Subsystem Design

I2C 转 SPI 桥接器

1 设计说明

该子系统用作 I2C 转 SPI 桥接器。MSPM0 器件是该子系统中的 I2C 目标和 SPI 控制器。当 I2C 控制器向桥接器 I2C 目标器件发送数据时，目标器件会收集接收到的所有数据。一旦目标检测到停止 I2C 条件，桥接器就会使用桥接器 SPI 控制器将数据发送出去。然后，桥接器等待来自 SPI 外设的 SPI 数据。当桥接器 SPI 控制器完成数据读取时，桥接器会等待 I2C 控制器发送读取数据的请求。最后，桥接器通过桥接器 I2C 目标传输数据，并复位桥接器状态。如果发生错误，桥接器会通过 I2C 目标发送错误消息。

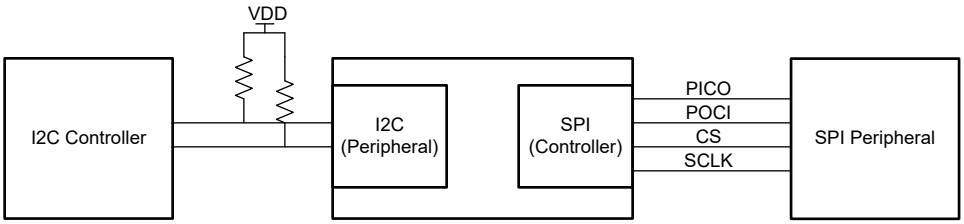

图 1-1. 系统功能方框图

2 所需外设

该子系统使用两个 MSPM0 外设：I2C 和 SPI。

表 2-1. 外设

子块功能	外设使用	注释
I2C 目标方	I2C	在代码中称为 I2C_INST。
SPI 控制器	SPI	在代码中称为 SPI_INST。默认传输速率 1MHz。

3 设计步骤

1. 子系统工程位于 MSP 子系统文件夹的 M0 SDK 中。
2. 在 SysConfig 中设置 SPI 模块。将器件置于 SPI 控制器模式，并将其余设置保留为默认值。现在，导航到“中断”配置选项卡并启用“接收”和“传输”中断。
3. 在 SysConfig 中设置 I2C 模块。将器件置于目标模式，并将其余设置保留为默认值。现在，导航到“中断”配置选项卡并启用 Start Detection、RX FIFO Trigger、TX FIFO Trigger、Stop Detection、TX FIFO Underflow、Target Arbitration Lost、RX FIFO Overflow 和 Interrupt Overflow 中断参数。
4. 将最大数据包大小定义为所需的数据包大小。

4 设计注意事项

1. 通信速度。
   a. 提高这两个接口的速度可增加数据吞吐量，减少数据冲突的可能性。
   b. 如果 I2C 速度提高，则需要根据 I2C 规范调整外部上拉电阻以允许通信。一般而言，10kΩ 适用于 100kHz 频率。较高的 I2C 总线速率需要值较低的上拉电阻。对于 400kHz 通信，请使用更接近 4.7kΩ 的电阻。
   c. 为提高桥接器利用率，有必要对该代码进行进一步优化。其他优化包括更高的器件运行速度、多个传输缓冲器或简化状态机。

