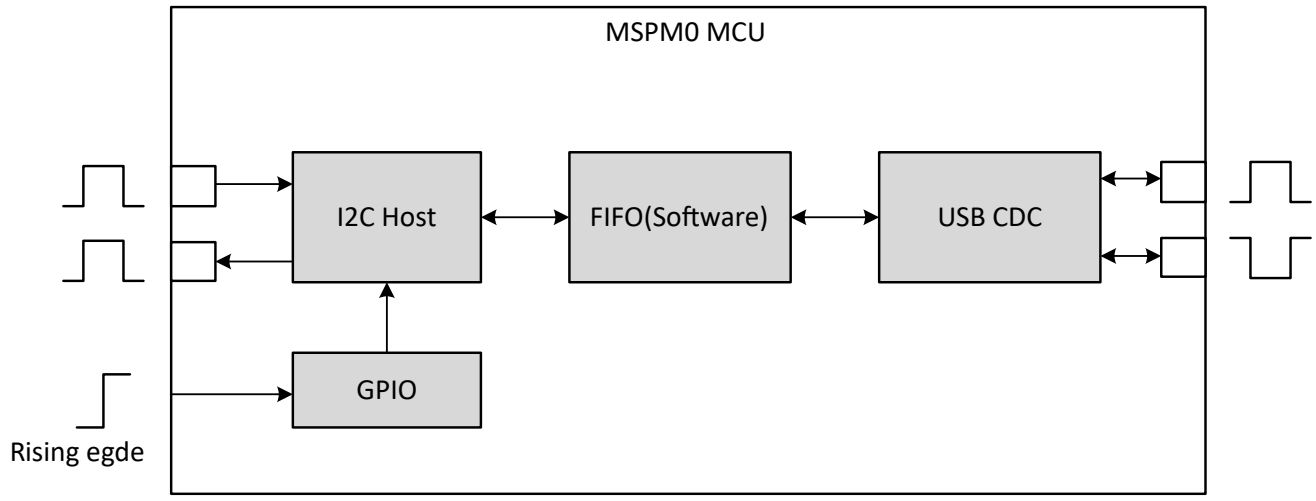

图 1-1. 系统功能方框图

2 所需外设

两个平台使用相同的外设类型。SysConfig 会根据具体器件分配对应的实例，但会生成一致的宏名称（I2C_0_INST、DMA_CH0_CHAN_ID、DMA_CH1_CHAN_ID），因此应用代码无需修改即可在两个器件上直接编译。

表 2-1. 外设信息

外设	功能	MSPM0G5187	MSPM0C5116
USB	USB 全速 CDC 器件	USBFS0 (USB_0)	USBLC0 (USB_0)
I2C	I2C 控制器到目标器件	UC1 (I2C_0_INST)	UC16 (I2C_0_INST)
DMA(CH1)	I2C TX 加速	DMA_CH1	DMA_CH1
DMA(CH0)	I2C RX 加速	DMA_CH0	DMA_CH0
GPIO	目标数据就绪通知输入	PB25	PB24

