Subsystem Design

USB to SPI Bridge

1 Description

This subsystem demonstrates how to implement MSPM0 devices as a USB Communications Device Class - Abstract Control Model (USB CDC-ACM) to serial peripheral interface (SPI) bridge. When connected to a USB host, the device enumerates as a virtual COM port. No custom drivers are required on Windows®, macOS®, or Linux®.

The bridge forwards USB host data to the SPI bus as write transactions, and returns SPI slave read data back to the USB host. Because SPI is a full-duplex bus, the bridge sends dummy bytes on MOSI during read transactions to generate the SPI clock and simultaneously receive data on MISO. The SPI slave device signals data availability through a dedicated GPIO pin (data-ready signal), which triggers an SPI read transaction. This example supports two target platforms: the MSPM0G5187 (LaunchPad™, LQFP-64) and the MSPM0C5116 (LaunchPad, LQFP-48). Both platforms run the same bridge application code, with device-specific SysConfig peripheral assignments.

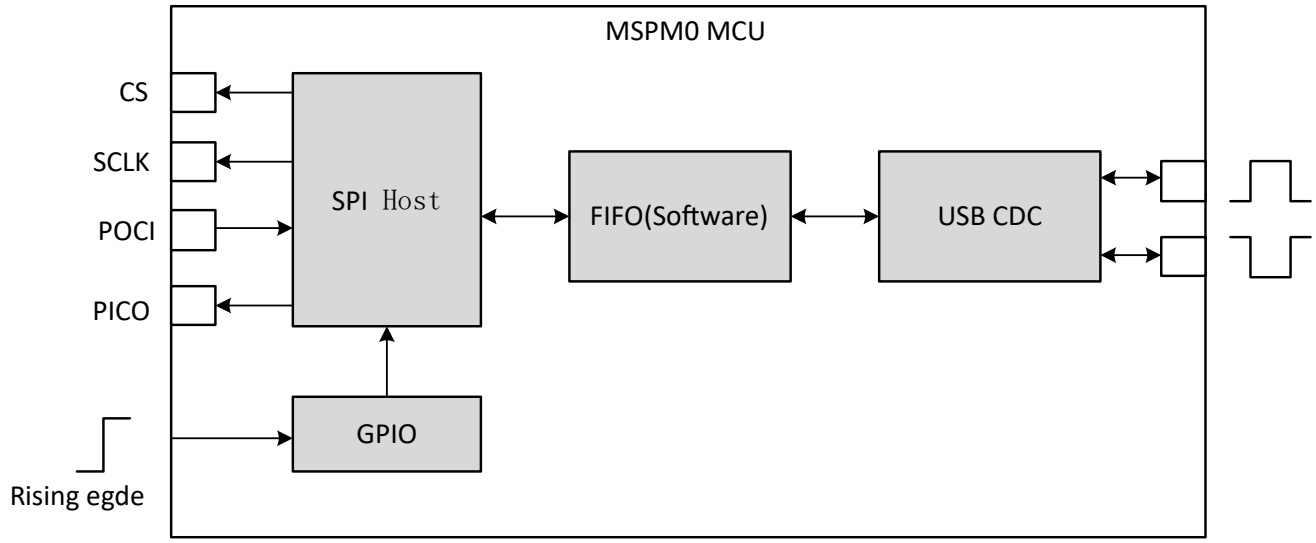

Figure 1-1. System Functional Block Diagram

2 Required Peripherals

Both platforms use the same peripheral types. SysConfig assigns device-specific instances but generates consistent macro names (SPI_0_INST, DMA_CH0_CHAN_ID, DMA_CH1_CHAN_ID), so the application code compiles unchanged on both devices.

Table 2-1. Peripheral Information

Peripheral	Function	MSPM0G5187	MSPM0C5116
USB	USB full speed CDC device	USBFS0 (USB_0)	USBLC0 (USB_0)
SPI	SPI controller to target device	UC2 (SPI_0_INST)	UC18 (SPI_0_INST)
DMA (CH1)	SPI TX acceleration	DMA_CH1	DMA_CH1
DMA (CH0)	SPI RX acceleration	DMA_CH0	DMA_CH0
GPIO	Target data-ready notification input	PB25	PB24

