Subsystem Design

SPI 转 I2C 桥接器

1 设计说明

该子系统用作 SPI 转 I2C 桥接器。在该子系统中，MSPM0 器件是 SPI 外设和 I2C 控制器。当 SPI 控制器向桥接器 SPI 外设传输数据时，外设会收集接收到的所有数据。一旦外设达到预期的最大消息数，外设就会使用 I2C 控制器传输数据。该器件会发送 I2C 传输请求并等待 I2C 目标器件提供 I2C 数据。I2C 控制器完成数据读取后，桥接器会等待 SPI 控制器发送从桥接器读取数据的请求。最后，桥接器通过其 SPI 外设传输从 I2C 控制器接收到的数据。

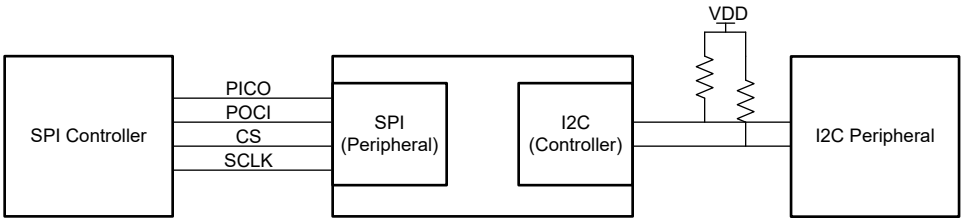

图 1-1. 系统功能方框图

2 所需外设

两个 MSPM0 外设用于该子系统：SPI 和 I2C。

表 2-1. 外设

子块功能	外设使用	注释
SPI 外设	SPI	在代码中称为 SPI_INST。
I2C 控制器	I2C	在代码中称为 I2C_INST。默认传输速率 100kHz。

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 I2C 模块。将器件置于控制器模式，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 TX Done 和 RX_FIFO_TRIGGER 中断。
3. 在 SysConfig 中设置 SPI 模块。将器件置于 SPI 外设模式，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。
4. 将最大数据包大小定义为所需的数据包大小。

English Document: SLAAES5
Copyright © 2025 Texas Instruments Incorporated

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，例如开漏引脚。

English Document: SLAAES5
Copyright © 2025 Texas Instruments Incorporated

5 软件流程图

图 5-1 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 SPI 数据填充数据缓冲区，然后通过 I2C 传输数据。

