Subsystem Design

USB to I2C Bridge

1 Description

This subsystem demonstrates how to implement MSPM0 devices as a USB Communications Device Class - Abstract Control Model (USB CDC-ACM) to I2C 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 I2C bus as write transactions, and returns I²C slave read data back to the USB host. The I²C slave device signals data availability through a dedicated GPIO pin (data-ready signal), which triggers an I²C read transaction. The example uses the open-source TinyUSB library for the USB stack. 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.

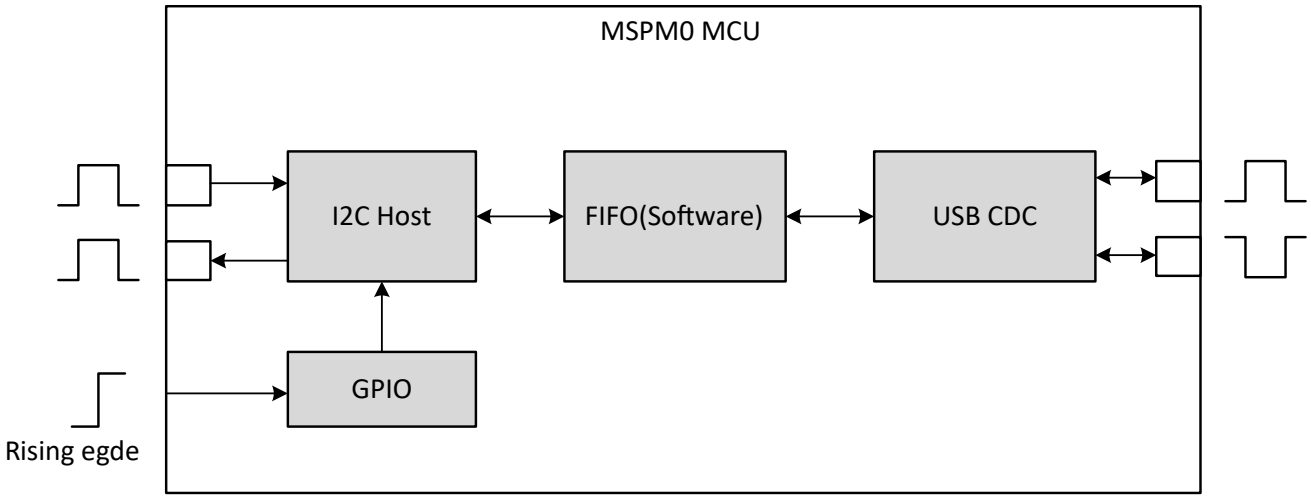

Figure 1-1. System Functional Block Diagram

2 Required Peripherals

Both platforms use the same peripheral types. SysConfig assigns device-specific instances but generates consistent macro names (I2C_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)
I2C	I2C controller to target device	UC1 (I2C_0_INST)	UC16 (I2C_0_INST)
DMA(CH1)	I2C TX acceleration	DMA_CH1	DMA_CH1
DMA(CH0)	I2C 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 I2C module in SysConfig.
   a. Enable I²C Controller mode.
   b. Set the bus speed to Fast Plus (1 MHz) on MSPM0G5187 or Fast (400 kHz) on MSPM0C5116.
   c. Configure DMA event1 trigger as CONTROLLER_RXFIFO_TRIGGER and assign DMA_CH0 in *Fixed addr. to Block addr.* mode.
   d. Configure DMA event2 trigger as CONTROLLER_TXFIFO_TRIGGER and assign DMA_CH0 in *Block addr. to Fixed addr.* mode.
   e. In the Interrupt Configuration tab, enable the EVENT1_DMA_DONE, EVENT2_DMA_DONE, and NACK interrupts.
4. Set up the GPIO module in SysConfig.
   a. Add one input GPIO pin named I2C_SIGNAL configured for rising-edge interrupt with pull-down resistance. This pin connects to the I2C 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_I2C_SIGNAL_IIDX:
                    // Slave signaled data ready - set flag for main loop
                    i2ctargetDataReady = 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_I2C_SIGNAL_IIDX:
            // Slave signaled data ready - set flag for main loop
            i2ctargetDataReady = true;
            break;

        default:
            break;
    }
}
```

5. Set up SysTick in SysConfig. Enable SysTick with a period matched to the CPU frequency: 80000 cycles for MSPM0G5187(80 MHz) 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 / I2C_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 USB→I2C queue
#define I2C_TX_BUF_SIZE     8             // Max tasks in I2C→USB queue

// ===== USB/I2C Parameters =====
#define USB_FRAME_PACKAGE   64             // USB Full Speed max packet size (bytes)
```

