Subsystem Design

I2C to SPI Bridge

1 Design Description

This subsystem serves as an I2C-to-SPI bridge. The MSPM0 device is the I2C target and the SPI Controller in this subsystem. When an I2C Controller transmits to the Bridge I2C target, the target collects all of the received data. Once the target detects a stop I2C condition, the bridge sends the data through the bridge SPI controller. The bridge then waits for SPI data from an SPI peripheral. When the Bridge SPI controller finishes reading the data, the bridge waits for an I2C controller to send a request to read the data. Finally, the bridge transmits the data through the bridge I2C target and resets the bridge state. If an error occurs, the bridge sends the error through the I2C target.

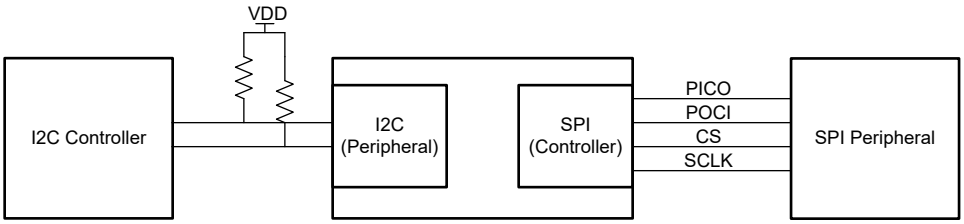

Figure 1-1. System Functional Block Diagram

2 Required Peripherals

Two MSPM0 peripherals are used for this subsystem: the I2C and SPI.

Table 2-1. Peripherals

Sub-Block Functionality	Peripheral Use	Notes
I2C Target	I2C	Called I2C_INST in code.
SPI Controller	SPI	Called SPI_INST in code. Default 1MHz transmission rate.

3 Design Steps

1. The subsystem project can be found in the M0 SDK under MSP Subsystems folder.
2. Set up the SPI module in SysConfig. Put the device in SPI Controller mode and leave the rest of the settings on default. Now, navigate to the Interrupt configuration tab and enable the Receive and Transmit interrupts.
3. Set up the I2C module in SysConfig. Set the device in Target Mode and leave the rest of the settings by default. Now navigate to the Interrupt configuration tab and enable the Start Detection, RX FIFO Trigger, TX FIFO Trigger, Stop Detection, TX FIFO Underflow, Target Arbitration Lost, RX FIFO Overflow, and Interrupt Overflow interrupts.
4. Define the maximum packet size to the desired package size.

4 Design Considerations

1. Communication speed.
   a. Increasing both interface speeds increases data throughput and decreases chances of data collisions.
   b. Adjusting external pull-up resistors according to I2C specifications is necessary to allow for communication if I2C speeds are increased. As a general guideline, 10kΩ is appropriate for 100kHz. Higher I2C bus rates require lower valued pullup resistors. For 400kHz communications, use resistors closer to 4.7kΩ.
   c. Additional optimization of this code can be necessary to meet increased bridge utilization. Additional optimizations include higher device operating speeds, multiple transfer buffers, or state machine simplification.

Note

Figure 1-1 example was only tested with default speed of 100kHz (I2C) speeds.

2. Check the pins being used for both peripherals. There are some pins who require special considerations like being open drained.

5 Software Flowchart

The picture below shows a high-level diagram of how the communication works. X bytes here represent the Maximum package of bytes found in the communication process.

