Subsystem Design

SPI to I2C Bridge

1 Design Description

This subsystem serves as a SPI-to-I2C bridge. In this subsystem, the MSPM0 device is the SPI Peripheral and I2C Controller. When a SPI controller transmits to the bridge SPI peripheral, the peripheral collects all the received data. Once the peripheral reaches the expected maximum message, the peripheral transmits the data using the I2C controller. The device sends an I2C transmit request and waits for I2C data from the I2C target. When the I2C controller finishes reading the data, the bridge waits for a SPI controller to send a request to read the data from the bridge. Finally, the bridge transmits the I2C controller received data through the bridge SPI peripheral.

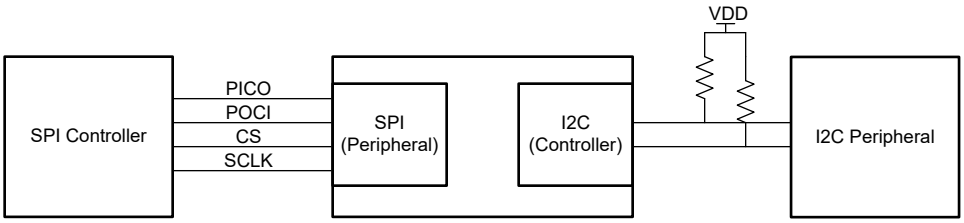

Figure 1-1. System Functional Block Diagram

2 Required Peripherals

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

Table 2-1. Peripherals

Sub-Block Functionality	Peripheral Use	Notes
SPI Peripheral	SPI	Called SPI_INST in code.
I2C Controller	I2C	Called I2C_INST in code. Default 100kHz transmission rate.

3 Compatible Devices

Any MSPM0 device and EVMs or Launchpads can use this subsystem if the required peripherals are present.

4 Design Steps

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

Copyright © 2025 Texas Instruments Incorporated

5 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.

Copyright © 2025 Texas Instruments Incorporated

6 Software Flowchart

Figure 6-1 shows the code flow diagram for this example and explains how the device fills the data buffers with received SPI data, then transfers the data out through I2C.