备注

图 1-1 示例仅使用 100kHz (I2C) 的默认速度进行了测试。

2. 检查两个外设所使用的引脚。有些引脚需要特别注意，例如漏极开路引脚。

5 软件流程图

以下图表显示了通信工作原理的简要示意图。此处的 X 个字节表示在通信过程中，程序包所能容纳的最大字节数。

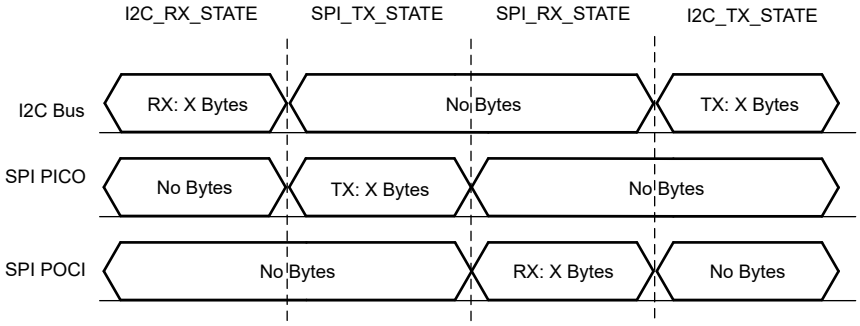

图 5-1. 简要通信图

图 5-2 展示了此示例的代码流程图，并说明了器件如何使用接收到的 I2C 数据填充数据缓冲区，然后通过 SPI 传输数据。

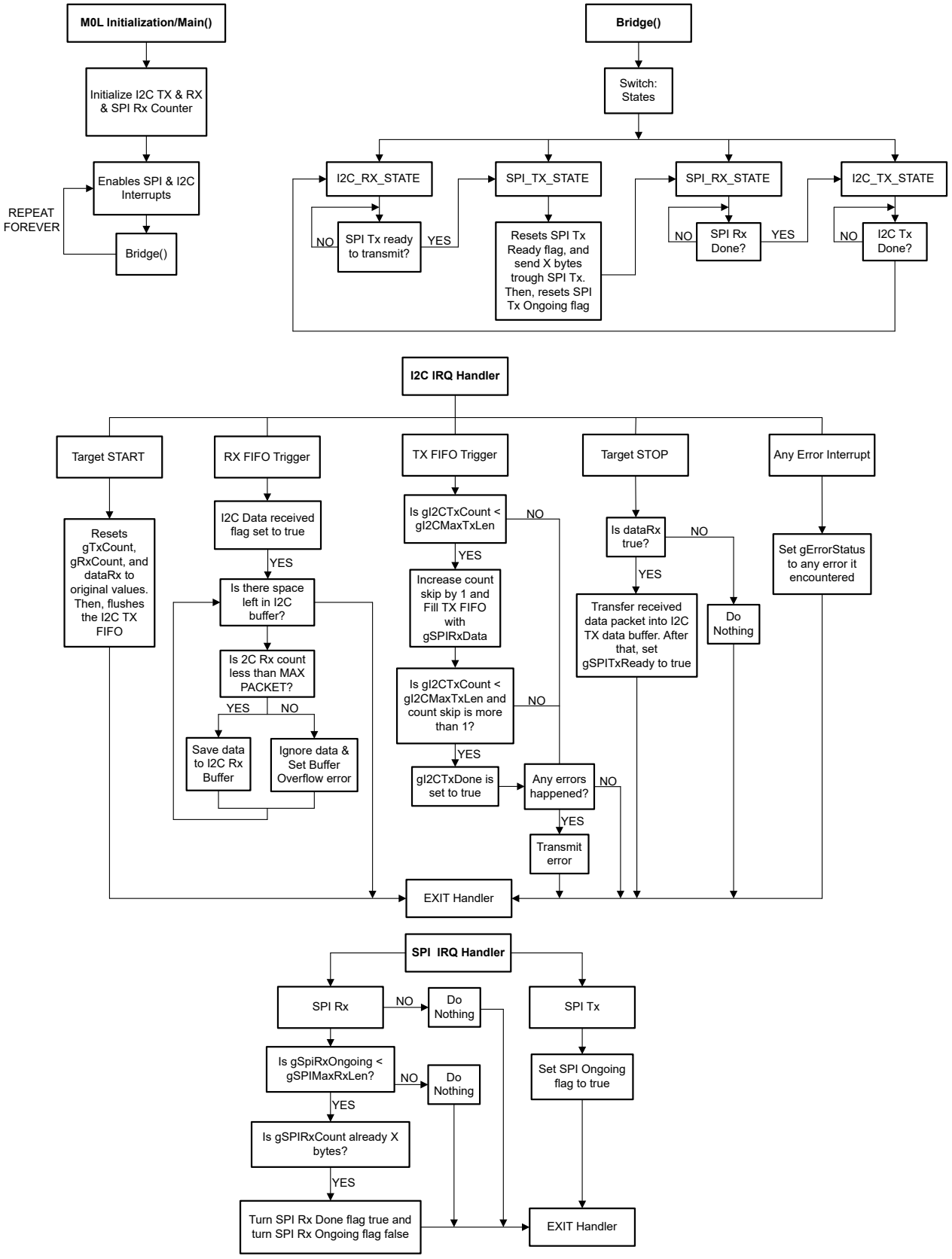

图 5-2. 应用软件流程图

6 器件配置

该应用利用 TI 系统配置工具 (SysConfig) 图形界面来生成器件外设的配置代码。使用图形界面配置器件外设可简化应用原型设计过程。

可以在 *i2c_to_spi_bridge.c* 文件的 main() 开头找到图 5-2 中所述内容的代码。

7 应用代码

缓冲器、计数器、枚举和标志的初始化如下所示。要更改 I2C 使用的特定值和 SPI 最大数据包大小，请修改文档开头的 #defines，如以下代码块所示。

```
#include "ti_msp_dl_config.h"

/* Maximum size of packet */
#define MAX_PACKET_SIZE      (4)

/* Data sent to Controller in response to Read transfer */
uint8_t gSPITxPacket[MAX_PACKET_SIZE] = {0x00};
uint8_t gI2CRxPacket[MAX_PACKET_SIZE];                              /* Data received from
Controller during a Write transfer */
uint8_t gSPIRxData[MAX_PACKET_SIZE];

/* Counters for I2C RX & TX */
uint32_t gI2CTxCount;
uint32_t gI2CRxCount;

/* Counters for SPI RX & TX */
uint32_t gSPIRxCount;

/* Variable used to skip false I2C Trigger when first sending a message */
uint32_t count_skip = 0;

enum error_codes{
    NO_ERROR = 0,
    DATA_BUFFER_OVERFLOW,
    I2C_TARGET_TXFIFO_UNDERFLOW,
    I2C_TARGET_RXFIFO_OVERFLOW,
    I2C_TARGET_ARBITRATION_LOST,
    I2C_INTERRUPT_OVERFLOW
};

/* Indicates status of Bridge */
enum BridgeStates {