3 兼容器件

- MSPM0G5187
- MSPM0G11x
- MSPM0C511x

4 设计步骤

1. 在 SysConfig 中设置 TinyUSB 模块。添加一个 CDC 实例 (CDC_0)。SysConfig 会自动分配三个端点：通知 IN (EP_IN2)、数据 IN (EP_IN1) 和数据 OUT (EP_OUT1)。
2. 配置 USB 时钟树。两个平台使用不同的时钟路径。在 MSPM0G5187 上：使能 PLL 和 USB FLL，将 UDIV 设置为 2，并选择 HSCLKMUX_SYSPLL0。在 MSPM0C5116 上：仅使能 USB FLL（无需 PLL），将 UDIV 设置为 2，并选择 HSCLKMUX_USBFLL。
3. 在 SysConfig 中设置 I2C 模块。
   a. 启用 I²C 控制器模式。
   b. 在 MSPM0G5187 上将总线速度设置为快速模式增强版 (1MHz)，或在 MSPM0C5116 上将总线速度设置为快速模式 (400kHz)。
   c. 将 DMA event1 触发器配置为 CONTROLLER_RXFIFO_TRIGGER，并在*固定地址到块地址*模式下分配 DMA_CH0。
   d. 将 DMA event2 触发器配置为 CONTROLLER_TXFIFO_TRIGGER，并在*块地址到固定地址*模式下分配 DMA_CH0。
   e. 在“Interrupt Configuration”选项卡中，启用 EVENT1_DMA_DONE、EVENT2_DMA_DONE 和 NACK 中断。
4. 在 SysConfig 中设置 GPIO 模块。
   a. 添加一个名为 I2C_SIGNAL 的输入 GPIO 引脚，将其配置为带下拉电阻的上升沿中断。此引脚连接到 I2C 目标器件的数据就绪输出。
   b. 启用输入中断。在 MSPM0G5187 上，将该 GPIO 映射到中断组 1。

```
void GROUP1_IRQHandler(void)
{
    switch (DL_Interrupt_getPendingGroup(DL_INTERRUPT_GROUP_1)) {
        case GPIO_INT_IIDX:
            switch (DL_GPIO_getPendingInterrupt(GPIO_PORT)) {
                case GPIO_I2C_SIGNAL_IIDX:
                    // Slave signaled data ready - set flag for main loop
                    i2ctargetDataReady = true;
                    break;

                default:
                    break;
            }
            break;

        default:
            break;
    }
}
```

在 MSPM0C5116 上，直接使用 GPIOB 中断。

```
void GPIOB_IRQHandler(void)
{
    switch (DL_GPIO_getPendingInterrupt(GPIO_PORT)) {
        case GPIO_I2C_SIGNAL_IIDX:
            // Slave signaled data ready - set flag for main loop
            i2ctargetDataReady = true;
            break;

        default:
            break;
    }
}
```

5. 在 SysConfig 中设置 SysTick。使能 SysTick 并设置与 CPU 频率匹配的周期：MSPM0G5187 (80MHz) 设为 80000 个周期，MSPM0C5116 (48MHz) 设为 48000 个周期。这两个设置值均可产生 1ms 的周期。TinyUSB 的 board_millis() 需要此 1ms 周期。

5 设计注意事项

1. *缓冲区大小设置：*USB_FRAME_PACKAGE 默认值为 64 字节，与 USB 全速模式的最大数据包大小相匹配。每个 FIFO 队列最多可容纳 8 个任务 (USB_TX_BUF_SIZE/I2C_TX_BUF_SIZE)，即每个方向可缓存 512 字节。增加队列深度可以处理突发流量，但会占用更多 RAM 资源。

```
// ===== Queue Configuration =====
#define USB_TX_BUF_SIZE     8              // Max tasks in USB→I2C queue
#define I2C_TX_BUF_SIZE    8              // Max tasks in I2C→USB queue

// ===== USB/I2C Parameters =====
#define USB_FRAME_PACKAGE   64             // USB Full Speed max packet size (bytes)
```

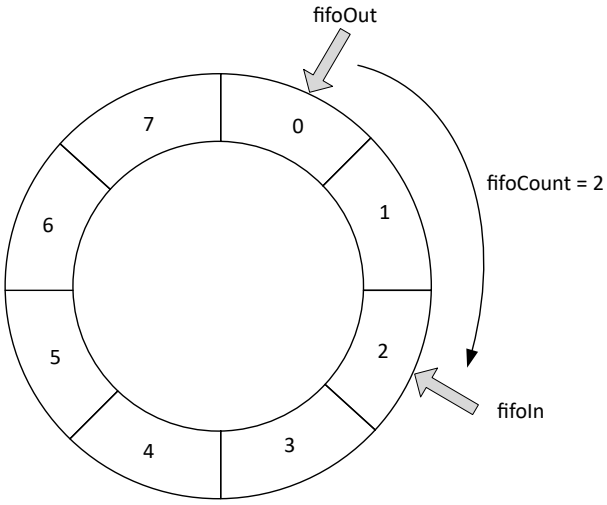

图 5-1. 缓冲区结构

```
// Represents one 64-byte unit of data (one USB frame)
typedef struct {
    uint8_t buffer[USB_FRAME_PACKAGE];   // Data payload (up to 64 bytes)
    uint16_t length;                     // Actual data length in buffer
} TxElement_t;

// ===== FIFO Queue Structure =====
// Circular queue to manage multiple tasks
typedef struct {
    uint16_t fifoIn;                     // Write pointer (next task to add)
    uint16_t fifoOut;                    // Read pointer (next task to process)
    uint16_t fifoCount;                  // Number of pending tasks in queue
    TxElement_t *fifoPointer;            // Pointer to task array
} TxFIFO_t;
```