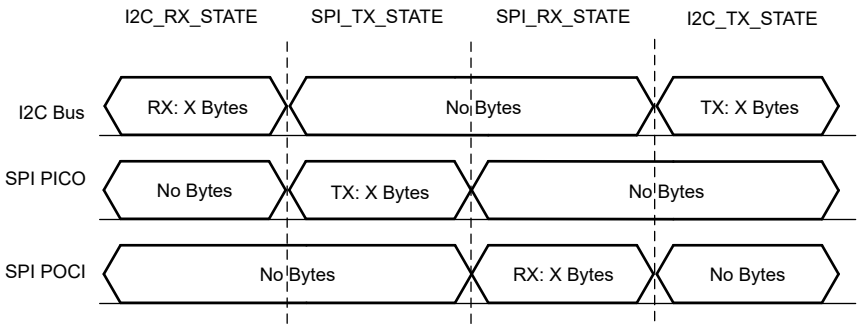

Figure 5-1. High Level Communication Diagram

Figure 5-2 shows the code flow diagram for this example and explains how the device fills the data buffers with received I2C data, then transfers the data out through SPI.

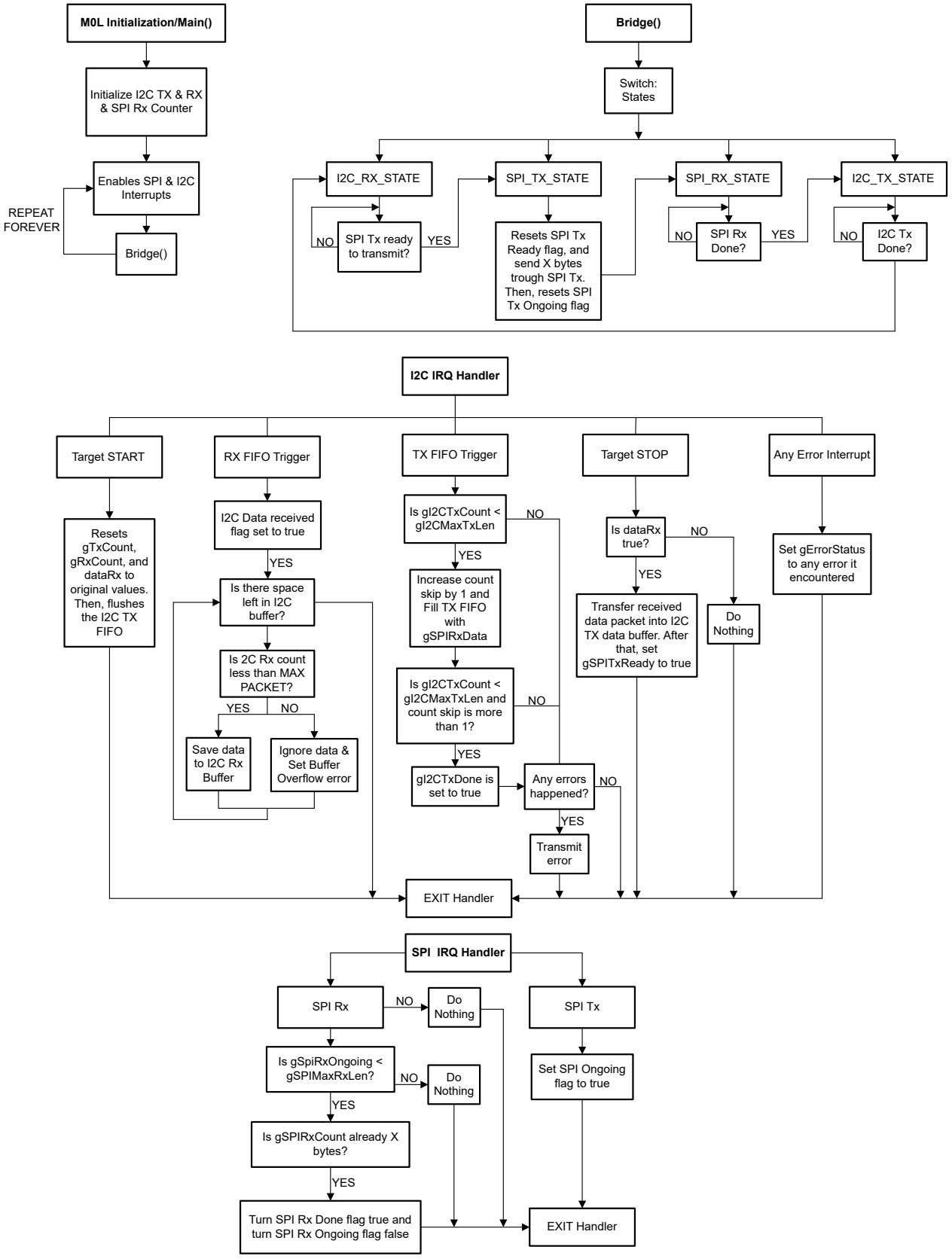

Figure 5-2. Application Software Flowchart

6 Device Configuration

This application makes use of the TI System Configuration Tool (SysConfig) graphical interface to generate the configuration code of the device peripherals. Using a graphical interface to configure the device peripherals streamlines the application prototyping process.

The code for what is described in Figure 5-2 is found in the beginning of main() in the *i2c_to_spi_bridge.c* file.

7 Application Code

The initialization of the buffers, counters, enum, and flag are shown here. To change the specific values used by the I2C and SPI maximum packet size, modify the #defines in the beginning of the document, as demonstrated in the following code block.

```
#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 */
```

The main body of the application code is relatively short. First, the device and peripherals get initialized. Then, a delay occurs for the SPI TX, which is idle before starting transmission, while counters and flag values are

initialized. Following up, the interrupts and events are enabled, and the main loop, which contains the bridge function, runs.

```
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();
    }
}
```

The bridge has four states. The first state focuses on the Bridge I2C Target receiving data. The second state happens when the I2C Target received data is sent through the Bridge SPI Controller. Then, the third state happens, where the SPI Controller waits for data from the Peripheral, to finally go into the fourth state, where the Bridge I2C Target waits for a transmit request and sends the data from the SPI Peripheral.

```
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;
    }
}
```

The next piece of this code is the I2C IRQ Handler. This code is used to handle the Bridge I2C Target interrupts. When the pending interrupt is an I2C Start condition detected, the counter variables and flags get set to default values. When the pending interrupt indicates that the I2C RX FIFO has data available, the received value is saved in the I2C RX Buffer if there is space left. If there is no space left, the received value gets ignored. When

the pending interrupt is the I2C TX FIFO Trigger, the skip counter increases and sends data through the I2C until reaching the maximum length. If the I2C TX count reaches the maximum package count and the count skip counter is more than 1 (Assuming the first send data from the I2C target is a false trigger and already happened.), the I2C Tx Done flag becomes true. Then, if the bridge detects any errors, the error code will be transmitted through the Bridge I2C Target.

When the pending interrupt is an I2C stop condition, the device checks to see if data was received; if true (I2C data received flag is true,) the received data buffer is stored into the transmit data buffer, and the SPI TX ready flag is set to true. If no I2C data is received, the device does not send anything. This ISR also handles I2C error interrupts by assigning the appropriate error code to the error status variable (TXFIFO Underflow, RXFIFO Overflow, Target Arbitration Lost, and Interrupt Overflow.)

```
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;
    }
}
```