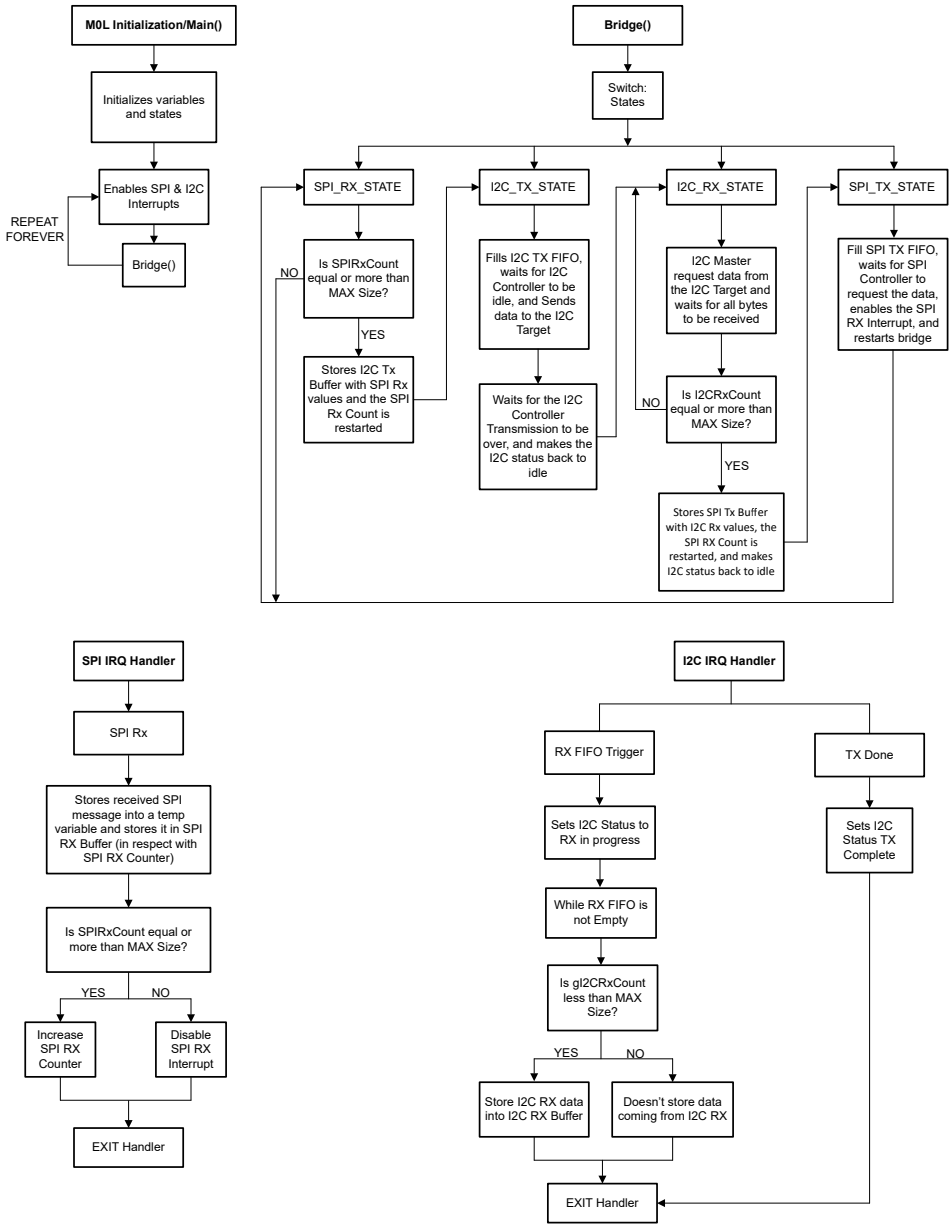

图 5-1. 应用软件流程图

图 5-1 展示了通信工作原理的简图。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。

English Document: SLAAES5
Copyright © 2025 Texas Instruments Incorporated

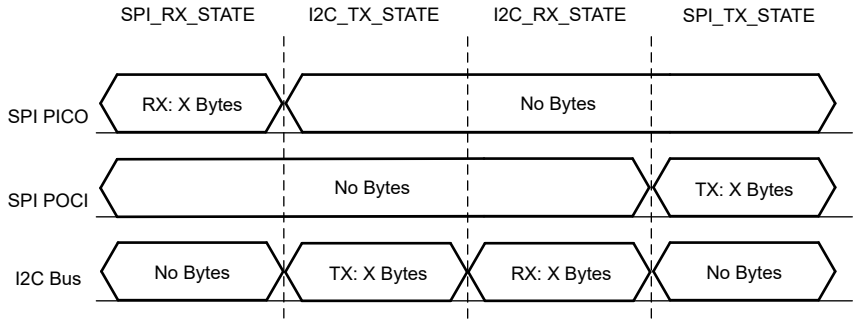

图 5-2. 简要通信图

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 *spi_to_i2c_bridge.c* 文件的 main() 开头找到 图 5-1 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 SPI 使用的特定值和最大 I2C 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"
/* Delay for 5ms to ensure SPI TX is idle before starting transmission */
#define SPI_TX_DELAY (160000)