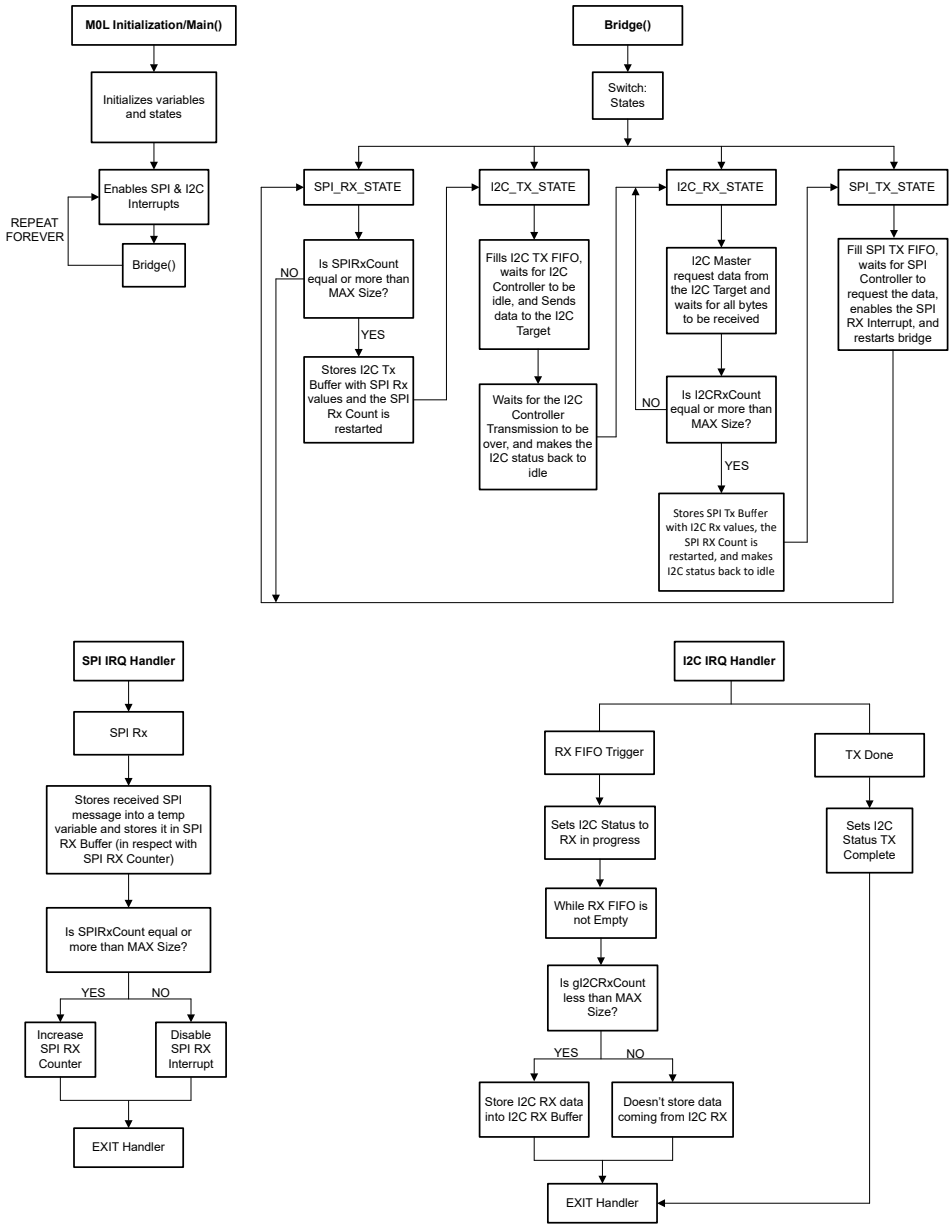

Figure 6-1. Application Software Flowchart

Figure 6-1 shows a high-level diagram of how the communication works. X bytes here represent the Maximum package of bytes found in the communication process.

Copyright © 2025 Texas Instruments Incorporated

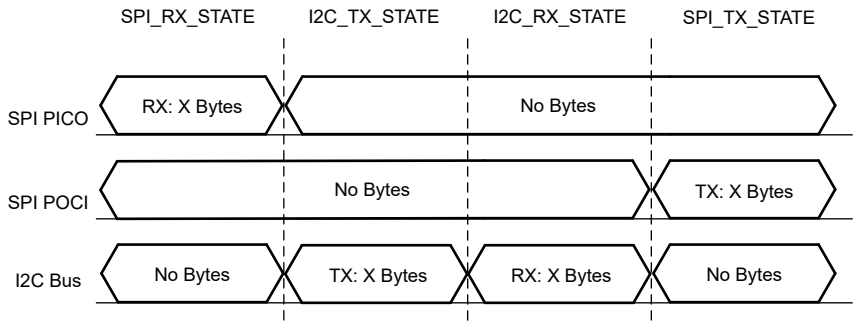

Figure 6-2. High Level Communication Diagram

7 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 6-1 is found in the beginning of main() in the *spi_to_i2c_bridge.c* file.

8 Application Code

The initialization of the buffers, counters, enum, and flag are shown here. To change the specific values used by the SPI and I2C 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"
/* 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);
```


Copyright © 2025 Texas Instruments Incorporated

The main body of the application code is relatively short. First, the device peripherals and interrupts get initialized. Then, a delay occurs for the SPI TX, which is idle before starting transmission, while also the state and flag values are initialized. Following up, the main loop, which contains the bridge function, runs.

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

The bridge has four states. The first state focuses on transferring the data from the SPI RX buffer to the I2C TX buffer while clearing the SPI RX buffer after the maximum package size has been received by the SPI Peripheral. The second state transmits the data from the I2C TX buffer to an I2C Target. Then, the third state makes the I2C Controller send a request data signal to the I2C target, and after recollecting the data, transfers the I2C RX Buffer to the SPI TX Buffer while clearing the I2C RX buffer. The final state fills the SPI Target with the contents of SPI TX Buffers, waits for the FIFOs to be empty (Controller asks for the data,) enables the SPI RX to Interrupt again, and restarts the bridge.