    I2C_RX_STATE = 0,
    SPI_TX_STATE,
    SPI_RX_STATE,
    I2C_TX_STATE
} gBridgeStates;

uint8_t gErrorStatus = NO_ERROR;

/* Flags */
bool gSpiTxReady = false;                                                   /* Flag to start SPI
transfer */
bool gSpiTxOngoing = false;                                                 /* Flag to indicate SPI
transfer Ongoing*/
bool gSpiRxDone = false;                                                    /* Flag to indicate SPI
data has been received */
bool gSpiRxOngoing = false;                                                 /* Flag to indicate SPI
data is being receive */
bool gI2cTxDone = false;                                                    /* Flag to start SPI
transfer */
```

应用代码的主体相对较短。首先，器件和外设被初始化。然后，在开始传输之前处于空闲状态的 SPI TX 发生延迟，同时计数器和标志值也被初始化。接下来，会启用中断和事件，并运行包含桥接函数的主循环。

```
int main(void)
{
    SYSCFG_DL_init();

    /* Initialize variables to send data inside TX ISR */
```

```
    gI2CTxCount = 0;

    /* Initialize variables to receive data inside RX ISR */
    gI2CRxCount = 0;

    /* Initialize variables to receive data inside RX ISR */
    gSPIRxCount = 0;

    // Setting flags to default values
    gSpiTxReady = false;
    gSpiRxDone = false;

    /* Enabling Interrupts on I2C & SPI Modules */
    NVIC_EnableIRQ(I2C_INST_INT_IRQN);
    NVIC_EnableIRQ(SPI_INST_INT_IRQN);
    while (1) {
        bridge();
    }
}
```