The final part of this subsystem code is the SPI IRQ Handler. The SPI IRQ handler saves received data and sets an ongoing transmission flag when the SPI transmits data. When an SPI RX interrupt is pending, the device saves the received data to the SPI RX Buffer, turns an SPI RX Ongoing flag true, and sets the SPI RX Done flag when the SPI RX counter reaches the maximum length.

```
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 Porting Guide

First, open the project SYSCONFIG file and click on the Show Device View icon at the top right corner of the SYSCONFIG window.

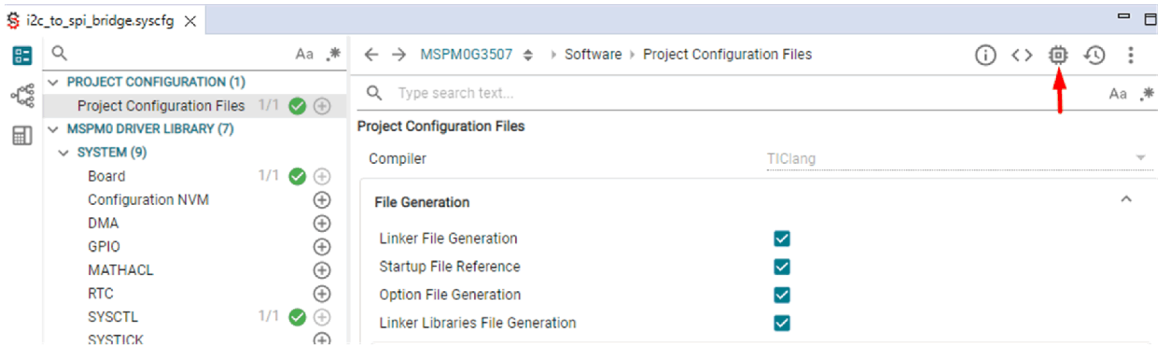

Figure 8-1. Show Device View icon location

After clicking the icon, the project target device package shows. Click the SWITCH button to change.

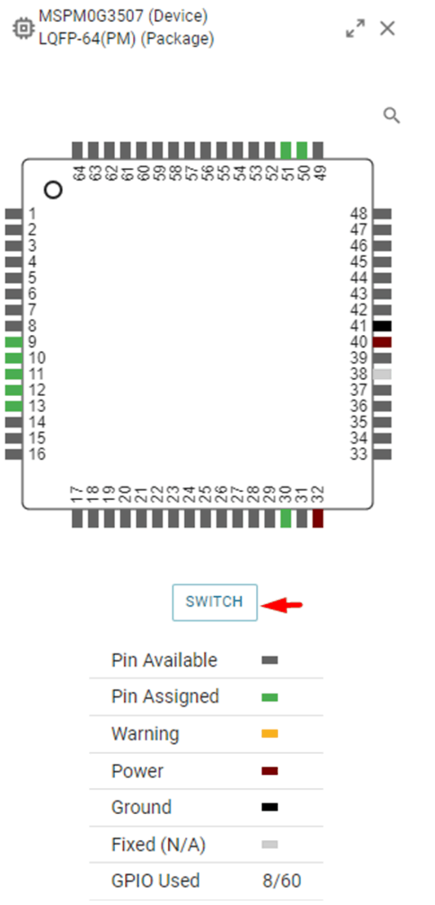

Figure 8-2. SWITCH button location