2. *GPIO 数据就绪信号：*当 I2C 目标器件数据准备就绪供控制器读取时，其会将专用 GPIO 输出置为有效（高电平有效）。桥接器通过 GPIO 中断检测上升沿，并设置 i2ctargetDataReady 标志。主循环轮询此标志并调用 i2c2usb_pushTask() 以启动 I2C 读取事务。目标必须将 GPIO 保持在高电平，直到控制器完成读取。GPIO 引脚使用内部下拉电阻器，因此当目标没有数据时，该线路处于空闲状态。
3. *用于 I2C TX 的 DMA：*DMA 通道 CH1 在*块地址到固定地址*模式下运行，并直接从任务缓冲区加载 I2C TX FIFO，而无需 CPU 参与。DMA 将最后一个字节加载到 FIFO 后，触发 DMA 完成中断。然后，桥接器动态启用 I²C STOP 条件中断。仅当触发 STOP 中断（表示总线上的最后一位已发送）时，桥接器才会释放 TX 任务并清除 i2cInProgress。

4. *用于 I2C RX 的 DMA*：DMA 通道 CH0 在*固定地址到块地址*模式下运行，并将接收到的字节从 I2C RX FIFO 直接移动到任务缓冲区。接收到预期的字节数后，发生 DMA 完成中断。处理程序将任务标记为 USB 传输准备就绪，并清除 i2ctargetDataReady 和 i2cInProgress。
5. *溢出保护：*当 USB 转 I2C 队列已满时，传入的 USB 数据将读取到虚拟缓冲区中，以防止 TinyUSB RX 缓冲区停止。当 I2C 转 USB 队列已满时，i2c2usb_pushTask() 函数会立即返回，而不会启动新的 I²C 读取。对于生产设计，请添加应用层流控制或增加队列深度。

6 软件流程图

图 6-1 展示了此示例的代码流程。

USB 转 I2C 方向：

1. 每当 USB 主机发送数据时，TinyUSB 都会调用 tud_cdc_rx_cb()。该回调函数会调用 usb2i2c_pushTask()，从 USB CDC 接口读取最多 64 个字节，并将其作为任务加入 gUSB2I2CTask 队列。
2. 在主循环中，当 gUSB2I2CTask.fifoCount > 0 时，usb2i2c_popTask() 会使用任务缓冲区的地址和长度配置 DMA 通道 CH1，然后启动对目标器件的 I²C 写传输。
3. 当 DMA 完成对 I²C TX FIFO 的加载时，会触发 TX DMA 完成中断。处理程序清除 STOP 中断状态标志并启用 STOP 中断。总线上出现 I²C STOP 条件时，将触发 STOP 中断。处理程序调用 fifoCompleteRead() 以释放任务并清除 i2cInProgress。

I2C 转 USB 方向：

1. I2C 目标器件将数据就绪 GPIO 引脚（上升沿）置为有效。GPIO 中断处理程序置位 i2ctargetDataReady
2. 在主循环中，当 i2ctargetDataReady 被置位且 i2cInProgress 为 false 时，i2c2usb_pushTask() 将 DMA 通道 CH0 配置为接收缓冲区，开始 64 个字节的 I2C 读取传输，并置位 i2cInProgress。数据长度始终固定为 64 个字节。
3. 当接收到所有 64 个字节时，将触发 RX DMA 完成中断。处理程序调用 fifoCompleteWrite() 将任务标记为 USB 传输准备就绪，清除 i2ctargetDataReady，然后清除 i2cInProgress。
4. 在主循环中，当 gI2C2USBTask.fifoCount > 0 时，i2c2usb_popTask() 检查 TinyUSB CDC TX FIFO 是否有足够的空间，写入缓冲的数据，将其刷新到主机，然后调用 fifoCompleteRead() 以释放该任务。