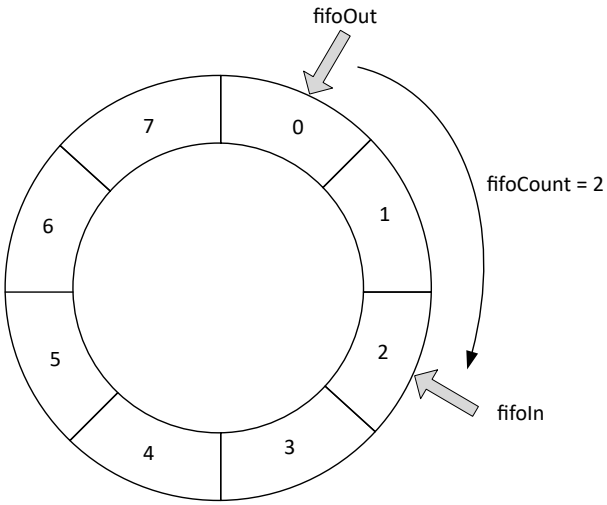

Figure 5-1. Buffer Structure

```
// 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 I2C 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 i2ctargetDataReady flag. The main loop polls this flag and calls i2c2usb_pushTask() to initiate the I2C read transaction. The target must hold the GPIO high until the controller completes the read. The GPIO pin uses an internal pull-down resistor so the line idles low when the target has no data.
3. *DMA for I2C TX:* DMA channel CH1 runs in *Block addr. to Fixed addr* mode and loads the I2C TX FIFO directly from the task buffer without CPU involvement. After the DMA loads the last byte into the FIFO, the DMA done interrupt fires. The bridge then enables the I²C STOP condition interrupt dynamically. Only when the STOP interrupt fires—indicating the last bit has been transmitted on the bus—does the bridge release the TX task and clear i2cInProgress.
4. *DMA for I2C RX*: DMA channel CH0 runs in *Fixed addr. to Block addr.* mode and moves received bytes from the I2C RX FIFO directly to the task buffer. When the expected number of bytes has been received,

the DMA done interrupt happens. The handler marks the task ready for USB transmission and clears i2ctargetDataReady and i2cInProgress.

5. *Overflow protection:* When the USB-to-I2C queue is full, incoming USB data is read into a dummy buffer to prevent stalling the TinyUSB RX buffer. When the I2C-to-USB queue is full, the i2c2usb_pushTask() function returns immediately without initiating a new I²C read. For production designs, add application-level flow control or increase the queue depth.

6 Software Flowchart

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

USB to I2C direction:

1. TinyUSB invokes tud_cdc_rx_cb() each time the USB host sends data. The callback calls usb2i2c_pushTask(), which reads up to 64 bytes from the USB CDC interface and queues them as a task in gUSB2I2CTask.
2. In the main loop, when gUSB2I2CTask.fifoCount > 0, usb2i2c_popTask() configures DMA channel CH1 with the task buffer address and length, then starts the I²C write transfer to the target device.
3. The TX DMA done interrupt fires when DMA finishes loading the I²C TX FIFO. The handler clears the STOP interrupt status flag and enables the STOP interrupt. The STOP interrupt fires when the I²C STOP condition appears on the bus. The handler calls fifoCompleteRead() to release the task and clears i2cInProgress.

I2C to USB direction:

1. The I2C target device asserts the data-ready GPIO pin (rising edge). The GPIO interrupt handler sets i2ctargetDataReady
2. In the main loop, when i2ctargetDataReady is set and i2cInProgress is false, i2c2usb_pushTask() configures DMA channel CH0 for the receive buffer, starts the I2C read transfer for 64 bytes, and sets i2cInProgress. The data length is always fixed to 64 bytes.
3. The RX DMA done interrupt fires when all 64 bytes have been received. The handler calls fifoCompleteWrite() to mark the task ready for USB transmission, clears i2ctargetDataReady, and clears i2cInProgress.
4. In the main loop, when gI2C2USBTask.fifoCount > 0, i2c2usb_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.

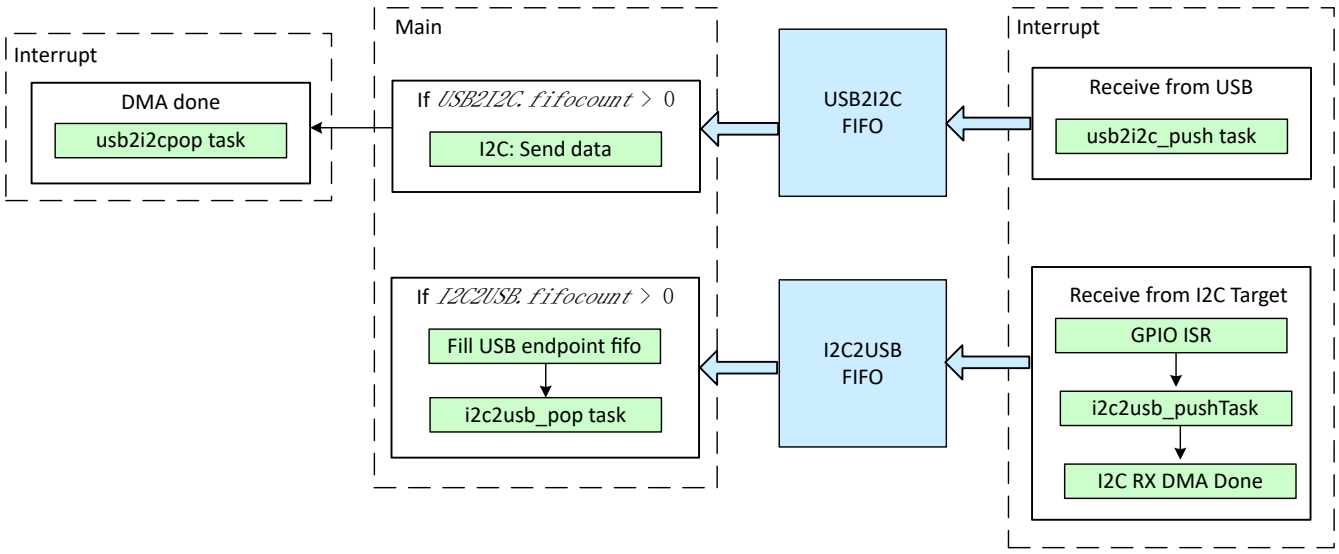

Figure 6-1. Application Software Flowchart

7 Application Code

USB to I2C:

```
/*******************************************************
* Function name  : usb2i2c_pushTask
* Description    : USB -> I2C. Read data from USB CDC interface and push
*                  into gUSB2I2CTask queue. Each task holds one USB frame
*                  (up to 64 bytes). Called from tud_cdc_rx_cb callback.
* Parameter      : None (modifies gUSB2I2CTask)
* Return         : None
*******************************************************/
void usb2i2c_pushTask(void)
{
    // Check space for new task
    if(gUSB2I2CTask.fifoCount >= I2C_TX_BUF_SIZE) return;

    // Get current task slot in queue
    TxElement_t *pTask = &gUSB2I2CTask.fifoPointer[gUSB2I2CTask.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(&gUSB2I2CTask, I2C_TX_BUF_SIZE);
}
```

```
/*******************************************************
* Function name  : usb2i2c_popTask
* Description    : USB -> I2C. Pop task from queue and transmit data through
*                  I2C to slave device. Waits if I2C TX already in progress.
*                  Called from main loop when fifoCount > 0.
* Parameter      : None (modifies gUSB2I2CTask, i2cInProgress)
* Return         : None
*******************************************************/
void usb2i2c_popTask(void)
{
    if(i2cInProgress) return;

    if(gUSB2I2CTask.fifoCount == 0) return;

    // Current process task
    TxElement_t *pTask = &gUSB2I2CTask.fifoPointer[gUSB2I2CTask.fifoOut];

    /* Setup DMA and start I2C write */
    DL_DMA_setSrcAddr(DMA, DMA_CH1_CHAN_ID, (uint32_t)(pTask->buffer));
    DL_DMA_setDestAddr(DMA, DMA_CH1_CHAN_ID, (uint32_t)(&I2C_0_INST->i2cc->TXDATA));
    DL_DMA_setTransferSize(DMA, DMA_CH1_CHAN_ID, pTask->length);
    DL_DMA_enableChannel(DMA, DMA_CH1_CHAN_ID);

    DL_I2CC_startTransfer(I2C_0_INST, I2C_ADDRESS, DL_I2CC_DIRECTION_TX, pTask->length);

    i2cInProgress = true;
}
```

I2C to USB:

```
/*********************************************************
* Function name  : i2c2usb_pushTask
* Description    : I2C -> USB. Read data from I2C slave and queue for USB transmission
*                  Triggered when slave device signals data ready via GPIO pin.
*                  Called from GPIO interrupt handler.
* Parameter      : None (modifies gI2C2USBTask)
* Return         : None
*********************************************************/
void i2c2usb_pushTask(void)
{
    // Check space for new task
    if (gI2C2USBTask.fifoCount >= USB_TX_BUF_SIZE) return;
    if(i2cInProgress) return;

    // Current process task
    TxElement_t *pTask = &gI2C2USBTask.fifoPointer[gI2C2USBTask.fifoIn];
    pTask->length = USB_FRAME_PACKAGE;

    /* Setup DMA and initiate I2C read */
    DL_DMA_setSrcAddr(DMA, DMA_CH0_CHAN_ID, (uint32_t)(&I2C_0_INST->i2cc->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);