3 Compatible Devices

- MSPM0G5187
- MSPM0G11x
- MSPM0C511x

4 Design Steps

1. Set up the TinyUSB module in SysConfig. Add one CDC instance (CDC_0). SysConfig allocates three endpoints automatically: notification IN (EP_IN2), data IN (EP_IN1), and data OUT (EP_OUT1).
2. Configure the USB clock tree. The two platforms use different clock paths. On MSPM0G5187: enable the PLL and USB FLL, set UDIV to 2, and select HSCLKMUX_SYSPLL0. On MSPM0C5116: enable only the USB FLL (no PLL needed), set UDIV to 2, and select HSCLKMUX_USBFLL.
3. Set up the SPI module in SysConfig.
   a. Enable SPI Controller mode.
   b. Set the bit rate: the default 8MHz for MSPM0G5187, or 4MHz for MSPM0C5116.
   c. Set both TX and RX FIFO thresholds to DL_SPI_RX/TX_FIFO_LEVEL_ONE_FRAME.
   d. Configure DMAEvent1 as DL_SPI_DMA_INTERRUPT_RX and assign DMA_CH0 in *Fixed addr. to Block addr.* mode.
   e. Configure DMAEvent2 as DL_SPI_DMA_INTERRUPT_TX and assign DMA_CH1 in *Block addr. to Fixed addr.* mode.
   f. In the Interrupt Configuration tab, enable the DMA_DONE_TX and DMA_DONE_RX interrupts.
4. Set up the GPIO module in SysConfig.
   a. Add one input GPIO pin named SPI_SIGNAL configured for rising-edge interrupt with pull-down resistance. This pin connects to the SPI target device's data-ready output.
   b. Enable the input interrupt. On MSPM0G5187, map the GPIO to Interrupt Group 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_SPI_SIGNAL_IIDX:
                    // Slave signaled data ready - set flag for main loop
                    SPISlaveDataReady = true;
                    break;

                default:
                    break;
            }
            break;

        default:
            break;
    }
}
```

   On MSPM0C5116, the GPIOB interrupt is used directly.

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

        default:
            break;
    }
}
```

   c. Set up SysTick in SysConfig. Enable SysTick with a period matched to the CPU frequency: 80000 cycles for MSPM0G5187 (80MHz) or 48000 cycles for MSPM0C5116 (48 MHz). Both values give a 1ms period. TinyUSB's board_millis() requires this 1ms period.

5 Design Considerations

1. *Buffer sizing:* USB_FRAME_PACKAGE defaults to 64 bytes, matching the USB Full Speed maximum packet size. Each FIFO queue holds up to 8 tasks (USB_TX_BUF_SIZE / SPI_TX_BUF_SIZE) for 512 bytes per direction. Increase the queue depth to handle burst traffic at the cost of RAM.

```
// ===== Queue Configuration =====
#define USB_TX_BUF_SIZE     8              // Max tasks in SPI→USB queue
#define SPI_TX_BUF_SIZE     8              // Max tasks in USB→SPI queue

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

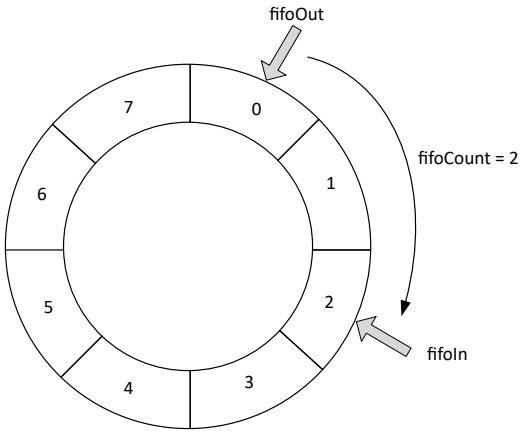

Figure 5-1. Buffer Structure

```
// ===== Single Task Structure =====
// 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 data-ready signal.* The SPI target device asserts a dedicated GPIO output (active high) when it has data ready for the controller to read. The bridge detects the rising edge via GPIO interrupt and sets the SPISlaveDataReady flag. The main loop polls this flag and calls spi2usb_pushTask() to initiate the SPI read transaction. The target must hold the GPIO high until the controller completes the transfer. The GPIO pin uses an internal pull-down resistor so the line idles low when the target has no data.
3. *Full-duplex SPI read:* SPI is a synchronous full-duplex bus. To receive data from the target, the controller must simultaneously transmit bytes on to generate the SPI clock. The bridge sends the contents of the receive task buffer as dummy PICO data (it is not a valid payload). The target ignores dummy bytes; the actual received data arrives on POCI and is captured by DMA_CH0. The TX DMA interrupt (DMA_DONE_TX) is disabled during SPI reads so that only DMA_DONE_RX marks the completion of a read transaction.
4. *DMA for SPI TX:* DMA channel CH1 runs in *Block addr. to Fixed addr.* mode and loads the SPI TX FIFO directly from the task buffer. After the DMA finishes loading the FIFO, the DMA_DONE_TX interrupt fires. The bridge enables the TX_EMPTY interrupt, which fires when the last byte has physically left the SPI bus. Only at that point does the bridge release the TX task and clear spiInProgress.
5. *DMA for SPI RX:* DMA channel CH0 runs in *Fixed addr. to Block addr.* mode and captures incoming bytes from the SPI RX FIFO into the task buffer. When all 64 bytes have been received, the DMA_DONE_RX

interrupt fires. The handler marks the task ready for USB transmission and clears SPISlaveDataReady and spiInProgress.