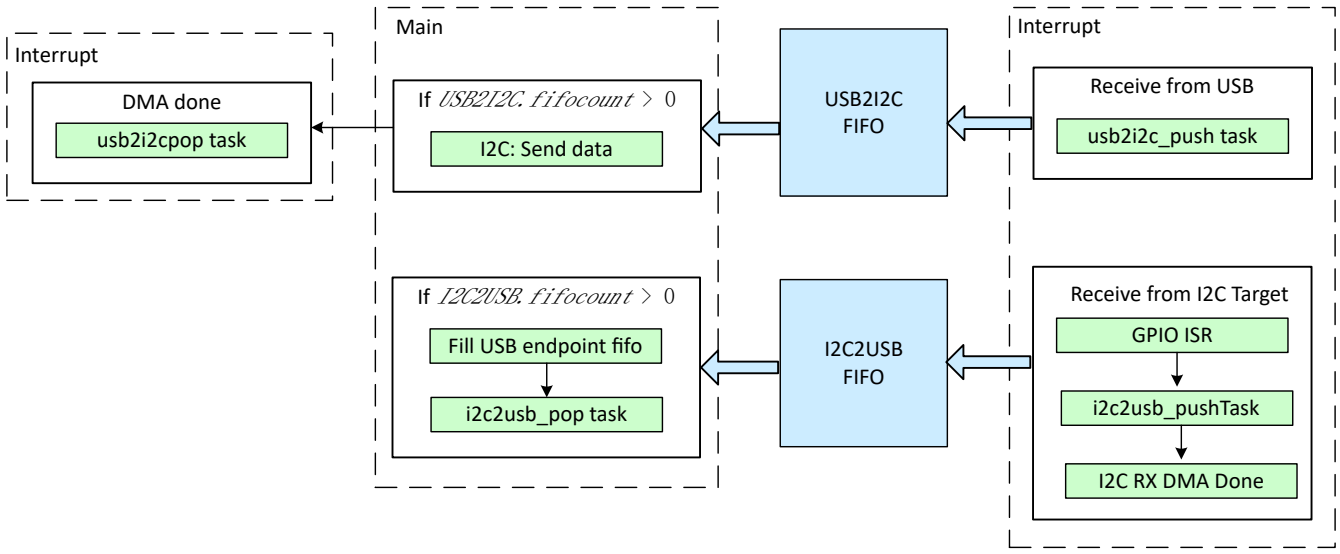

图 6-1. 应用软件流程图

7 应用代码

USB 转 I2C：

```
/*********************************************************
* Function name  : usb2i2c_pushTask
* Description    : USB -> I2C. Read data from USB CDC interface and push
*                  into gUSB2I2CTask queue. Each task holds one USB frame
*                  (up to 64 bytes). Called from tud_cdc_rx_cb callback.
* Parameter      : None (modifies gUSB2I2CTask)
* Return         : None
*********************************************************/
void usb2i2c_pushTask(void)
{
    // Check space for new task
    if(gUSB2I2CTask.fifoCount >= I2C_TX_BUF_SIZE) return;

    // Get current task slot in queue
    TxElement_t *pTask = &gUSB2I2CTask.fifoPointer[gUSB2I2CTask.fifoIn];

    // Read up to 64 bytes from USB CDC
    uint32_t cnt = tud_cdc_n_read(0, pTask->buffer, 64);
    if(cnt == 0) return;                    // Skip if no data

    pTask->length = cnt;

    // Advance queue pointer and mark task as ready
    fifoCompleteWrite(&gUSB2I2CTask, I2C_TX_BUF_SIZE);
}
```

```
/*********************************************************
* Function name  : usb2i2c_popTask
* Description    : USB -> I2C. Pop task from queue and transmit data through
*                  I2C to slave device. Waits if I2C TX already in progress.
*                  Called from main loop when fifoCount > 0.
* Parameter      : None (modifies gUSB2I2CTask, i2cInProgress)
* Return         : None
*********************************************************/
void usb2i2c_popTask(void)
{
    if(i2cInProgress) return;

    if(gUSB2I2CTask.fifoCount == 0) return;

    // Current process task
    TxElement_t *pTask = &gUSB2I2CTask.fifoPointer[gUSB2I2CTask.fifoOut];

    /* Setup DMA and start I2C write */
    DL_DMA_setSrcAddr(DMA, DMA_CH1_CHAN_ID, (uint32_t)(pTask->buffer));
    DL_DMA_setDestAddr(DMA, DMA_CH1_CHAN_ID, (uint32_t)(&I2C_0_INST->i2cc->TXDATA));
    DL_DMA_setTransferSize(DMA, DMA_CH1_CHAN_ID, pTask->length);
    DL_DMA_enableChannel(DMA, DMA_CH1_CHAN_ID);

    DL_I2CC_startTransfer(I2C_0_INST, I2C_ADDRESS, DL_I2CC_DIRECTION_TX, pTask->length);

    i2cInProgress = true;
}
```