    DL_I2CC_startTransfer(I2C_0_INST, I2C_ADDRESS, DL_I2CC_DIRECTION_RX, USB_FRAME_PACKAGE);

    i2cInProgress = true;
}
```

```
/*********************************************************
* Function name  : i2c2usb_popTask
* Description    : I2C -> 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 gI2C2USBTask)
* Return         : None
*********************************************************/
void i2c2usb_popTask(void)
{
    if(gI2C2USBTask.fifoCount == 0) return;   // Queue empty

    TxElement_t *pTask = &gI2C2USBTask.fifoPointer[gI2C2USBTask.fifoOut];

    if(tud_cdc_n_write_available(0) >= pTask->length){  // USB FIFO has space
        uint32_t written = tud_cdc_n_write(0, pTask->buffer, pTask->length);

        if(written == pTask->length){                   // Entire task sent atomically
            tud_cdc_n_write_flush(0);
            fifoCompleteRead(&gI2C2USBTask, USB_TX_BUF_SIZE);
        }
    }
}
```

I2C Interrupt Handler:

```
//-----------------------------------------------------------------------+
// I2C Interrupt Handler
//-----------------------------------------------------------------------+
void I2C_0_INST_IRQHandler(void) {
    switch(DL_I2CC_getPendingInterrupt(I2C_0_INST)) {
        case DL_I2CC_IIDX_EVENT1_DMA_DONE:
            // TX: DMA transfer to I2C FIFO complete
            // IMPORTANT: Don't pop task yet - STOP condition still in progress
            DL_I2CC_clearInterruptStatus(I2C_0_INST, DL_I2CC_INTERRUPT_STOP);
            DL_I2CC_enableInterrupt(I2C_0_INST, DL_I2CC_INTERRUPT_STOP);
            break;

        case DL_I2CC_IIDX_STOP:
            // TX complete: I2C STOP condition detected - transmission finished
            // Only enabled when I2C Host write
            DL_I2CC_disableInterrupt(I2C_0_INST, DL_I2CC_INTERRUPT_STOP);
            fifoCompleteRead(&gUSB2I2CTask, I2C_TX_BUF_SIZE);
            i2cInProgress = false;
            break;

        case DL_I2CC_IIDX_EVENT2_DMA_DONE:
            // RX complete: DMA received 64 bytes from I2C slave
            fifoCompleteWrite(&gI2C2USBTask, USB_TX_BUF_SIZE);
            i2ctargetDataReady = false;  // Clear slave notification flag
            i2cInProgress = false;
            break;

        case DL_I2CC_IIDX_NACK:
            // I2C Error: Slave did not acknowledge address
            i2cInProgress = false;
            __BKPT(0);  // Halt for debugging - indicates I2C communication failure
            break;

        default:
            break;
    }
}
```

Overflow Handling:

```
void tud_cdc_rx_cb(uint8_t itf)
{
    if (gUSB2I2CTask.fifoCount < I2C_TX_BUF_SIZE) {
        usb2i2c_pushTask();
    } else {
        /* Queue full: discard to unblock TinyUSB RX buffer */
        uint8_t dummyBuf[64];
        tud_cdc_n_read(itf, dummyBuf, 64);
    }
}
```

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. usb2i2c_pushTask()
   b. Queue [H, E, L, L, O] with length = 5
3. Main loop: usb2i2c_popTask()
   a. Get task from queue
   b. I2C write: address=0x48, data=[H,E,L,L,O]
   c. DMA transfers to I2C TX FIFO
4. I2C Hardware executes
   a. START + 0x48 (write) + H + E + L + L + O + STOP
   b. Triggers I2C interrupt
5. I2C_0_INST_IRQHandler()
   a. EVENT1_DMA_DONE: DMA finished
   b. STOP Condition: Data transmitted
   c. fifoCompleteRead() marks task consumed

Scenario B: Host Reads 64 Bytes from Target

1. I2C target has data, pulls GPIO high
2. GPIO interrupt fires
   a. i2ctargetDataReady = true
3. Main loop detects: i2c2usb_pushTask()
   a. I2C read: address=0x48, 64 bytes expected
   b. DMA setup: I2C RX → buffer
   c. Start I2C read transfer
4. I2C Hardware executes
   a. START + 0x48 (read) + [64 bytes] + STOP
   b. All bytes go to DMA destination
   c. Triggers I2C interrupt
5. I2C_0_INST_IRQHandler()
   a. EVENT2_DMA_DONE: All data received
   b. fifoCompleteWrite(&gI2C2USBTask), and mark ready for USB transmission
6. Main loop: i2c2usb_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

LaunchPad™ is a trademark of Texas Instruments.
Windows® is a registered trademark of Microsoft Corporation.
macOS® is a registered trademark of Apple Inc..
Linux® is a registered trademark of Linus Torvalds.
All trademarks are the property of their respective owners.

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