Subsystem Design

LIN to I2C Bridge

1 Description

This subsystem functions as a LIN-to-I2C bridge utilizing an MSPM0 device. The process initiates when the bridge's LIN responder collects data from an external LIN commander; upon detecting a valid message, the data is transferred to the bridge's I²C controller and transmitted to an external I²C target device. The target device then responds with new data, and the bridge subsequently waits for a request from the external LIN commander to transmit the captured data via the bridge's LIN responder. Once the transaction is complete, the bridge resets its states to idle. In the event of an error during any stage, both the I²C and LIN interfaces exhibit specific behaviors to signal a fault to the user. Figure 1-1 illustrates a common hardware configuration used for the LIN-to-I2C subsystem.

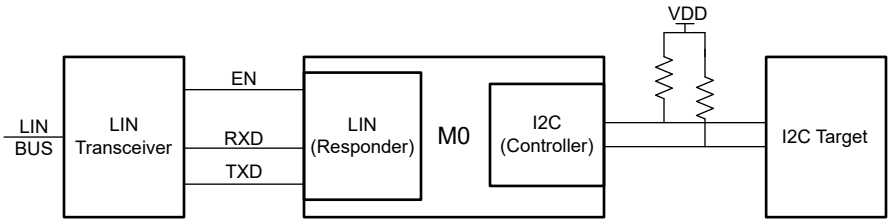

Figure 1-1. System Functional Block Diagram

2 Required Peripherals

Two MSPM0 modules are used for this subsystem: LIN and I²C.

Table 2-1. MSPM0 Modules

Sub-Block Functionality	Peripheral Use	Notes
LIN Responder	LIN	Called LIN_INST in code.
I²C Controller	I²C	Called I2C_INST in code. Default 400kHz transmission rate.

3 Compatible Devices

This subsystem is compatible with any MSPM0 with an UART-LIN module and a I²C module (whether a dedicated or a UNICOMM derived module).

4 Design Steps

1. Locate the project in the *MSPM0 SDK* (MSP Subsystems folder) and import it into your CCS Studio workspace.
2. Use lin_2_I2C.syscfg to configure both the I²C and LIN modules for your specific application.
3. Set the maximum packet size. Note: Since LIN is limited to 8 bytes per transmission, verify your implementation accounts for multi-frame data handling if transmitting larger payloads.

Note

Since LIN is limited to 8 bytes per transmission, verify your implementation accounts for multi-frame data handling if transmitting larger payloads.

5 Design Considerations

1. Hardware Selection
   a. Select a LIN transceiver that aligns with your specific application requirements, such as voltage levels, data rates, and fault protection. Texas Instruments offers an extensive *catalog of LIN transceivers* that can be used to identify the best device for your system. For this example, we use the *TLIN2029EVM* as the LIN transceiver thanks to the easy-to-follow userguide.
   b. The I²C pins (and the LIN pins, depending on the system requirements) require pull-up resistors for proper operation. Similarly, if any MSPM0 pin is configured as open-drain—such as PA0 in the current project's default configuration—a pull-up resistor is required to allow the pin to drive other components or signals.
   c. Review the power, voltage, and current requirements for all components within the system. This includes verifying the specific electrical requirements for each module, such as the I²C, UART-LIN, and GPIO interfaces.
   d. Evaluate the specifications of both the internal and external oscillators of the MSPM0 to maintain that they provide sufficient timing accuracy for your application.
2. Protocol related and Firmware Logic
   a. Adjusting external pull-up resistors in accordance with I²C specifications is necessary to maintain communication integrity if I²C speeds are increased. As a general guideline, 10kΩ is appropriate for 100kHz operation. Higher I2C bus rates require lower-valued pull-up resistors; for 400kHz communication, resistors closer to 4.7kΩ must be used.
   b. Further code optimization can be necessary to accommodate increased bridge utilization. Recommended optimizations include increasing device operating speeds, implementing multiple transfer buffers, utilizing DMA, or simplifying the state machine. Note that the current example was only tested at I²C speed of 400kHz.
   c. For more recommendations regarding LIN implementation with the MSPM0, please refer to the *LIN Basics and Implementation guide for MSPM0*.

6 Software Flowchart

Figure 6-1 shows the flow diagram of the subsystem project.

