Subsystem Design

USB to UART Bridge

1 Description

This subsystem demonstrates how to implement MSPM0 devices as a USB Communications Device Class - Abstract Control Model (USB CDC-ACM) to Universal Asynchronous Receiver-Transmitter (UART) 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 UART interface and UART device data back to the USB host. 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.

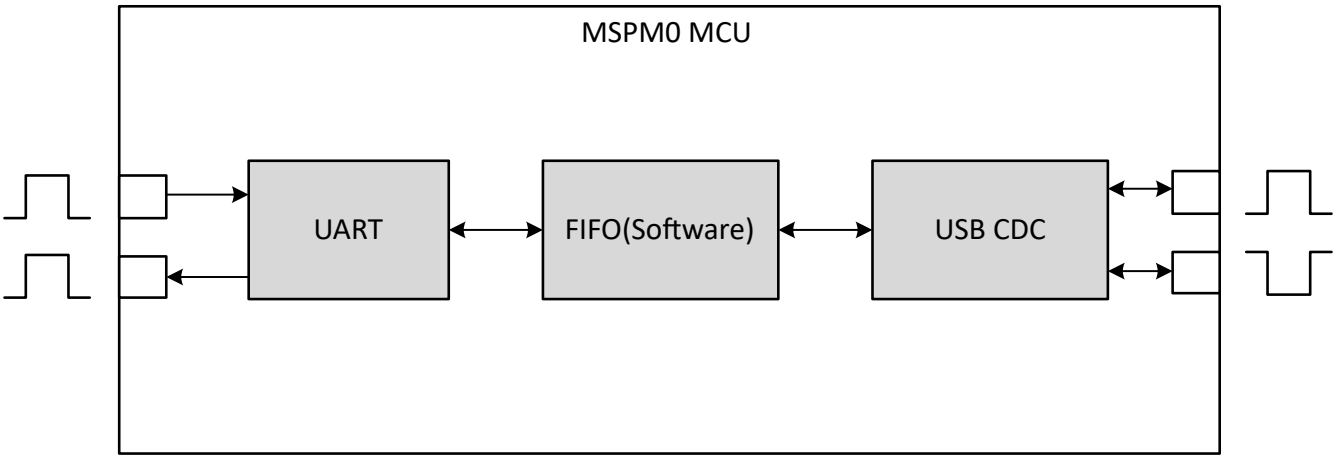

Figure 1-1. System Functional Block Diagram

2 Required Peripherals

Both platforms use the same peripheral types. SysConfig assigns device-specific instances but generates consistent macro names (UART_0_INST, TIMER_0_INST, DMA_CH0_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)
UART	Bridge to target UART device	UC0 (UART_0_INST)	UC16 (UART_0_INST)
DMA	UART TX acceleration	DMA_CH0	DMA_CH0
Timer	UART RX inter-frame timeout	TIMA0 (TIMER_0_INST)	TIMG0 (TIMER_0_INST)

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 UART module in SysConfig.
   a. Set Target Baud Rate to 115200.
   b. Enable the FIFO. Set the RX FIFO threshold to DL_UART_RX_FIFO_LEVEL_ONE_ENTRY (interrupt on every received byte). Set the TX FIFO threshold to DL_UART_TX_FIFO_LEVEL_ONE_ENTRY.
   c. In the Interrupt Configuration tab, enable the RX interrupt and the DMA Done TX interrupt. In the DMA Configuration tab, add a TX DMA channel (DMA_CH0) configured for Block addr. to Fixed addr. mode. And source length is byte
4. Set up the Timer module in SysConfig.
   a. Add one timer instance (TIMER_0). Select LFCLK as the clock source and enable the ZERO interrupt.
   b. Timeout period is set in the software.

```
// ===== UART RX Timeout =====
// Timeout period: (UART_TIMEOUT+1)/32768 ≈ 200us
// Triggers when no new UART byte arrives for 200us
#define UART_TIMEOUT        6
```