6. *Overflow protection:* When the USB-to-SPI queue is full, incoming USB data is read into a dummy buffer to prevent stalling the TinyUSB RX buffer. When the SPI-to-USB queue is full, spi2usb_pushTask() returns immediately without initiating a new SPI read.

6 Software Flowchart

Figure 6-1 shows the code flow for this example.

USB to SPI direction:

1. TinyUSB invokes tud_cdc_rx_cb() each time the USB host sends data. The callback calls usb2spi_pushTask(), which reads up to 64 bytes from the USB CDC interface and queues them as a task in gUSB2SPITask.
2. In the main loop, when gUSB2SPITask.fifoCount > 0, usb2spi_popTask() flushes the SPI TX FIFO, configures DMA channel CH1 with the task buffer address and length, enables the DMA_DONE_TX interrupt, and starts the SPI write transfer.
3. The DMA_DONE_TX interrupt fires when DMA finishes loading the TX FIFO. The handler clears and enables the TX_EMPTY interrupt. The TX_EMPTY interrupt fires when the last byte has been clocked out. The handler calls fifoCompleteRead() to release the task and clears spiInProgress.

SPI to USB direction:

1. The SPI target device asserts the data-ready GPIO pin (rising edge). The GPIO interrupt handler sets SPISlaveDataReady = true and returns immediately.
2. In the main loop, when SPISlaveDataReady is set and spiInProgress is false, spi2usb_pushTask() flushes both RX and TX FIFOs, configures DMA channel CH0 for the receive buffer, configures DMA channel CH1 to send dummy data on MOSI, and starts both DMAs. The TX_DONE_TX interrupt is explicitly disabled so that only DMA_DONE_RX signals completion.
3. The DMA_DONE_RX interrupt fires when all 64 bytes have been received. The handler calls fifoCompleteWrite() to mark the task ready for USB transmission, clears SPISlaveDataReady, and clears spiInProgress.
4. In the main loop, when gSPI2USBTask.fifoCount > 0, spi2usb_popTask() checks that the TinyUSB CDC TX FIFO has sufficient space, writes the buffered data, flushes it to the host, and calls fifoCompleteRead() to release the task.