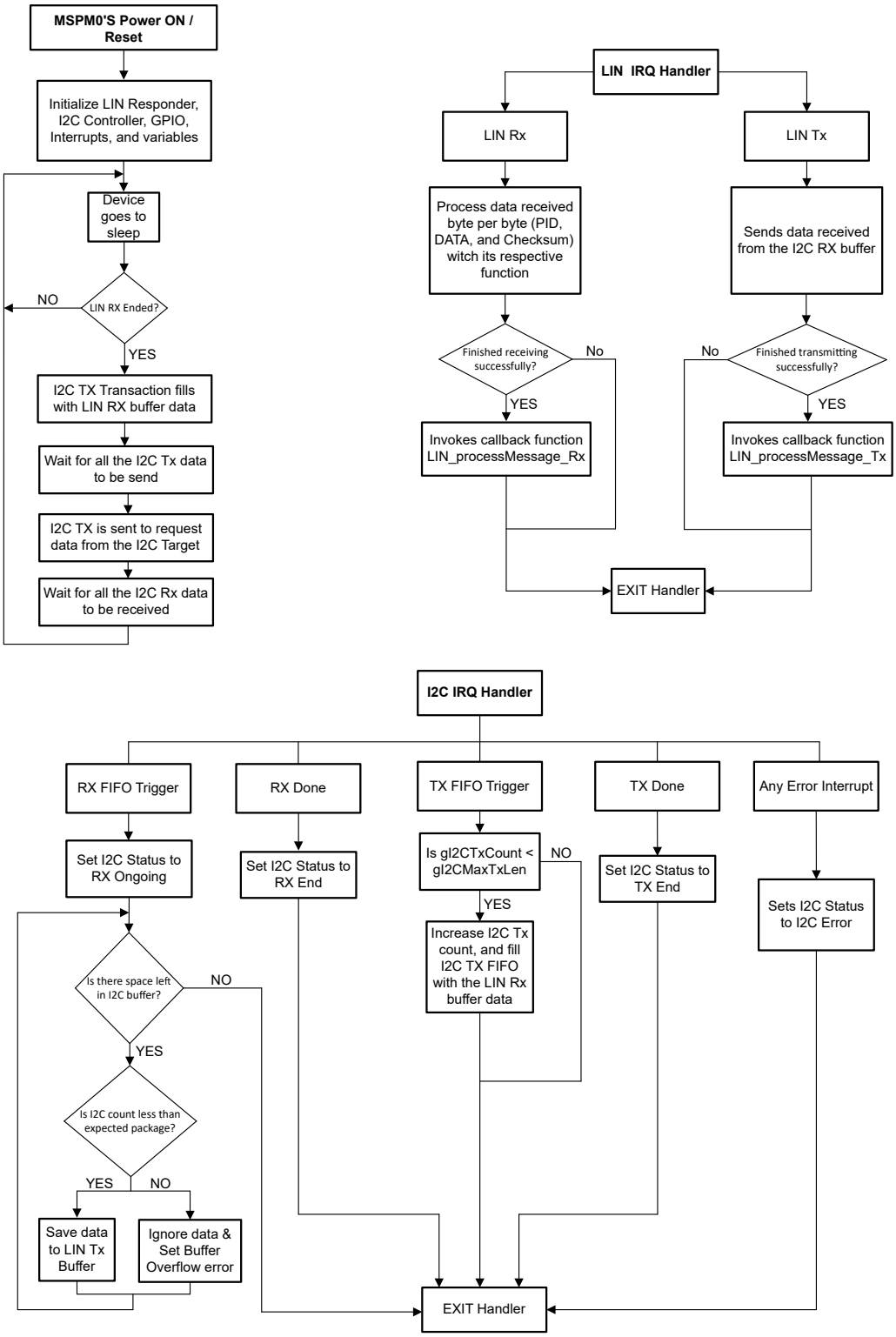

Figure 6-1. LIN-to-I2C Subsystem Software Flowchart

Figure 6-2 shows a high-level time diagram how the communication works. X bytes represent the data sent by the external LIN commander, while Y bytes represent the data being retrieved from the I²C target, and then sent back to the LIN commander.