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 1 ms period. TinyUSB's board_millis() requires this 1 ms 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 / UART_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→UART queue
#define UART_TX_BUF_SIZE    8              // Max tasks in UART→USB queue

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

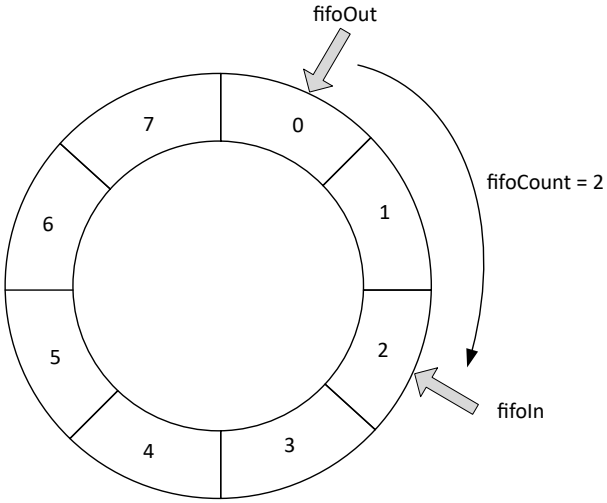

Figure 5-1. Buffer Structure

```
// 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. *UART RX timeout mechanism:* The bridge accumulates UART bytes into a 64-byte task buffer. At 64 bytes, the bridge forwards the task immediately. If the UART goes idle with fewer than 64 bytes buffered, the 200µs timer fires and flushes the partial buffer, keeping forwarding latency low.
3. *DMA for UART TX:* DMA eliminates byte-by-byte ISR overhead for UART transmission on the USB-to-UART path. The DMA Done TX interrupt fires when the DMA loads data into the UART TX FIFO, not when the last byte leaves the pin. Enable the EOT interrupt after each DMA completion to confirm full transmission before starting the next task.
4. *Overflow protection:* A full FIFO queue causes the bridge to discard incoming data. On the USB-to-UART path, the bridge reads and discards from the TinyUSB RX buffer to prevent stalling. On the UART-to-USB path, it reads and discards from the UART RX register to clear the interrupt. For production designs, add hardware flow control (UART RTS/CTS) or USB CDC line-state flow control.
5. *Baud rate configuration:* The default baud rate is 115200 for both platforms. To change it, update the Target Baud Rate field in the SysConfig UART tab. SysConfig shows the achievable baud rate below the target field. Ensure the UART clock source provides sufficient frequency for the target baud rate. The clock divider setting may differ between platforms.

6 Software Flowchart

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

USB to UART direction:

1. TinyUSB invokes tud_cdc_rx_cb() each time the USB host sends data. The callback calls usb2uart_pushTask(), which reads up to 64 bytes from the USB CDC interface and queues them as a task in gUSB2UARTTask.
2. In the main loop, when gUSB2UARTTask.fifoCount > 0, usb2uart_popTask() configures the DMA channel with the task buffer address and length, then starts the transfer to the UART TX FIFO.
3. The DMA Done TX interrupt fires when the DMA finishes loading the FIFO. The handler enables the EOT interrupt. When EOT fires, the handler clears uartTxInProgress and the next task can proceed.

UART to USB direction:

1. Each byte received from the UART device triggers the UART RX interrupt, which calls uart2usb_pushTask() to accumulate the byte and restart the 200µs timeout timer.
2. When the task buffer reaches 64 bytes, the bridge calls fifoCompleteWrite() immediately.
3. When the timeout expires with fewer than 64 bytes buffered, the timer ISR calls fifoCompleteWrite() to flush the partial frame.
4. In the main loop, when gUART2USBTask.fifoCount > 0, uart2usb_popTask() writes the buffered data to the USB CDC TX FIFO and flushes it to the host.

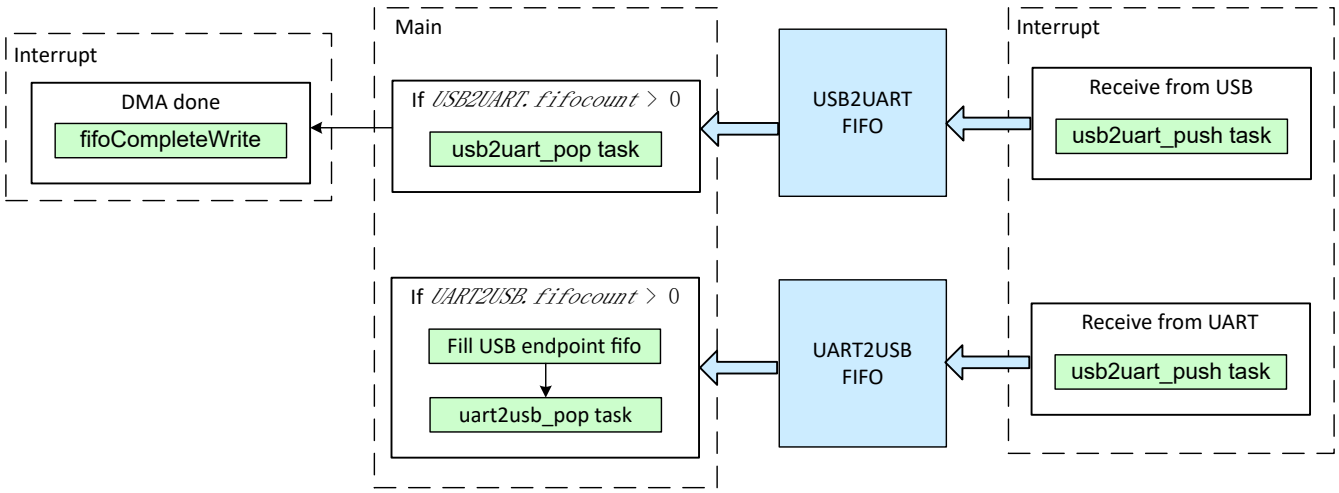

Figure 6-1. Application Software Flowchart

7 Application Code

USB to UART:

```
/*******************************************************
* Function name  : usb2uart_popTask
* Description    : USB -> UART. Pop task from queue and transmit data through
*                  UART using DMA transfer. Waits if UART TX already in progress.
*                  Called from main loop when fifoCount > 0.
* Parameter      : None (modifies gUSB2UARTTask, uartTxInProgress)
* Return         : None
*******************************************************/
void usb2uart_popTask(void)
{
    if(uartTxInProgress) return;

    //Current process task
    TxElement_t *pTask = &gUSB2UARTTask.fifoPointer[gUSB2UARTTask.fifoOut];

    //Transmit data through UART DMA
    DL_DMA_setSrcAddr(DMA, DMA_CH0_CHAN_ID, (uint32_t) pTask->buffer);
    DL_DMA_setDestAddr(DMA, DMA_CH0_CHAN_ID, (uint32_t)(&UART_0_INST->uart->TXDATA));
    DL_DMA_setTransferSize(DMA, DMA_CH0_CHAN_ID, pTask->length);
    DL_DMA_enableChannel(DMA, DMA_CH0_CHAN_ID);
    uartTxInProgress = true;

    //Update Task status
    fifoCompleteRead(&gUSB2UARTTask, UART_TX_BUF_SIZE);
}
```

```
/*******************************************************
* Function name  : usb2uart_pushTask
* Description    : USB -> UART. Read data from USB CDC interface and push
*                  into gUSB2UARTTask queue. Each task holds one USB frame
*                  (up to 64 bytes). Called from tud_cdc_rx_cb interrupt.
* Parameter      : None (modifies gUSB2UARTTask)
* Return         : None
*******************************************************/
void usb2uart_pushTask(void)
{
    // Get current task slot in queue
    TxElement_t *pTask = &gUSB2UARTTask.fifoPointer[gUSB2UARTTask.fifoIn];