I2C 转 USB：

```
/*********************************************************
* Function name  : i2c2usb_pushTask
* Description    : I2C -> USB. Read data from I2C slave and queue for USB transmission
*                  Triggered when slave device signals data ready via GPIO pin.
*                  Called from GPIO interrupt handler.
* Parameter      : None (modifies gI2C2USBTask)
* Return         : None
*********************************************************/
void i2c2usb_pushTask(void)
{
    // Check space for new task
    if (gI2C2USBTask.fifoCount >= USB_TX_BUF_SIZE) return;
    if(i2cInProgress) return;

    // Current process task
    TxElement_t *pTask = &gI2C2USBTask.fifoPointer[gI2C2USBTask.fifoIn];
    pTask->length = USB_FRAME_PACKAGE;

    /* Setup DMA and initiate I2C read */
    DL_DMA_setSrcAddr(DMA, DMA_CH0_CHAN_ID, (uint32_t)(&I2C_0_INST->i2cc->RXDATA));
    DL_DMA_setDestAddr(DMA, DMA_CH0_CHAN_ID, (uint32_t)(pTask->buffer));
    DL_DMA_setTransferSize(DMA, DMA_CH0_CHAN_ID, USB_FRAME_PACKAGE);
    DL_DMA_enableChannel(DMA, DMA_CH0_CHAN_ID);

    DL_I2CC_startTransfer(I2C_0_INST, I2C_ADDRESS, DL_I2CC_DIRECTION_RX, USB_FRAME_PACKAGE);

    i2cInProgress = true;
}
```

```
/*********************************************************
* Function name  : i2c2usb_popTask
* Description    : I2C -> USB. Pop task from queue and transmit data through
*                  USB CDC interface. All data in one task is sent in same USB
*                  frame. Checks if USB TX buffer has space. Called from main
*                  loop when fifoCount > 0.
* Parameter      : None (modifies gI2C2USBTask)
* Return         : None
*********************************************************/
void i2c2usb_popTask(void)
{
    if(gI2C2USBTask.fifoCount == 0) return;   // Queue empty

    TxElement_t *pTask = &gI2C2USBTask.fifoPointer[gI2C2USBTask.fifoOut];

    if(tud_cdc_n_write_available(0) >= pTask->length){  // USB FIFO has space
        uint32_t written = tud_cdc_n_write(0, pTask->buffer, pTask->length);

        if(written == pTask->length){                   // Entire task sent atomically
            tud_cdc_n_write_flush(0);
            fifoCompleteRead(&gI2C2USBTask, USB_TX_BUF_SIZE);
        }
    }
}
```