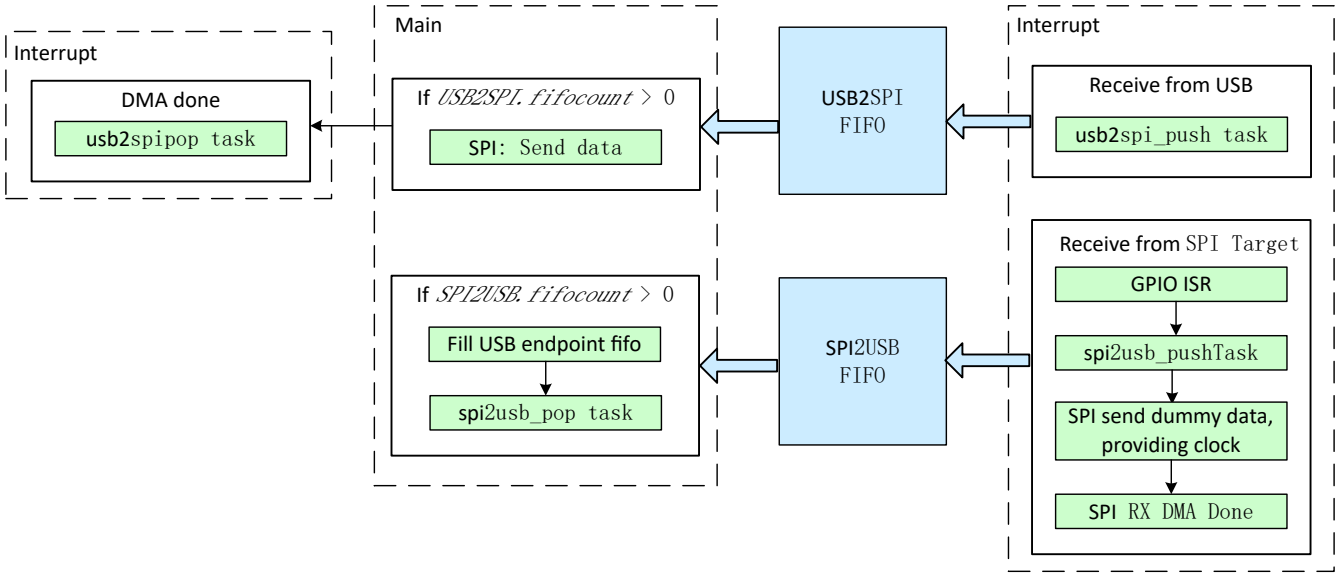

Figure 6-1. Application Software Flowchart

7 Application Code

USB to SPI:

```
/*********************************************************
* Function name  : usb2spi_popTask
* Description    : USB -> SPI. Pop task from queue and transmit data through
*                  SPI to slave device. Waits if SPI TX already in progress.
*                  Called from main loop when fifoCount > 0.
* Parameter      : None (modifies gUSB2SPITask, spiInProgress)
* Return         : None
*********************************************************/
void usb2spi_popTask(void)
{
    if(spiInProgress) return;
    if(gUSB2SPITask.fifoCount == 0) return;

    // Current process task
    TxElement_t *pTask = &gUSB2SPITask.fifoPointer[gUSB2SPITask.fifoOut];

    //Flush TXFIFO
    DL_SPI_flushTXFIFO(SPI_0_INST);

    /* Setup DMA and start SPI write */
    DL_SPI_clearInterruptStatus(SPI_0_INST, DL_SPI_INTERRUPT_DMA_DONE_TX);
    DL_SPI_enableInterrupt(SPI_0_INST, DL_SPI_INTERRUPT_DMA_DONE_TX);
    DL_DMA_setSrcAddr(DMA, DMA_CH1_CHAN_ID, (uint32_t)(pTask->buffer));
    DL_DMA_setDestAddr(DMA, DMA_CH1_CHAN_ID, (uint32_t)(&SPI_0_INST->spi->TXDATA));
    DL_DMA_setTransferSize(DMA, DMA_CH1_CHAN_ID, pTask->length);
    DL_DMA_enableChannel(DMA, DMA_CH1_CHAN_ID);

    spiInProgress = true;
}
```

```
/* ===== USB RX -> SPI TX ===== */

/*********************************************************
* Function name  : usb2spi_pushTask
* Description    : USB -> SPI. Read data from USB CDC interface and push
*                  into gUSB2SPITask queue. Each task holds one USB frame
*                  (up to 64 bytes). Called from tud_cdc_rx_cb callback.
* Parameter      : None (modifies gUSB2SPITask)
* Return         : None
*********************************************************/
void usb2spi_pushTask(void)
{
    // Check space for new task
    if(gUSB2SPITask.fifoCount >= SPI_TX_BUF_SIZE) return;

    // Get current task slot in queue
    TxElement_t *pTask = &gUSB2SPITask.fifoPointer[gUSB2SPITask.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(&gUSB2SPITask, SPI_TX_BUF_SIZE);
}
```

SPI to USB:

```
/* ===== SPI RX -> USB TX ===== */

/*******************************************************
* Function name  : spi2usb_pushTask
* Description    : SPI -> USB. Read data from SPI slave and queue for USB transmission
*                  Triggered when target device signals data ready via GPIO pin.
*                  Called from GPIO interrupt handler.
* Parameter      : None (modifies gSPI2USBTask)
* Return         : None
*******************************************************/
void spi2usb_pushTask(void)
{
    // Check if SPI→USB queue has space for new task
    if (gSPI2USBTask.fifoCount >= USB_TX_BUF_SIZE) return;
    if(spiInProgress) return;

    // Get current task buffer
    TxElement_t *pTask = &gSPI2USBTask.fifoPointer[gSPI2USBTask.fifoIn];

    // Pre-set length since SPI master reads fixed 64 bytes per transaction
    pTask->length = USB_FRAME_PACKAGE;

    //Flush RXFIFO
    DL_SPI_flushRXFIFO(SPI_0_INST);

    //Flush TXFIFO
    DL_SPI_flushTXFIFO(SPI_0_INST);

    // Transmit dummy data
    DL_SPI_disableInterrupt(SPI_0_INST, DL_SPI_INTERRUPT_DMA_DONE_TX);
    DL_DMA_setSrcAddr(DMA, DMA_CH1_CHAN_ID, (uint32_t)(pTask->buffer));
    DL_DMA_setDestAddr(DMA, DMA_CH1_CHAN_ID, (uint32_t)(&SPI_0_INST->spi->TXDATA));
    DL_DMA_setTransferSize(DMA, DMA_CH1_CHAN_ID, USB_FRAME_PACKAGE);
    DL_DMA_enableChannel(DMA, DMA_CH1_CHAN_ID);

    /* DMA to receive data */
    DL_SPI_flushRXFIFO(SPI_0_INST);

    DL_DMA_setSrcAddr(DMA, DMA_CH0_CHAN_ID, (uint32_t)(&SPI_0_INST->spi->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);

    spiInProgress = true;
}
```

```
/*******************************************************
* Function name  : spi2usb_popTask
* Description    : SPI -> 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 gSPI2USBTask)
* Return         : None
*******************************************************/
void spi2usb_popTask(void)
{
    if(gSPI2USBTask.fifoCount == 0) return;

    TxElement_t *pTask = &gSPI2USBTask.fifoPointer[gSPI2USBTask.fifoOut];

    if(tud_cdc_n_write_available(0) >= pTask->length){
        uint32_t written = tud_cdc_n_write(0, pTask->buffer, pTask->length);

        if(written == pTask->length){
            tud_cdc_n_write_flush(0);
            fifoCompleteRead(&gSPI2USBTask, USB_TX_BUF_SIZE);
        }
    }
}
```