/*Define max packet sizes*/
#define MAX_PACKET_SIZE 4

/*SPI Buffers & Variables*/
uint8_t gSPITxData[MAX_PACKET_SIZE];
uint8_t gSPIRxData[MAX_PACKET_SIZE];
volatile uint8_t gSPIRxCount = 0;                    // Variable to track # of bytes SPI Received
/*I2C Controller Buffers &  Variable*/
uint8_t gI2CTxData[MAX_PACKET_SIZE];
uint8_t gI2CRxData[MAX_PACKET_SIZE];
volatile uint8_t gI2CAddress = 0x48;                 // Target Address to communicate to
volatile uint8_t gI2CTxCount = 0;                    // Variable to track # of bytes I2C
Transmitted
volatile uint8_t gI2CRxCount = 0;                    // Variable to track # of bytes I2C Received
volatile uint8_t rxTemp = 0;
/* Indicates status of Bridge */
enum BridgeStates {
    SPI_RX_STATE = 0,
    I2C_TX_STATE,
    I2C_RX_STATE,
    SPI_TX_STATE
} gBridgeStates;
/* Indicates status of I2C Controller */
enum I2cControllerStatus {
    I2C_C_STATUS_IDLE = 0,
    I2C_C_STATUS_TX_COMPLETE,
    I2C_STATUS_RX_STARTED,
    I2C_C_STATUS_RX_INPROGRESS,
    I2C_STATUS_RX_COMPLETE
} gI2cControllerStatus;

void bridge(void);
```

应用代码的主体相对较短。首先，器件外设和中断被初始化。然后，在开始传输之前处于空闲状态的 SPI TX 发生延迟，同时状态和标志值也被初始化。接下来，具有桥接器功能的主循环将会运行。

```
int main(void)
{
```


English Document: SLAAES5
Copyright © 2025 Texas Instruments Incorporated

```
    SYSCFG_DL_init();
    /* Activate Interrupts */
    NVIC_ClearPendingIRQ(SPI_INST_INT_IRQN);
    NVIC_EnableIRQ(SPI_INST_INT_IRQN);
    NVIC_EnableIRQ(I2C_INST_INT_IRQN);

    /* Optional delay to ensure SPI TX is idle before starting transmission */
    delay_cycles(SPI_TX_DELAY);

    /*Initial states*/
    gBridgeStates = SPI_RX_STATE;
    gI2cControllerStatus = I2C_C_STATUS_IDLE;

    /* Start bridge */
    while (1) {
        bridge();   // Runs bridge
    }
}
```

桥接器有四种状态。第一个状态侧重于将数据从 SPI RX 缓冲器传输到 I2C TX 缓冲器，同时在 SPI 外设接收到最大封装大小后清除 SPI RX 缓冲器。第二种状态则是将数据从 I2C TX 缓冲器传输到 I2C 目标器件。然后，在第三种状态下，I2C 控制器会向 I2C 目标器件发送请求数据信号，并在重新收集数据后将 I2C RX 缓冲器传输到 SPI TX 缓冲器，同时清除 I2C RX 缓冲器。最后的状态则是以 SPI TX 缓冲器的内容填充 SPI 目标器件，等待 FIFO 变空（控制器请求获取数据），使 SPI RX 再次中断，然后重新启动桥接器。

```
void bridge(){
    uint8_t i=0; uint8_t j=0;uint8_t k=0;                                   // Setting counter
variables
    switch (gBridgeStates) {
        case SPI_RX_STATE:
            if (gSPIRxCount >= MAX_PACKET_SIZE){
                // Storing data from SPI Buffer to message that is addressed to Bridge A
                for(k = 0; k < MAX_PACKET_SIZE; k++){
                    gI2CTxData[k] = gSPIRxData[k];
                    gSPIRxData[k] = 0;
                }
                // Resetting gSPIRxCount variable
                gSPIRxCount = 0;
                gBridgeStates = I2C_TX_STATE;
            }
            else {
                break;
            }
        case I2C_TX_STATE:
            // Sending the I2C WRITE message to the 9724
            gI2CTxCount = DL_I2C_fillControllerTXFIFO(I2C_INST, &gI2CTxData[0], MAX_PACKET_SIZE);

            /* Send the packet to the target. This function will send Start + Stop automatically. */
            while (!(DL_I2C_getControllerStatus(I2C_INST) & DL_I2C_CONTROLLER_STATUS_IDLE));
            DL_I2C_startControllerTransfer(I2C_INST, gI2CAddress, DL_I2C_CONTROLLER_DIRECTION_TX,
MAX_PACKET_SIZE);