桥接器有四种状态。第一种状态侧重于桥接器 I2C 目标器件接收数据。第二种状态在通过桥接器 SPI 控制器发送 I2C 目标器件接收到的数据时发生。然后，出现第三种状态，SPI 控制器等待来自外设的数据；最终进入第四种状态，在这种状态下，桥接 I2C 目标器件等待发送请求并从 SPI 外设发送数据。

```
void bridge(){
    switch (gBridgeStates) {
        case I2C_RX_STATE:
            if (gSpiTxReady){
                gBridgeStates = SPI_TX_STATE;
            }
            else {
                break;
            }
        case SPI_TX_STATE:
            gSpiTxReady = false;
            for(int i = 0; i < gI2CRxCount; i++){
                /* Transmit data out via SPI and wait until transfer is complete */
                DL_SPI_transmitDataBlocking8(SPI_INST, gSPITxPacket[i]);
            }
            gSpiTxOngoing = false;
            gBridgeStates = SPI_RX_STATE;
            break;
        case SPI_RX_STATE:
            if(gSpiRxDone){
                gSPIRxCount = 0;
                gBridgeStates = I2C_TX_STATE;
            }
            break;
        case I2C_TX_STATE:
            if(gI2cTxDone){
                gI2cTxDone = false;
                gBridgeStates = I2C_RX_STATE;
            }
            break;
        default:
            break;
    }
}
```

此代码的下一段是 I2C IRQ 处理程序。此代码用于处理桥接 I2C 目标中断情况。当挂起中断是检测到的 I2C 启动条件时，计数器变量和标志将被设置为默认值。当挂起中断表示 I2C RX FIFO 有数据可用时，如果还有空间，接收到的值将保存在 I2C RX 缓冲区中。如果没有更多的空间，接收到的值会被忽略。当挂起中断是 I2C TX FIFO 触发信号时，跳过计数器会增加并通过 I2C 发送数据，直至达到最大长度。如果 I2C TX 计数达到最大数据包计数且计数跳过计数器大于 1（假设来自 I2C 目标的第一个发送数据是 false 触发并且已经发生），则 I2C Tx “Done”标志变为 true。然后，如果桥接器检测到任何错误，则错误代码将通过桥接器 I2C 目标进行传输。

当挂起中断是 I2C 停止条件时，器件会检查是否接收到数据；如果为 true（I2C 数据接收标志为 true），接收到的数据缓冲器将存储在发送数据缓冲器中，SPI TX “ready”标志设置为 true。如果未收到任何 I2C 数据，器件不