I2C 中断处理程序：

```
//-----------------------------------------------------------------------+
// I2C Interrupt Handler
//-----------------------------------------------------------------------+
void I2C_0_INST_IRQHandler(void) {
    switch(DL_I2CC_getPendingInterrupt(I2C_0_INST)) {
        case DL_I2CC_IIDX_EVENT1_DMA_DONE:
            // TX: DMA transfer to I2C FIFO complete
            // IMPORTANT: Don't pop task yet - STOP condition still in progress
            DL_I2CC_clearInterruptStatus(I2C_0_INST, DL_I2CC_INTERRUPT_STOP);
            DL_I2CC_enableInterrupt(I2C_0_INST, DL_I2CC_INTERRUPT_STOP);
            break;

        case DL_I2CC_IIDX_STOP:
            // TX complete: I2C STOP condition detected - transmission finished
            // Only enabled when I2C Host write
            DL_I2CC_disableInterrupt(I2C_0_INST, DL_I2CC_INTERRUPT_STOP);
            fifoCompleteRead(&gUSB2I2CTask, I2C_TX_BUF_SIZE);
            i2cInProgress = false;
            break;

        case DL_I2CC_IIDX_EVENT2_DMA_DONE:
            // RX complete: DMA received 64 bytes from I2C slave
            fifoCompleteWrite(&gI2C2USBTask, USB_TX_BUF_SIZE);
            i2ctargetDataReady = false;  // Clear slave notification flag
            i2cInProgress = false;
            break;

        case DL_I2CC_IIDX_NACK:
            // I2C Error: Slave did not acknowledge address
            i2cInProgress = false;
            __BKPT(0);  // Halt for debugging - indicates I2C communication failure
            break;

        default:
            break;
    }
}
```

溢出处理：

```
void tud_cdc_rx_cb(uint8_t itf)
{
    if (gUSB2I2CTask.fifoCount < I2C_TX_BUF_SIZE) {
        usb2i2c_pushTask();
    } else {
        /* Queue full: discard to unblock TinyUSB RX buffer */
        uint8_t dummyBuf[64];
        tud_cdc_n_read(itf, dummyBuf, 64);
    }
}
```

8 数据流示例

场景 A：主机向目标写入 5 个字节

1. 主机通过 COM 端口发送 *HELLO*
2. tud_cdc_rx_cb() 将数据加入队列
   a. usb2i2c_pushTask()
   b. 队列 [H, E, L, L, O]，长度 = 5
3. 主循环：usb2i2c_popTask()
   a. 从队列获取任务
   b. I2C 写入：地址=0x48，数据=[H,E,L,L,O]
   c. DMA 传输到 I2C TX FIFO
4. I2C 硬件执行
   a. START + 0x48（写入）+ H + E + L + L + O + STOP
   b. 触发 I2C 中断
5. I2C_0_INST_IRQHandler()
   a. EVENT1_DMA_DONE：DMA 完成
   b. 停止条件：数据已传输
   c. fifoCompleteRead() 标记任务已消耗

场景 B：主机从目标读取 64 个字节

1. I2C 目标有数据，拉至 GPIO 高电平
2. GPIO 中断触发
   a. i2ctargetDataReady = true
3. 主循环检测：i2c2usb_pushTask()
   a. I2C 读取：地址=0x48，预期 64 个字节
   b. DMA 设置：I2C RX → 缓冲器
   c. 开始 I2C 读取传输
4. I2C 硬件执行
   a. START + 0x48（读取）+ [64 个字节] + STOP
   b. 所有字节均发送至 DMA 目标
   c. 触发 I2C 中断
5. I2C_0_INST_IRQHandler()
   a. EVENT2_DMA_DONE：所有数据接收完毕
   b. fifoCompleteWrite(&gI2C2USBTask)，并标记为 USB 传输准备就绪
6. 主循环：i2c2usb_popTask()
   a. 发送 64 个字节至 USB
   b. fifoCompleteRead() 标记任务已完成
7. 主机在 COM 端口接收 64 个字节

9 其他资源

- 德州仪器 (TI)，*下载 MSPM0 SDK*
- 德州仪器 (TI)，*详细了解 SysConfig*
- 德州仪器 (TI)，*MSPM0G5187*
- 德州仪器 (TI)，*MSPM0G5117*

10 商标

LaunchPad™ is a trademark of Texas Instruments.
Windows® is a registered trademark of Microsoft Corporation.
macOS® is a registered trademark of Apple Inc..
Linux® is a registered trademark of Linus Torvalds.
所有商标均为其各自所有者的财产。

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、与某特定用途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安全、安保法规或其他要求。

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。对于因您对这些资源的使用而对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，您将全额赔偿，TI 对此概不负责。

TI 提供的产品受 TI 销售条款)、TI 通用质量指南 或 ti.com 上其他适用条款或 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。 除非德州仪器 (TI) 明确将某产品指定为定制产品或客户特定产品，否则其产品均为按确定价格收入目录的标准通用器件。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

最后更新日期：2025 年 10 月