            /* Wait until the Controller sends all bytes AKA the I2C_C_STATUS_TX_COMPLETE to be
true */
            while (gI2cControllerStatus != I2C_C_STATUS_TX_COMPLETE) {
                __WFE();
            }
            while (DL_I2C_getControllerStatus(I2C_INST) & DL_I2C_CONTROLLER_STATUS_BUSY_BUS);

            gI2cControllerStatus = I2C_C_STATUS_IDLE;
            gBridgeStates = I2C_RX_STATE;                                   // Move to next
Bridge stage
            break;
        case I2C_RX_STATE:
            /* Add delay between transfers */
            delay_cycles(1000);

            /* Send a read request to Target */
            gI2cControllerStatus = I2C_STATUS_RX_STARTED;
            DL_I2C_startControllerTransfer(I2C_INST, gI2CAddress, DL_I2C_CONTROLLER_DIRECTION_RX,
MAX_PACKET_SIZE);

            /* Wait for all bytes to be received in interrupt */
            while (gI2cControllerStatus != I2C_STATUS_RX_COMPLETE) {
                __WFE();
```


English Document: SLAAES5
Copyright © 2025 Texas Instruments Incorporated

```
            }
            while (DL_I2C_getControllerStatus(I2C_INST) &
                   DL_I2C_CONTROLLER_STATUS_BUSY_BUS);

            /* Waiting for I2C Rx buffer interrupt to happen AKA When expected package size
               is received from I2C Target */
            if(gI2CRxCount >= MAX_PACKET_SIZE){
                // Extract the received bytes from the I2C read and store them in SPI buffer #2 (Tx)
                for(j = 0; j < MAX_PACKET_SIZE; j++){
                    gSPITxData[j] = gI2CRxData[j];
                    gI2CRxData[j] = 0;
                }
                // Resetting gI2CRxCount variable
                gI2CRxCount = 0;
                gI2cControllerStatus = I2C_C_STATUS_IDLE;
                gBridgeStates = SPI_TX_STATE;                                    // Move to next
Bridge stage
            }
            break;
        case SPI_TX_STATE:
            DL_SPI_fillTXFIFO8(SPI_INST, &gSPITxData[0], MAX_PACKET_SIZE);
            while(!DL_SPI_isTXFIFOEmpty(SPI_INST));

            DL_SPI_enableInterrupt(SPI_INST, DL_SPI_INTERRUPT_RX);
            gBridgeStates = SPI_RX_STATE;
            break;
    }
}
```

此示例的下一部分是 SPI IRQ 处理程序。本示例仅使用了一个中断：SPI RX。激活后，数据先存储在时间变量中，然后再存储在 SPI RX FIFO 缓冲器中。然后，如果 SPI RX 计数器小于最大封装大小，则 SPI RX 计数器会增加；否则，SPI RX FIFO 中断会被禁用。

```
void SPI_INST_IRQHandler(void)
{
    switch (DL_SPI_getPendingInterrupt(SPI_INST)) {
        case DL_SPI_IIDX_RX:
            rxTemp = DL_SPI_receiveDataBlocking8(SPI_INST);
            gSPIRxData[gSPIRxCount] = rxTemp;
            if (gSPIRxCount >= MAX_PACKET_SIZE){
                DL_SPI_disableInterrupt(SPI_INST, DL_SPI_INTERRUPT_RX);
            }else {
                gSPIRxCount++;
            }
            break;
        default:
            break;
    }
}
```

本示例中的最后一段代码是 I2C IRQ 处理程序。该示例中的两个中断分别是控制器 TX Done 和控制器 RX FIFO Trigger。触发 TX Done 后，I2C 控制器状态会更新为“TX 已完成”。当触发 RX FIFO Trigger 时，I2C 控制器状态会更新为“RX 进行中”；I2C RX 缓冲器会接收存储在 I2C RX FIFO 中的所有消息，并将 I2C 控制器状态设置为“I2C RX 完成”。

