Subsystem Design

I2C to UART Subsystem Design

Design Description

This subsystem serves as an I2C-to-UART bridge. In this subsystem, the MSPM0 device is the I2C target device. When the I2C controller transmits to the I2C target, the target collects all of the received data. Once the target detects a stop condition, the target transmits the data out using the UART interface. When the I2C controller attempts to read from the bridge, the bridge transmits the last byte received from the UART device. When the I2C controller reads two bytes, the bridge transmits the last byte received from the UART device and the latest error code generated by the bridge.

The MSPM0 is connected to the I2C controller with the I2C SCL and SDA lines. The MSPM0 is also connected to a UART device using the UART TX and RX lines.

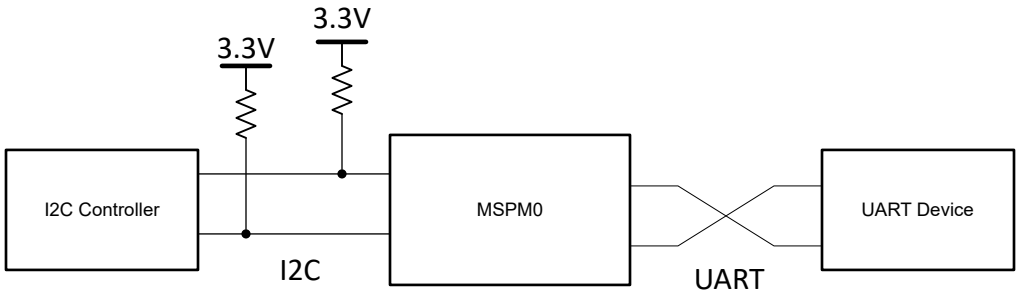

Figure 1-1. System Functional Block Diagram

Required Peripherals

Peripheral Used	Notes
I2C	Called I2C_INST in code
UART	Called UART_INST in code

Design Steps

1. Set up the I2C module in SysConfig. Set the device in Target Mode, and enable the RX FIFO Trigger, Start Detection, Stop Detection, Target Arbitration Lost, TX FIFO Underflow, RX FIFO Overflow, and Interrupt Overflow Interrupts.
2. Set up the UART module in SysConfig. Choose the desired baud rate for the device. Enable the Receive, Transmit, Overrun error, Break error, Frame error, Parity error, Noise error, and RX Timeout.

Design Considerations

1. In the application code, make sure that I2C_MAX_PACKET_SIZE is large enough to contain the packets to be transmitted.
2. Make sure to select the appropriate pullup resistor values for the I2C module being used. As a general guideline, 10 kΩ is appropriate for 100 kHz. Higher I2C bus rates require lower valued pullup resistors. For 400-kHz communications, use resistors closer to 4.7 kΩ.
3. To increase the UART baud rate, adjust the value in the SysConfig UART tab labeled *Target Baud Rate*. Below this, observe the Calculated baud rate change to reflect the target baud rate. This is calculated using the available clocks and dividers.
4. Check error flags and handle them appropriately. The UART and I2C peripherals are both capable of throwing informative error interrupts. For easy debugging this subsystem uses an enum and a global variable to save error codes when error codes are thrown. In real-world applications, handle errors in the code so the errors do not break down the project.

Software Flowchart

Figure 1-2 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 via UART.