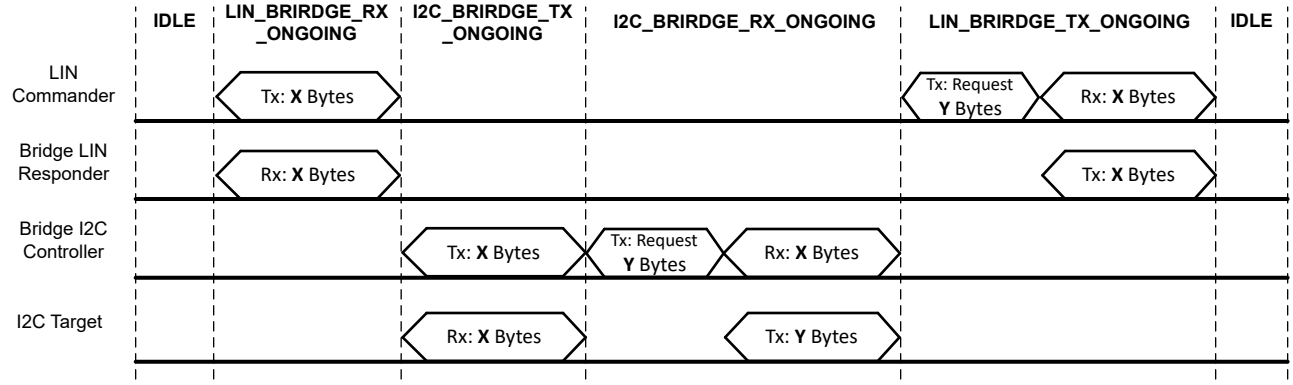

Figure 6-2. High Level Communication Diagram

7 Application Code

The base code used for the subsystem is derived from the lin_responder and i2c_controller_rw_multibyte_fifo_interrupts examples provided with the MSPM0 SDK. For a clearer understanding of both examples, please refer to their respective *README* files.

To present the bridge structure, it is important to highlight the location of each component of the implementation so users know where to make their modifications.

The *lin_config.h* file serves as the central repository for all structures and global variables used throughout the project, and it is where the bridge structure is declared to capture the various stages the bridge will occupy at specific points in its execution. A global variable labeled gBridgeState is referenced by every source file in the project and is updated as different stages of the LIN or I²C transactions occur.

```
/*! @enum BRIDGE_STATE - Bridge state machine states */
typedefenum {BRIDGE_IDLE = 0,
             BRIDGE_LIN_RX_STATE_ONGOING = 1,
             BRIDGE_LIN_RX_STATE_END = 2,
             BRIDGE_I2C_TX_STATE_ONGOING = 3,
             BRIDGE_I2C_RX_STATE_ONGOING = 4,
             BRIDGE_LIN_TX_STATE_ONGOING = 5,
} BRIDGE_STATE;
volatile BRIDGE_STATE gBridgeState;
```

The *lin_config.c* file implements the mechanisms required for LIN protocol standards on the MSPM0. Certain functions in the file have been modified to keep track of the LIN-bus states (e.g., LIN_RX and LIN_TX ongoing states).

Moving to the main file, *lin_to_I2C.c*, the lin_config.h is called to inherit all defines and structures used across the LIN and I²C modules. Afterwards, the following items are created to implement the bridge:

1. I²C local global variables *gTxLen*/*gTxCount* and *gRxLen*/*gRxCount*, which track the number of bytes scheduled and the actual number of bytes transmitted or received during an I²C transaction.
2. The variable *gI2cControllerStatus*, which captures every stage of an I²C transaction—including *IDLE*, *TX_STARTED, TX_INPROGRESS, TX_COMPLETE, RX_STARTED, RX_INPROGRESS, RX_COMPLETE*, and *ERROR*. This status is updated by the I²C ISR and polled by the main loop to maintain safe operation.

   Note

   Errors detected in the I²C flow are treated as arbitration-lost and NACK-related conditions and labeled the same. Additional error states and handling can be added as needed for a specific application.
3. The declaration of the LIN callback functions associated with each PID (useful for taking action at specific points in LIN transactions; in this example they are employed at the end of each transaction).