会发送任何内容。该 ISR 还可处理 I2C 错误中断，方法是向错误状态变量（TXFIFO Underflow、RXFIFO Overflow、Target Arbitration Lost 和 Interrupt Overflow）分配适当的错误代码。

```
void I2C_INST_IRQHandler(void)
{
    static bool gI2CRxDone = false;     // Flag that indicates I2C data received
    switch (DL_I2C_getPendingInterrupt(I2C_INST)) {
        case DL_I2C_IIDX_TARGET_START:
            /* Initialize (resets) RX or TX after Start condition is received */
            gI2CTxCount = 0;
            gI2CRxCount = 0;
            gI2CRxDone   = false;
            /* Flush TX FIFO to refill it */
            DL_I2C_flushTargetTXFIFO(I2C_INST);
            break;
        case DL_I2C_IIDX_TARGET_RXFIFO_TRIGGER:
            /* Store received data in buffer */
            gI2CRxDone = true;
            while (DL_I2C_isTargetRXFIFOEmpty(I2C_INST) != true) {
                if(gI2CRxCount < MAX_PACKET_SIZE){
                    gI2CRxPacket[gI2CRxCount++] = DL_I2C_receiveTargetData(I2C_INST);
                }else{
                    /* Prevent overflow and just ignore data */
                    DL_I2C_receiveTargetData(I2C_INST);
                    gErrorStatus = DATA_BUFFER_OVERFLOW;
                }
            }
            break;
        case DL_I2C_IIDX_TARGET_TXFIFO_TRIGGER:                     // Restarts the flag
                /* Fill TX FIFO if there are more bytes to send */
                if (gI2CTxCount < MAX_PACKET_SIZE) {
                    count_skip += 1;
                    gI2CTxCount += DL_I2C_fillTargetTXFIFO(I2C_INST, &gSPIRxData[gI2CTxCount],
(MAX_PACKET_SIZE - gI2CTxCount));
                    if(gI2CTxCount >= MAX_PACKET_SIZE && count_skip > 1){
                        gI2cTxDone = true;
                    }
                }
                if(gErrorStatus != NO_ERROR)
                {
                    /* Fill FIFO with error status after sending latest received
                     * byte */
                    while (DL_I2C_transmitTargetDataCheck(I2C_INST, gErrorStatus) != false);
                }
            break;
        case DL_I2C_IIDX_TARGET_STOP:
            /* If data was received, store it in SPI TX buffer */
            if (gI2CRxDone == true) {
                for (uint16_t i = 0; (i < gI2CRxCount) && (i < MAX_PACKET_SIZE); i++) {
                    gSPITxPacket[i] = gI2CRxPacket[i];
                    DL_I2C_flushTargetTXFIFO(I2C_INST);
                }
                gI2CRxDone = false;
                /* Set flag to indicate data ready for SPI TX */
                gSpiTxReady = true;
            }
            break;
        case DL_I2C_IIDX_TARGET_TXFIFO_UNDERFLOW:
            gErrorStatus = I2C_TARGET_TXFIFO_UNDERFLOW;
            break;
        case DL_I2C_IIDX_TARGET_RXFIFO_OVERFLOW:
            gErrorStatus = I2C_TARGET_RXFIFO_OVERFLOW;
            break;
        case DL_I2C_IIDX_TARGET_ARBITRATION_LOST:
            gErrorStatus = I2C_TARGET_ARBITRATION_LOST;
            break;
        case DL_I2C_IIDX_INTERRUPT_OVERFLOW:
            gErrorStatus = I2C_INTERRUPT_OVERFLOW;
            break;
        default:
            break;
    }
}
```

此子系统代码中的最后一段代码是 SPI IRQ 处理程序。SPI IRQ 处理程序会保存接收到的数据，并在 SPI 发送数据时设置“ongoing”传输标志。当 SPI RX 中断挂起时，器件将接收到的数据保存到 SPI RX 缓冲区，将 SPI RX “Ongoing”标志变为 true，并在 SPI RX 计数器达到最大长度时设置 SPI RX “Done”标志。

```
void SPI_INST_IRQHandler(void)
{
    switch (DL_SPI_getPendingInterrupt(SPI_INST)) {
        case DL_SPI_IIDX_RX:
            /* Read RX FIFO */
            if(gSPIRxCount < MAX_PACKET_SIZE){
                gSpiRxOngoing = true;
                gSPIRxData[gSPIRxCount++] = DL_SPI_receiveData8(SPI_INST);
                if(gSPIRxCount >= MAX_PACKET_SIZE){
                    gSpiRxDone = true;
                    gSpiRxOngoing = false;
                }
            }
            break;
        case DL_SPI_IIDX_TX:
            gSpiTxOngoing = true;
            break;
        default:
            break;
    }
}
```