```
void I2C_INST_IRQHandler(void)
{
    switch (DL_I2C_getPendingInterrupt(I2C_INST)) {
        case DL_I2C_IIDX_CONTROLLER_TX_DONE:
            gI2cControllerStatus = I2C_C_STATUS_TX_COMPLETE;
            break;
        case DL_I2C_IIDX_CONTROLLER_RXFIFO_TRIGGER:
            gI2cControllerStatus = I2C_C_STATUS_RX_INPROGRESS;
            /* Receive all bytes from target */
            while (DL_I2C_isControllerRXFIFOEmpty(I2C_INST) != true){
                if(gI2CRxCount < MAX_PACKET_SIZE) {
                    gI2CRxData[gI2CRxCount++] = DL_I2C_receiveControllerData(I2C_INST);
                }else{
                    DL_I2C_receiveControllerData(I2C_INST);
                }
            }
            gI2cControllerStatus = I2C_STATUS_RX_COMPLETE;
            break;
```


English Document: SLAAES5
Copyright © 2025 Texas Instruments Incorporated

```
        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。

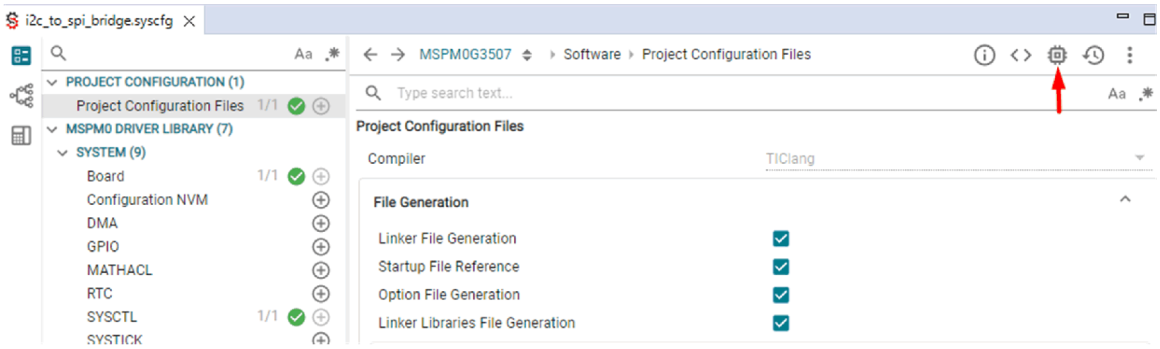

图 8-1. “显示设备视图”图标位置

点击此图标后，当前工程目标器件包将会显示。点击“切换”按钮以进行更改。

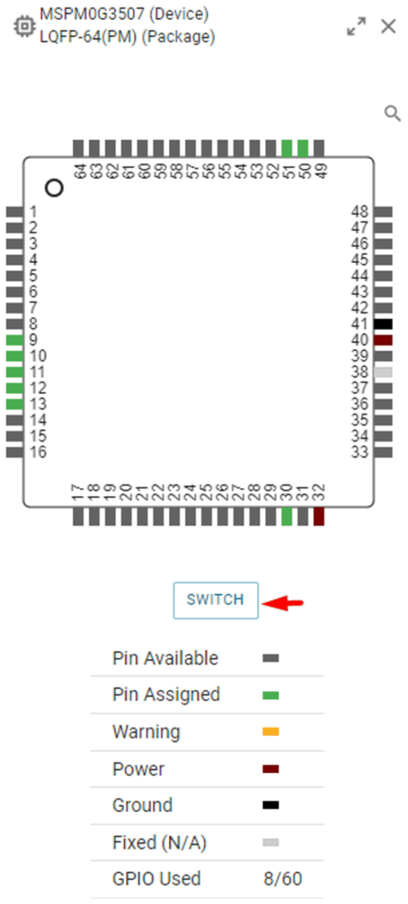

图 8-2. SWITCH 按钮位置

English Document: SLAAES5
Copyright © 2025 Texas Instruments Incorporated

接下来，“迁移设置”将会打开。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。确保选择正确的 MCU 型号和封装。完成新器件的设置后，点击“确认”按钮。