    // Read up to 64 bytes from USB CDC interface (interface 0 is the first/only CDC interface)
    uint32_t cnt = tud_cdc_n_read(0,
                                  pTask->buffer,
                                  USB_FRAME_PACKAGE);

    // Skip if no data received
    if(cnt == 0) return;

    pTask->length = cnt;                    // Store actual data length

    // Advance queue pointer and mark task as ready
    fifoCompleteWrite(&gUSB2UARTTask, UART_TX_BUF_SIZE);
}
```

UART to USB:

```
/*******************************************************
* Function name  : uart2usb_pushTask
* Description    : UART -> USB. Read one byte from UART and accumulate into
*                  current task buffer. When buffer full (64 bytes) or when
*                  timeout triggers, move to next task. Called from UART RX
*                  interrupt handler.
* Parameter      : None (modifies gUART2USBTask, TIMER_0_INST)
* Return         : None
*******************************************************/
void uart2usb_pushTask(void)
{
    uint8_t byte = DL_UART_Main_receiveData(UART_0_INST);

    // Get current task buffer
    TxElement_t *pTask = &gUART2USBTask.fifoPointer[gUART2USBTask.fifoIn];

    // Check if current task buffer is full (64 bytes)
    if (pTask->length < USB_FRAME_PACKAGE - 1) {
        // Buffer not full: add byte and restart timeout timer
        pTask->buffer[pTask->length++] = byte;

        // Restart timeout to wait for more data
        DL_Timer_setTimerCount(TIMER_0_INST, UART_TIMEOUT);
        DL_Timer_startCounter(TIMER_0_INST);
    }
    else if(pTask->length == USB_FRAME_PACKAGE - 1){
        // Adding last byte (position 63): task becomes full
        pTask->buffer[pTask->length++] = byte;

        // Move to next task since buffer is now full (64 bytes)
        fifoCompleteWrite(&gUART2USBTask, USB_TX_BUF_SIZE);
    }
    else {
        // Error: buffer overflow (should never happen if logic is correct)
        __BKPT(0);
    }
}
```

```
/*******************************************************
* Function name  : uart2usb_popTask
* Description    : UART -> 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 gUART2USBTask)
* Return         : None
*******************************************************/
void uart2usb_popTask(void)
{
    // Get current task
    TxElement_t *pTask = &gUART2USBTask.fifoPointer[gUART2USBTask.fifoOut];

    // Wait for USB TX FIFO to be completely empty before sending next task
    // This ensures clean task boundaries and clear completion points
    if(tud_cdc_n_write_available(0) >= pTask->length){
        // FIFO has enough space for entire task
        uint32_t written = tud_cdc_n_write(0, pTask->buffer, pTask->length);

        // Only advance queue if entire task was written in one go
        if(written == pTask->length){
            tud_cdc_n_write_flush(0);
            fifoCompleteRead(&gUART2USBTask, USB_TX_BUF_SIZE);
        }
    }
}
```

8 Data Flow Examples

Scenario A: Host Writes 5 Bytes to Device

1. Host sends *HELLO* via COM port
2. tud_cdc_rx_cb() queues data
3. Main loop: usb2uart_popTask() transmits via UART DMA
4. UART hardware sends bytes with start/stop bits
5. Device receives *HELLO* successfully

Scenario B: Device Sends Variable Length Data

1. UART device sends *DATA* (4 bytes)
2. Each byte triggers UART RX interrupt
3. After 4th byte, 200us timeout fires
4. uart2usb_pushTask() queues the 4-byte frame
5. Main loop: uart2usb_popTask() sends to USB
6. Host receives 4 bytes via COM port

Scenario C: Fast Burst Data (64 Bytes)

1. UART device sends rapid stream
2. Bytes accumulate in FIFO
3. At 64 bytes, buffer-full triggers queuing
4. uart2usb_popTask() sends full 64-byte frame to USB
5. Host receives complete block

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