8 移植指南

首先，打开工程 SYSCONFIG 文件，然后点击 SYSCONFIG 窗口右上角的“显示器件视图”图标。

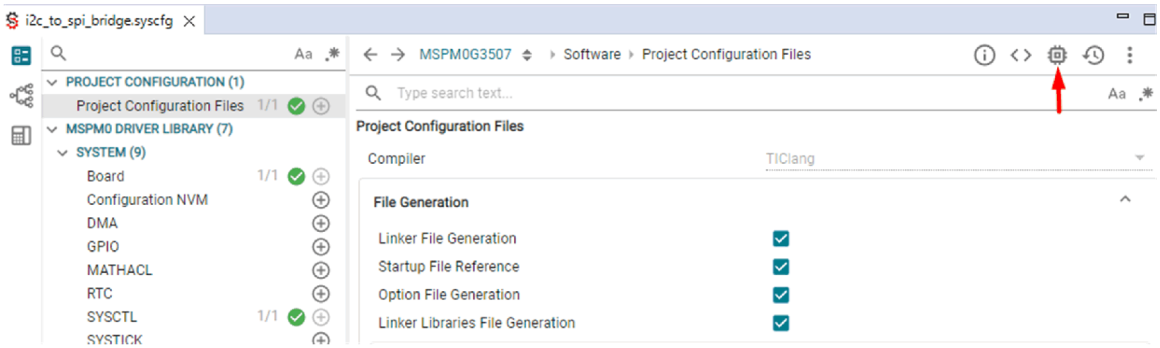

图 8-1. “显示器件视图”图标位置

点击此图标后，将显示工程目标器件包。点击“开关”按钮进行更改。

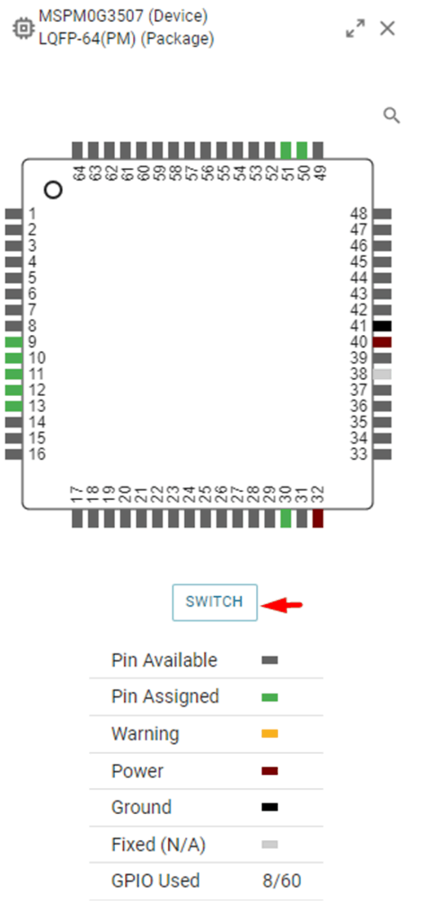

图 8-2. “开关”按钮位置

接下来，“迁移设置”随即打开。在此处，您可以为电路板（如果用户使用）、器件和封装选择新的值。在 SYSCONFIG 中，将芯片组切换为所需的器件。请确保在设置新器件后选择正确的 MCU 型号和封装，点击 CONFIRM 按钮。