Next, the Migrate Settings open. Here, you can select the new value for a Board (if a user uses one), device, and package. Switch the chipset to the desired device in SYSCONFIG. Make sure to select the right MCU model and package When you finish setting up the new device, click the CONFIRM button.

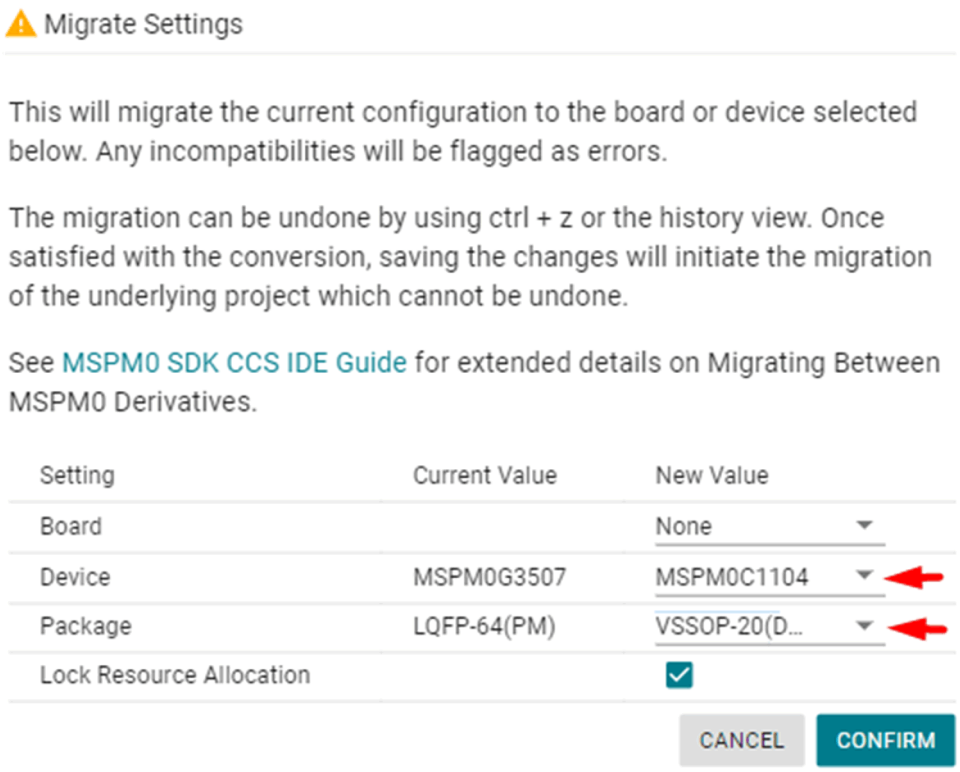

Figure 8-3. Migrate Settings window

After doing so, SYSCONFIG automatically adjusts the pins and peripherals to the new device (unless the pins are locked). There is a chance of errors showing up due to invalid values from the previous device in the new device, such as different pin values or a lack of a feature. The errors shows as a red X symbol.

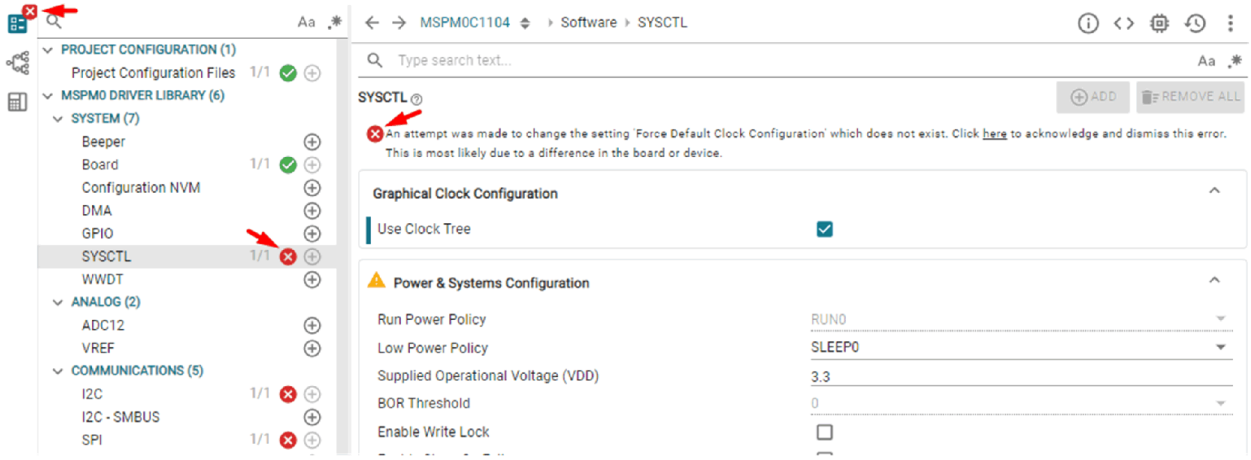

Figure 8-4. Errors After Switching Devices Example

To show all errors, click the Show Problems icon at the top right corner of the SYSCONFIG window. Read the errors and follow the instructions on how to fix them so the project compiles appropriately.

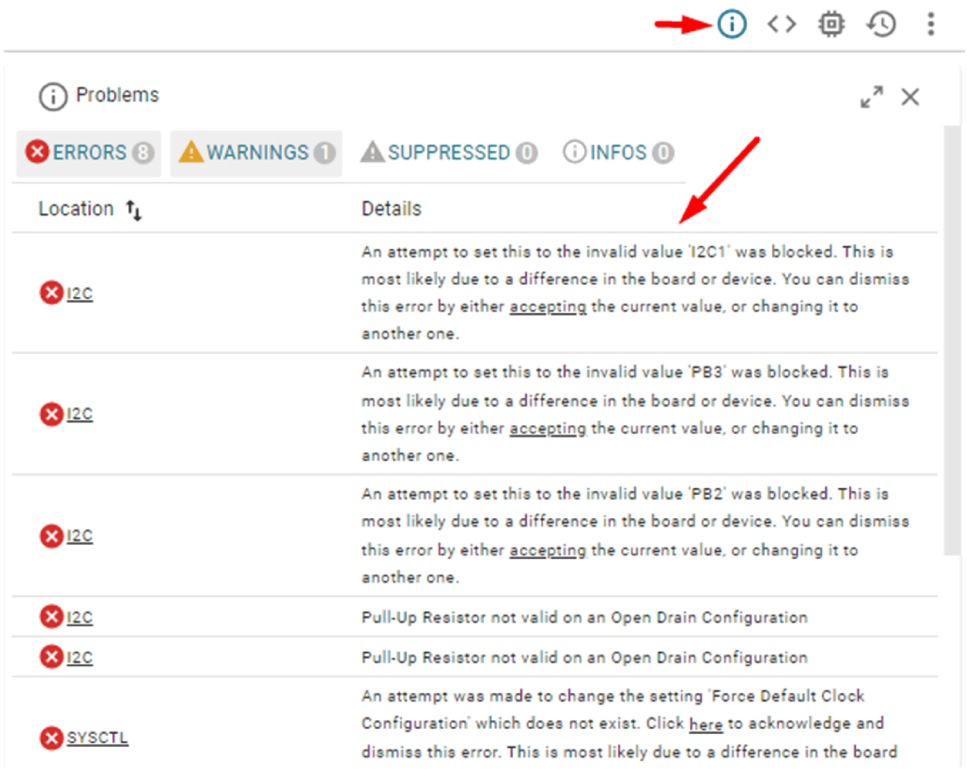