```
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 */
```


Copyright © 2025 Texas Instruments Incorporated

```
            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();
            }
            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;
    }
}
```

The next piece of this example is the SPI IRQ Handler. Only one interrupt is used for this example: SPI RX. When activated, the data is stored in a temporal variable and then stored in the SPI RX FIFO Buffer. Then, if the SPI RX Counter is less than the maximum package size, the SPI Rx counter increases; otherwise, the SPI RX FIFO interrupt is disabled.

```
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;
    }
}
```

The final piece of code in this example is the I2C IRQ Handler. The two interrupts in this example are Controller TX Done and Controller RX FIFO Trigger. The I2C Controller Status is updated to TX Completed when TX Done is triggered. When the RX FIFO Trigger is triggered, the I2C Controller Status is updated to RX In Progress; the I2C Rx buffer receives the message stored in the I2C RX FIFO until is empty and sets the I2C Controller Status to I2C RX Complete.

```
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;
```


Copyright © 2025 Texas Instruments Incorporated

```
        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;
        default:
            break;
    }
}
```

9 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.

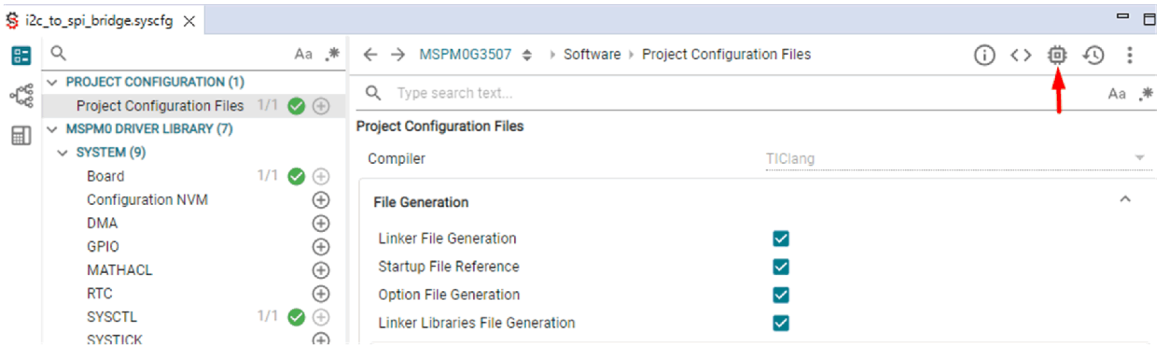

Figure 9-1. Show Device View Icon Location

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

Copyright © 2025 Texas Instruments Incorporated

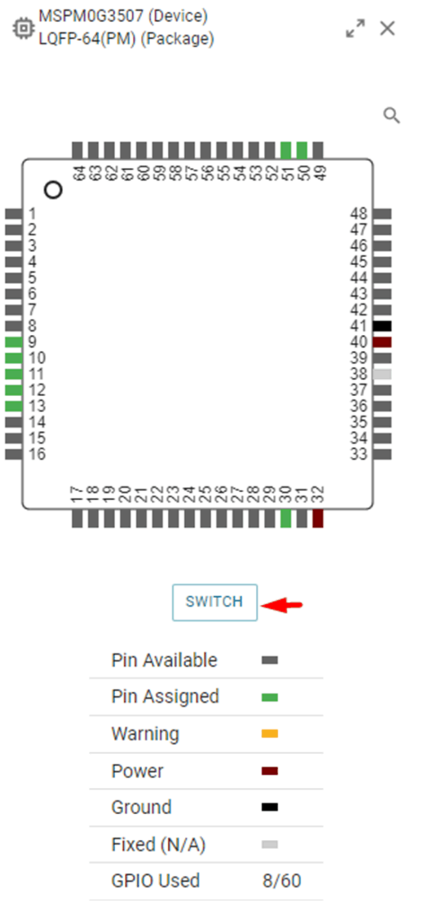

Figure 9-2. SWITCH Button Location

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.

Copyright © 2025 Texas Instruments Incorporated

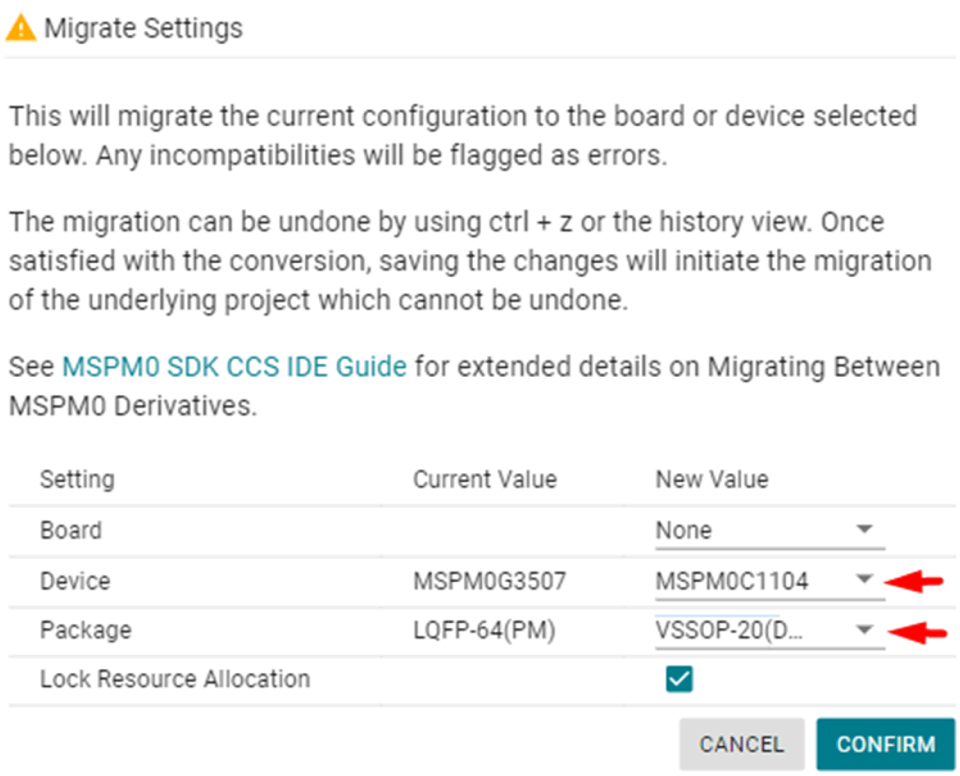