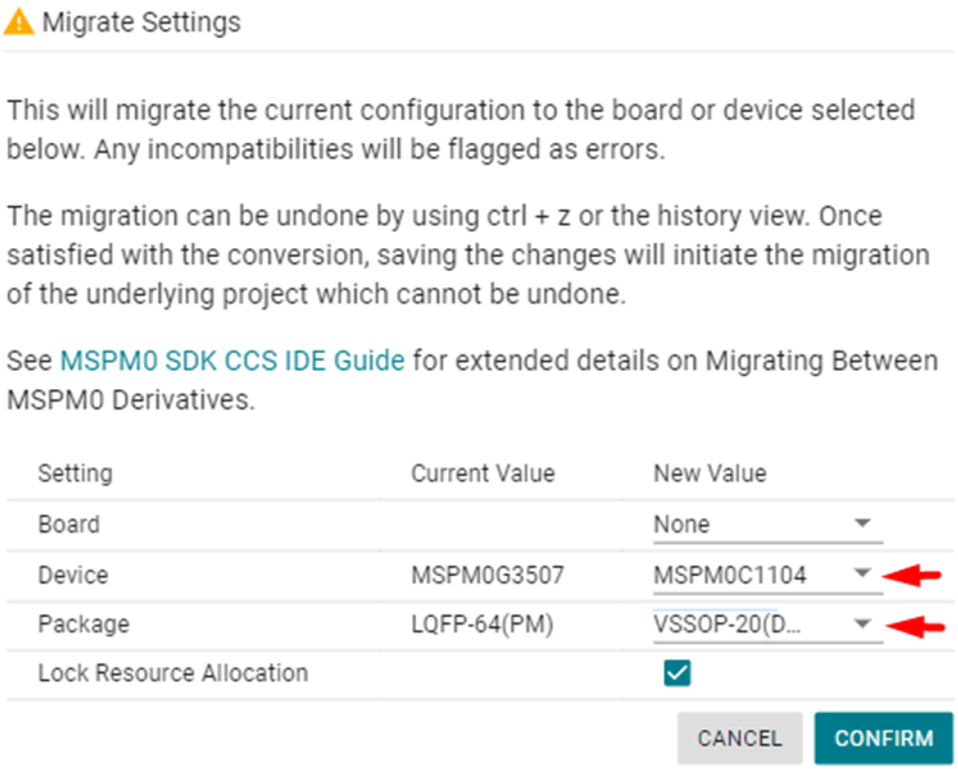

图 8-3. “迁移设置”窗口

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。

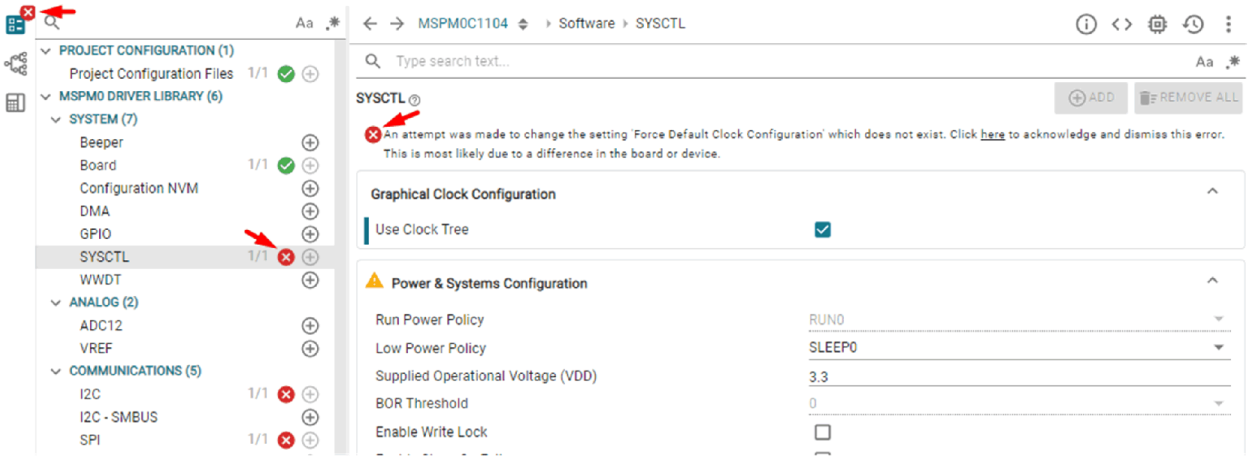

图 8-4. “切换器件后发生错误”示例

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。

English Document: SLAAES5
Copyright © 2025 Texas Instruments Incorporated

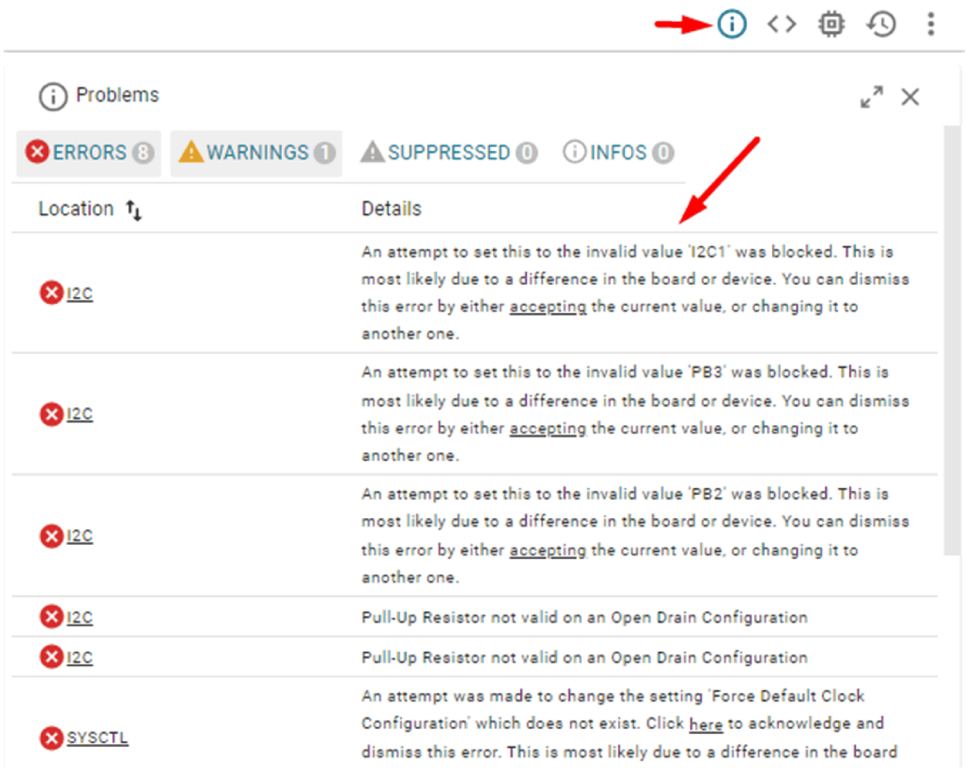

图 8-5. “显示问题”图标位置和内容示例

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，更换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。