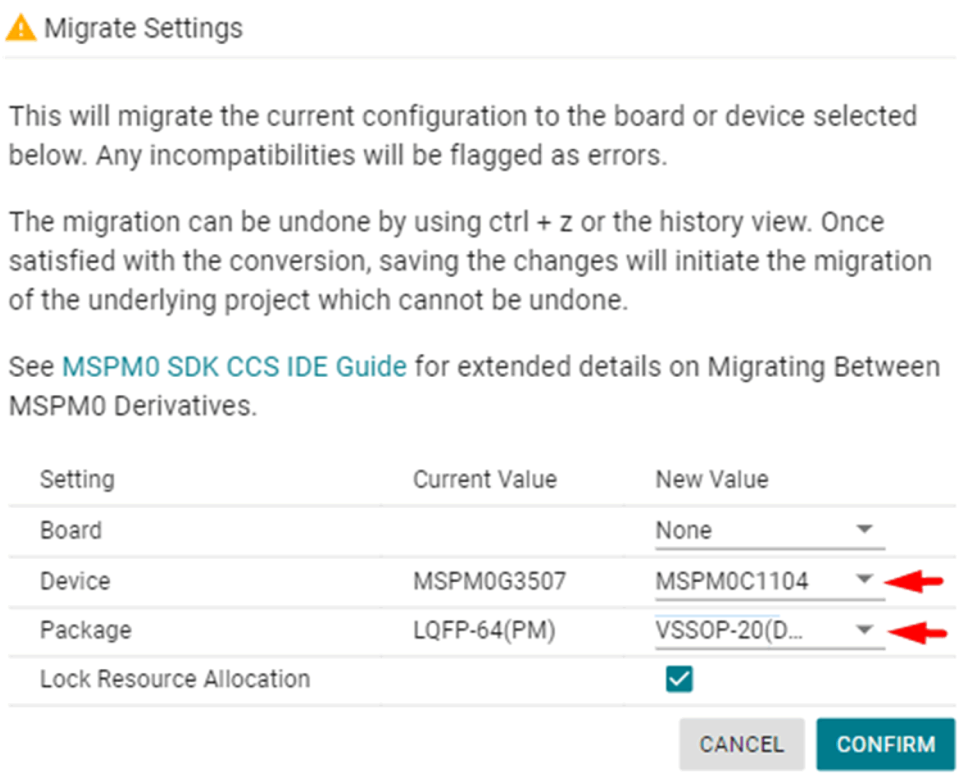

图 8-3. “迁移设置”窗口

执行此操作后，SYSCONFIG 会自动调整新器件的引脚和外设（除非引脚被锁定）。在新器件中，可能会出现因前一个器件的值无效而引发的错误，例如引脚值不同或缺乏特性。错误以红色 X 符号显示。