Figure 8-5. Show Problem Icon Location and Content Example

Sysconfig, checks the pins for the peripheral you want to use to so there is no conflict from the previous MCU. If necessary, change sysconfig to the desired pins you want to use in the new device running the project. Finally, build or compile the project in the new device. If done correctly, you see the following message and the ti_msp_dl_config.c and .h files inside the Debug folder.

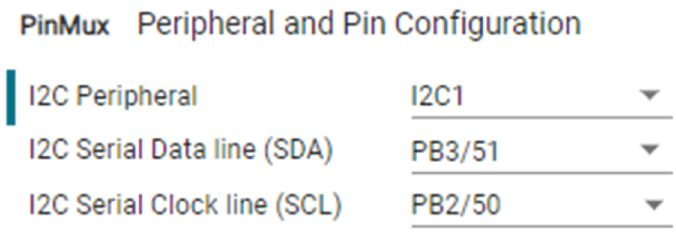

Figure 8-6. Pin Peripheral Configuration Example

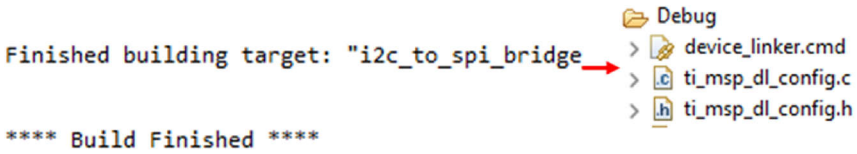

Figure 8-7. Project Build and File Generation Example

9 Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Revision * (February 2025) to Revision A (August 2025)	Page
• Removed compatible devices section	1

Trademarks

All trademarks are the property of their respective owners.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265