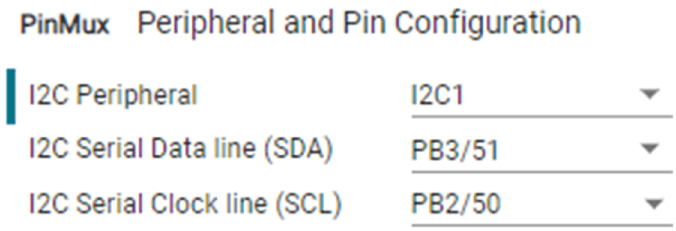

图 8-6. 引脚外设配置示例

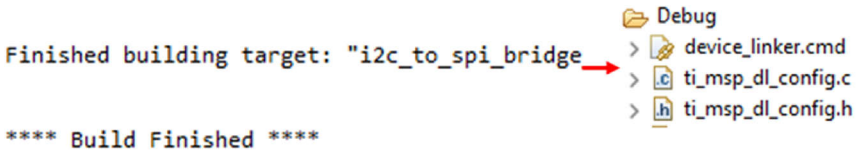

图 8-7. 工程构建和文件生成示例

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

English Document: SLAAES5
Copyright © 2025 Texas Instruments Incorporated

商标

所有商标均为其各自所有者的财产。

English Document: SLAAES5
Copyright © 2025 Texas Instruments Incorporated

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
版权所有 © 2025，德州仪器 (TI) 公司