SPI Interrupt Handler:

```
//-----------------------------------------------------------------------+
// SPI Interrupt Handler
//-----------------------------------------------------------------------+
void SPI_0_INST_IRQHandler(void) {
    switch(DL_SPI_getPendingInterrupt(SPI_0_INST)) {
        case DL_SPI_IIDX_DMA_DONE_TX:
            // TX: DMA transfer to SPI FIFO complete
            DL_SPI_clearInterruptStatus(SPI_0_INST, DL_SPI_INTERRUPT_TX_EMPTY);
            DL_SPI_enableInterrupt(SPI_0_INST, DL_SPI_INTERRUPT_TX_EMPTY);
            break;

        case DL_SPI_IIDX_DMA_DONE_RX:
            // RX complete: DMA received 64 bytes from SPI slave
            fifoCompleteWrite(&gSPI2USBTask, USB_TX_BUF_SIZE);
            SPISlaveDataReady = false;  // Clear slave notification flag
            spiInProgress = false;
            break;

            // TX complete
        case DL_SPI_IIDX_TX_EMPTY:
            DL_SPI_disableInterrupt(SPI_0_INST, DL_SPI_INTERRUPT_TX_EMPTY);
            fifoCompleteRead(&gUSB2SPITask, SPI_TX_BUF_SIZE);
            spiInProgress = false;
            break;

        default:
            break;
    }
}
```

8 Data Flow Examples

Scenario A: Host writes 5 bytes to target

1. Host sends *HELLO* via COM port
2. tud_cdc_rx_cb() queues data
   a. usb2spi_pushTask()
   b. Queue [H, E, L, L, O] with length = 5
3. Main loop: usb2spi_popTask()
   a. Get task from queue
   b. SPI write: data=[H,E,L,L,O]
   c. DMA transfers to SPI TX FIFO
4. SPI hardware executes
5. SPI_0_INST_IRQHandler()
   a. DL_SPI_IIDX_DMA_DONE_TX: DMA finished
   b. DL_SPI_IIDX_TX_EMPTY: Transmission complete
   c. fifoCompleteRead() marks task consumed

Scenario B: host reads 64 bytes from target

1. SPI target has data, pulls GPIO high
2. GPIO interrupt fires
   a. SPISlaveDataReady = true
3. Main loop detects: spi2usb_pushTask()
   a. Flush SPI RX/TX FIFOs
   b. Setup DMA TX: send dummy data (clock generation)
   c. Setup DMA RX: SPI RX → buffer
   d. Start SPI transfer (64 bytes)
4. SPI Hardware executes
5. SPI_0_INST_IRQHandler()
   a. DL_SPI_IIDX_DMA_DONE_RX: All data received
   b. fifoCompleteWrite(&gSPI2USBTask), and mark ready for USB transmission
6. Main loop: spi2usb_popTask()
   a. Send 64 bytes to USB
   b. fifoCompleteRead() marks task done
7. Host receives 64 bytes on COM port

9 Additional Resources

- Texas Instruments, *Download the MSPM0 SDK*
- Texas Instruments, *Learn more about SysConfig*
- Texas Instruments, *MSPM0G5187*
- Texas Instruments, *MSPM0G5117*

10 Trademarks

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), 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 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, TI’s General Quality Guidelines, 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. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

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

Last updated 10/2025