Figure 9-3. Migrate Settings Window

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 show as a red X symbol.

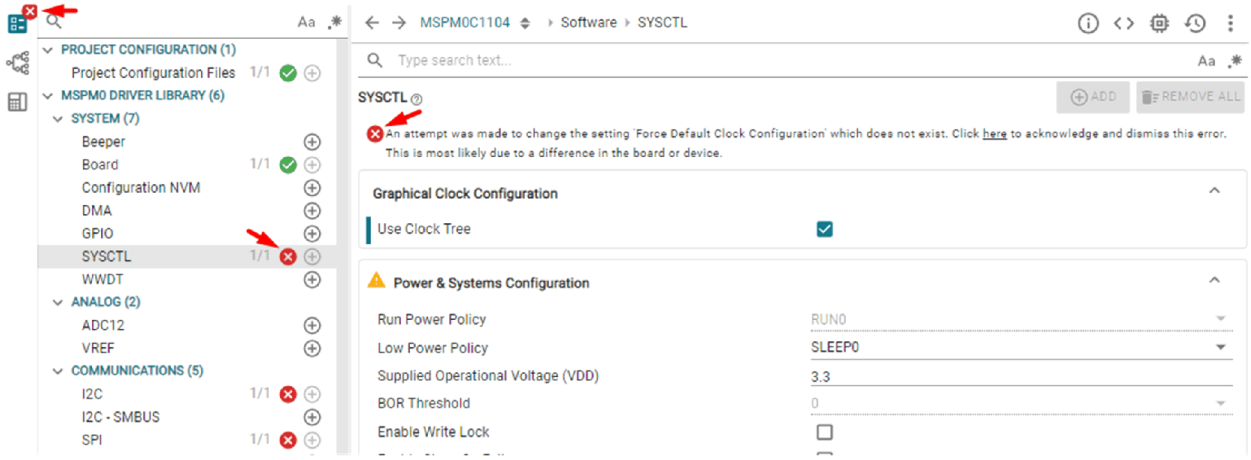

Figure 9-4. Errors After Switching Devices Example

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.

Copyright © 2025 Texas Instruments Incorporated

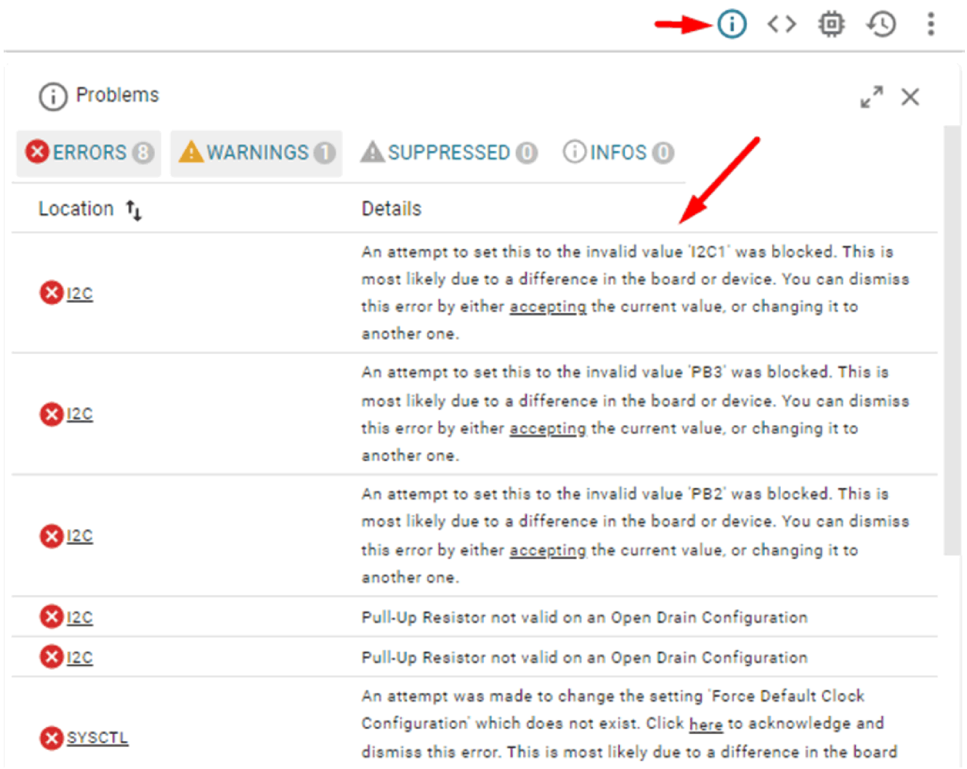

Figure 9-5. Show Problem Icon Location and Content Example

Sysconfig, check the pins for the peripheral you want to use so there is no conflict from the previous MCU. If necessary, change 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.

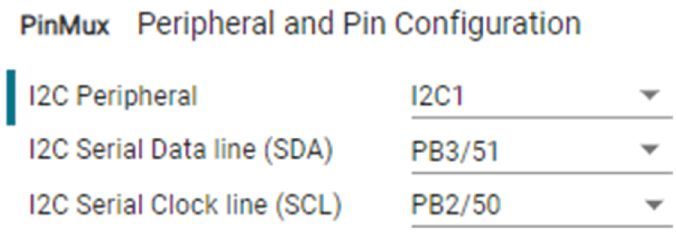

Figure 9-6. Pin Peripheral Configuration Example

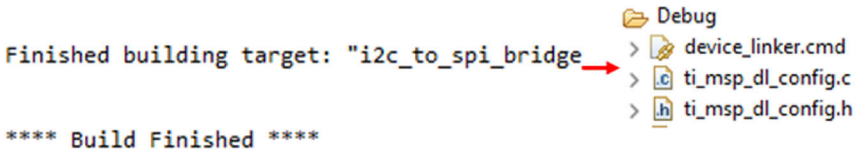

Figure 9-7. Project Build and File Generation Example

Copyright © 2025 Texas Instruments Incorporated

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
Copyright © 2025, Texas Instruments Incorporated