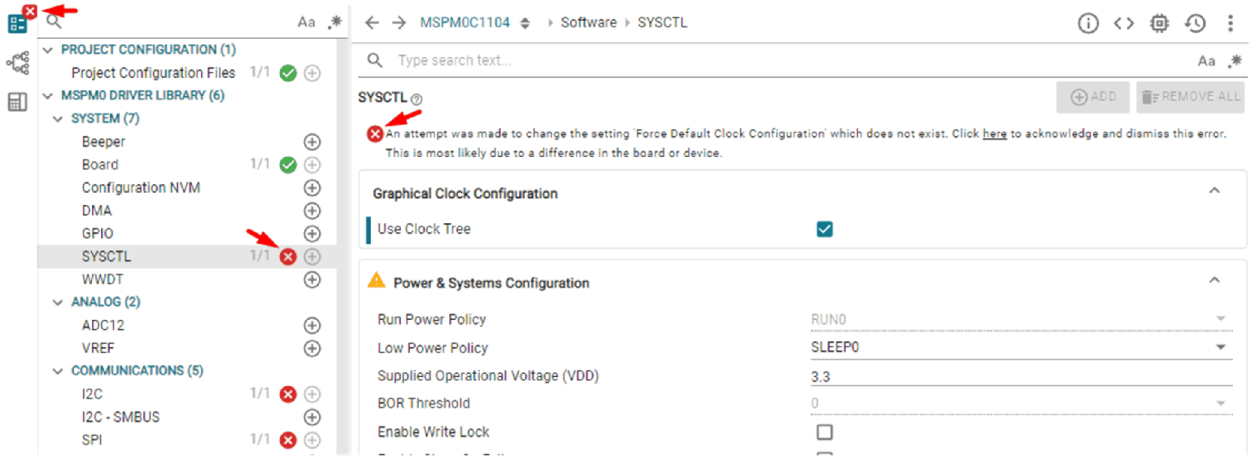

图 8-4. “切换器件后发生错误”示例

要显示所有错误，请点击 SYSCONFIG 窗口右上角的“显示问题”图标。阅读这些错误并按照有关如何修复错误的说明进行操作，从而实现正确的工程编译。

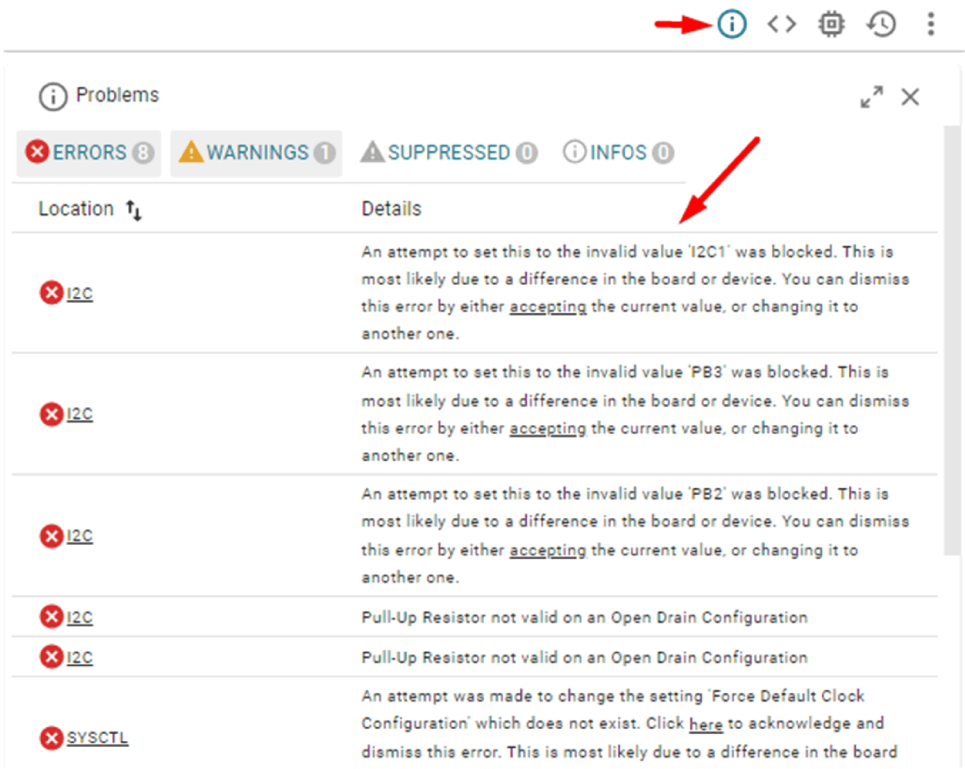

图 8-5. “显示问题”图标位置和内容示例

Sysconfig，检查您要使用的外设的引脚，确保不与之前的 MCU 发生冲突。如有必要，将 sysconfig 更改为要在运行工程的新器件中使用的引脚。最后，在新器件中构建或编译工程。如果正确完成，您可以在 Debug 文件夹中看到以下消息以及 ti_msp_dl_config.c 和 .h 文件。

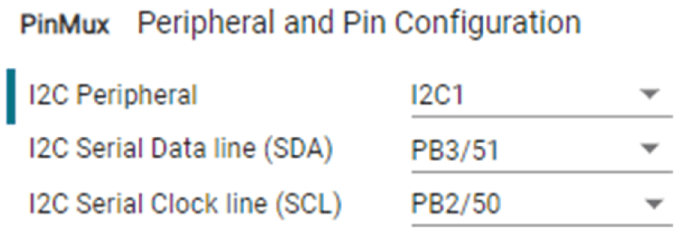

图 8-6. 引脚外设配置示例

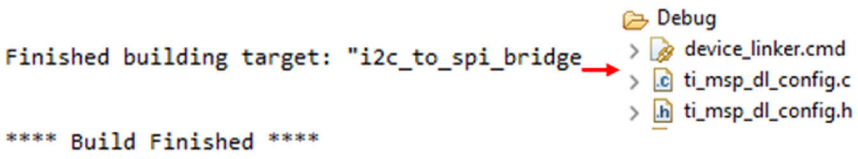

图 8-7. 工程构建和文件生成示例

9 修订历史记录

注：以前版本的页码可能与当前版本的页码不同

Changes from Revision * (February 2025) to Revision A (August 2025)	Page
• 删除了“兼容器件”部分	1

商标

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、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