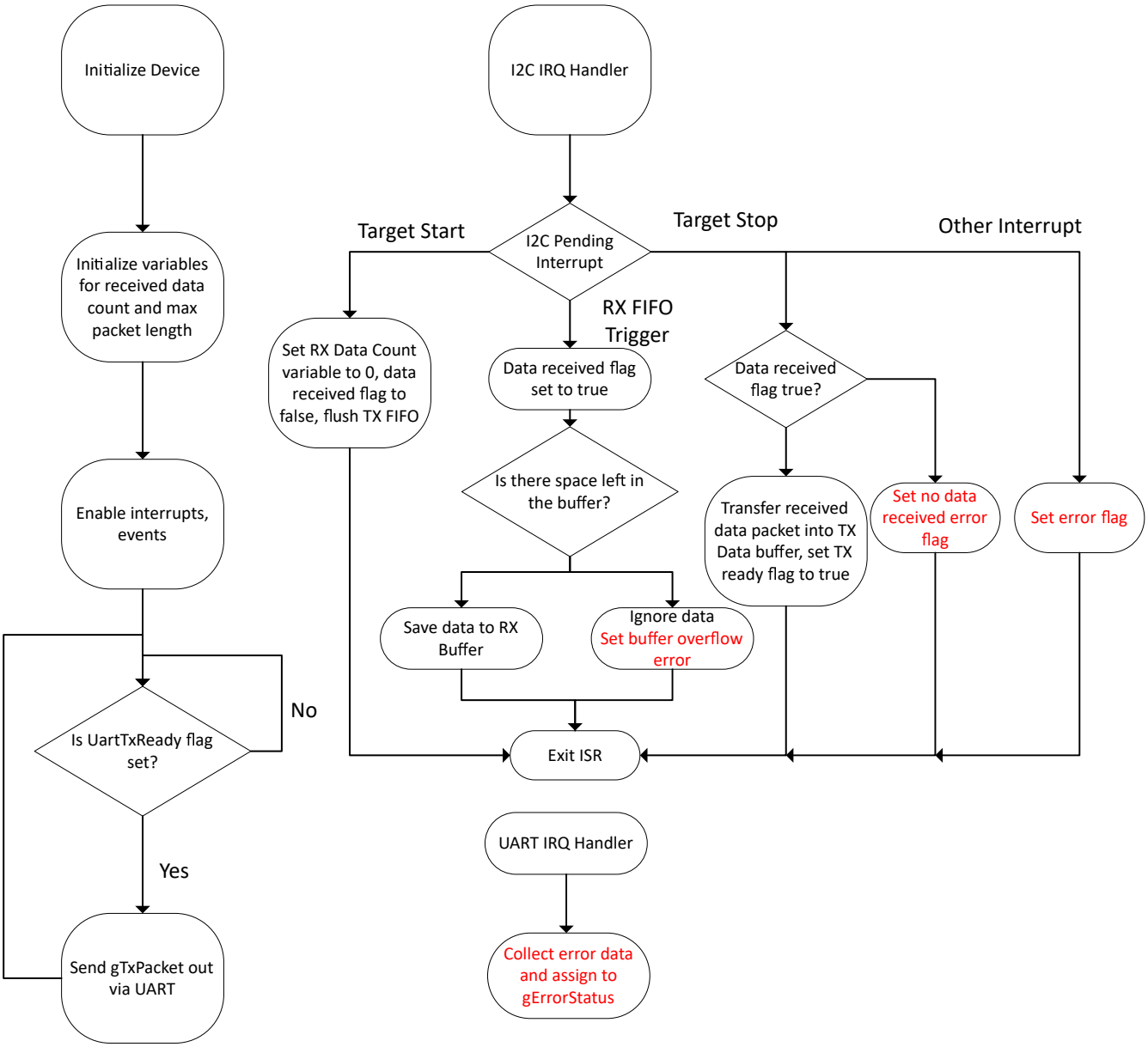

Figure 1-2. Application Software Flowchart

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 1-2 is found in the beginning of main() in the *i2c_to_uart_bridge.c* file.

Application Code

This application must allocate memory for the received data and the data to be transmitted out. The application also needs to keep count for how much data was received and transmitted. A flag is necessary to determine when the data being received is completed and ready to transmit out via UART. There is also an enum for error codes, along with a variable to save them. The initialization of the buffers, counters, enum, and flag are shown here:

```
#include "ti_msp_dl_config.h"

/* Maximum size of TX packet */
#define I2C_TX_MAX_PACKET_SIZE (1)

/* Maximum size of RX packet */
#define I2C_RX_MAX_PACKET_SIZE (16)

/* Data sent to Controller in response to Read transfer */
uint8_t gTxPacket[I2C_TX_MAX_PACKET_SIZE] = {0x00};

/* Counters for TX length and bytes sent */
uint32_t gTxLen, gTxCount;

/* Data received from Controller during a Write transfer */
uint8_t gRxPacket[I2C_RX_MAX_PACKET_SIZE];
/* Counters for TX length and bytes sent */
uint32_t gRxLen, gRxCount;

enum error_codes{
    NO_ERROR,
    DATA_BUFFER_OVERFLOW,
    RX_FIFO_FULL,
    NO_DATA_RECEIVED,
    I2C_TARGET_TXFIFO_UNDERFLOW,
    I2C_TARGET_RXFIFO_OVERFLOW,
    I2C_TARGET_ARBITRATION_LOST,
    I2C_INTERRUPT_OVERFLOW,
    UART_OVERRUN_ERROR,
    UART_BREAK_ERROR,
    UART_PARITY_ERROR,
    UART_FRAMING_ERROR,
    UART_RX_TIMEOUT_ERROR
};

uint8_t gErrorStatus = NO_ERROR;

/* Buffer to hold data received from UART device */
uint8_t gUARTRxData = 0;
/* Flags */
bool gUartTxReady = false;  /* Flag to start UART transfer */
bool gUartRxDone = false;   /* Flag to indicate UART data has been received */
```

The main body of the application code is relatively short. First the device and the peripherals are initialized. Then interrupts and events are enabled. The counter values are also initialized. Finally, the main loop is reached, where a flag is polled detect when the received data is ready to be transferred back out via UART:

```
int main(void)
{
    SYSCFG_DL_init();

    gTxCount = 0;
    gTxLen   = I2C_TX_MAX_PACKET_SIZE;
    DL_I2C_enableInterrupt(I2C_INST, DL_I2C_INTERRUPT_TARGET_TXFIFO_TRIGGER);

    /* Initialize variables to receive data inside RX ISR */
    gRxCount = 0;
    gRxLen   = I2C_RX_MAX_PACKET_SIZE;

    NVIC_EnableIRQ(I2C_INST_INT_IRQN);
    NVIC_EnableIRQ(UART_INST_INT_IRQN);

    while (1) {
        if(gUartTxReady){
            gUartTxReady = false;
            for(int i = 0; i < gRxCount; i++){
```

```
                /* Transmit data out via UART and wait until transfer is complete */
                DL_UART_Main_transmitDataBlocking(UART_INST, gTxPacket[i]);
            }
        }
    }
}
```

The next piece of this code is the I2C IRQ Handler. This code is used to start and then stop data collection. Next, the code saves the data as the data is received. When the pending interrupt is an I2C Start condition detected, the device initializes the counter variables. When the pending interrupt indicates that the RX FIFO has data available, the device checks to see if there is space left in the data buffer. If there is space, the received value is saved. If there is not any more space, the received value is ignored. When the pending interrupt is the TX FIFO Trigger, the device checks to see how many bytes have been sent. If the device has already sent a byte, then the FIFO is filled with the most recently reported error code. When the pending interrupt is an I2C stop condition, the device checks to see if data was received. If data was received, the received data buffer is copied into the transmit data buffer, and the UART TX ready flag is set to true. If no data was received, the device does not send anything. This ISR also handles I2C error interrupts by assigning the appropriate error code to the gErrorStatus variable.

```
void I2C_INST_IRQHandler(void)
{
    static bool dataRx = false;

    switch (DL_I2C_getPendingInterrupt(I2C_INST)) {
        case DL_I2C_IIDX_TARGET_START:
            /* Initialize RX or TX after Start condition is received */
            gTxCount = 0;
            gRxCount = 0;
            dataRx   = 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 */
            dataRx = true;
            while (DL_I2C_isTargetRXFIFOEmpty(I2C_INST) != true) {
                if (gRxCount < gRxLen) {
                    gRxPacket[gRxCount++] = DL_I2C_receiveTargetData(I2C_INST);
                } else {
                    /* Prevent overflow and just ignore data */
                    DL_I2C_receiveTargetData(I2C_INST);
                }
            }
            break;
        case DL_I2C_IIDX_TARGET_TXFIFO_TRIGGER:
            /* Fill TX FIFO if there are more bytes to send */
            if (gTxCount < gTxLen) {
                gTxCount += DL_I2C_fillTargetTXFIFO(
                    I2C_INST, &gUARTRxData, (gTxLen - gTxCount));
            } else {
                /*
                 * 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, echo to TX buffer */
            if (dataRx == true) {
                for (uint16_t i = 0;
                     (i < gRxCount) && (i < I2C_TX_MAX_PACKET_SIZE); i++) {
                    gTxPacket[i] = gRxPacket[i];
                    DL_I2C_flushTargetTXFIFO(I2C_INST);
                }
                dataRx = false;
            }
            /* Set flag to indicate data ready for UART TX */
            gUartTxReady = true;
            break;
        case DL_I2C_IIDX_TARGET_RX_DONE:
            /* Not used for this example */
```

```
        case DL_I2C_IIDX_TARGET_RXFIFO_FULL:
            /* Not used for this example */
        case DL_I2C_IIDX_TARGET_GENERAL_CALL:
            /* Not used for this example */
        case DL_I2C_IIDX_TARGET_EVENT1_DMA_DONE:
            /* Not used for this example */
        case DL_I2C_IIDX_TARGET_EVENT2_DMA_DONE:
            /* Not used for this example */
        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 piece of code in this example is the UART IRQ Handler. The UART IRQ handler is only used to save received data, and check for errors. When a UART RX interrupt is pending, the device saves the received data to a buffer, gUARTRxData, then sets a flag to indicate there is new RX data saved. When a UART error does occur, this ISR executes to assign the correct error code to gErrorStatus.

```
void UART_INST_IRQHandler(void)
{
    switch (DL_UART_Main_getPendingInterrupt(UART_INST)) {
        case DL_UART_MAIN_IIDX_RX:
            DL_UART_Main_receiveDataCheck(UART_INST, &gUARTRxData);
            gUartRxDone = true;
            break;
        case DL_UART_INTERRUPT_OVERRUN_ERROR:
            gErrorStatus = UART_OVERRUN_ERROR;
            break;
        case DL_UART_INTERRUPT_BREAK_ERROR:
            gErrorStatus = UART_BREAK_ERROR;
            break;
        case DL_UART_INTERRUPT_PARITY_ERROR:
            gErrorStatus = UART_PARITY_ERROR;
            break;
        case DL_UART_INTERRUPT_FRAMING_ERROR:
            gErrorStatus = UART_FRAMING_ERROR;
            break;
        case DL_UART_INTERRUPT_RX_TIMEOUT_ERROR:
            gErrorStatus = UART_RX_TIMEOUT_ERROR;
            break;
        default:
            break;
    }
}
```

Additional Resources

1. Texas Instruments, Download the MSPM0 SDK
2. Texas Instruments, Learn more about SysConfig
3. Texas Instruments, MSPM0L LaunchPad™
4. Texas Instruments, MSPM0G LaunchPad™
5. Texas Instruments, MSPM0 I2C Academy
6. Texas Instruments, MSPM0 UART Academy

1 Revision History

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

Changes from Revision * (December 2023) to Revision A (August 2025)	Page
• Removed compatible devices section	1

Trademarks

LaunchPad™ is a trademark of Texas Instruments.
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