4. The LIN *responderMessageTable* table variable that store two types of messages: a *SUBSCRIBE* entry at ID *0x10*, where the responder receives eight bytes from the commander and executes the *LIN_processMessage_Rx* callback, and a *PUBLISH* entry at ID *0x20*, where the responder sends eight bytes and executes *LIN_processMessage_Tx* callback (in both cases, the callbacks are invoked only if the operation was successful).

Note

To maximize the general-purpose utility of the bridge, certain original LIN-Responder messages have been omitted. For comprehensive implementation details, please refer to the *lin_responder* example included in the MSPM0 SDK.

At startup, *SYSCFG_DL_init()* configures the peripherals, then the LIN-EN assigned I/O pin is set high to activate the physical bus driver, and a loop executes to find the PID values of the two messages in the LIN table based on their respective IDs.

Afterward, the table pointer is registered in the global *gLIN* variable so that the interrupt-driven LIN driver knows which PIDs to monitor, and the LIN counter-compare register is loaded with *LIN_BREAK_FIELD_MIN_CYCLES* to establish the minimum break-detection window.

Finally, pending interrupts are cleared for the modules used by the bridge (UART-LIN, timers, and I²C), the red LED that toggles at various bridge states is cleared, and both *gBridgeState* and *gI2cControllerStatus* are reset to their idle values.

The main loop operates in an interrupt-driven manner; the CPU executes *__WFI()* to enter low-power sleep and wakes only when a peripheral generates an interrupt.

The first state the bridge reaches is *BRIDGE_LIN_RX_STATE_ONGOING*, which begins when the bridge's LIN responder accepts the Subscribe message from an external LIN commander. Once the *X-bytes* message is received successfully, the *LIN_processMessage_Rx* callback is invoked, the bridge state advances to *BRIDGE_LIN_RX_STATE_END*, and the red LED toggles for one second to indicate that the end state has been reached.

Returning to the main loop after this event, the I²C sequence starts. The state is reset to *BRIDGE_IDLE*, and, if the received PID matches the Subscribe PID (signaling that a LIN message is ready for I²C transmission), the LIN receive buffer (*gLIN.rxBuf*) is copied into the I²C hardware TX FIFO. If the number of bytes can be transferred in a single burst, the TXFIFO-trigger interrupt is disabled; otherwise, the interrupt remains enabled so that the ISR can replenish the FIFO as it drains.

The system then marks the I²C status as *TX_STARTED*, sets the bridge status to *BRIDGE_I2C_TX_STATE_ONGOING*, waits for the I²C bus to become idle, and calls *DL_I2C_startControllerTransfer* to initiate a write of the previously received LIN *X-bytes* to the address defined in the I²C configuration (0x48 in this case). The CPU executes *__WFI()* in a loop until the ISR signals either *TX_COMPLETE* or *ERROR*; any error triggers a software breakpoint via *__BKPT(0)*. Upon completion, the red LED toggles again to indicate that the I²C-TX portion of the bridge operation has finished.

After a short delay, the system marks the I²C status as *RX_STARTED*, sets the bridge status to *BRIDGE_I2C_RX_STATE_ONGOING*, and issues a read request from the bridge I²C controller for *Y-bytes* from the external I²C target. The CPU again executes *__WFI()* until the ISR signals that the I²C reception has completed, at which point the red LED toggles to indicate that the I²C-RX portion of the bridge operation has finished.

Finally, an external LIN commander requests the *Y-bytes* that were retrieved from the I²C target. When the bridge's LIN responder accepts the Publish message, the bridge status changes to the final state *BRIDGE_LIN_TX_STATE_ONGOING*. After all the bytes have been transmitted successfully, the *LIN_processMessage_Tx* callback is invoked, the bridge state returns to idle, and the red LED toggles for one second to indicate that the bridge has completed its cycle. After this, the bridge status and cycle are reset to their initial values, as they are at the beginning of the program.

In the event of an error, the following behaviors are expected (the errors behavior can be modified to fit an application's requirements):

1. *LIN Error:* The LIN global variable internal error values will be set as true, and the Red LED will blink 5 times.
2. *I2C Error:* The code will stop in a software breakpoint, and the I2C status global variable will be updated to an error.

8 Results

As illustrated in the LIN to I²C transaction capture, the bridge's operation proceeds as follows:

1. The cycle is initiated by the LIN commander, which transmits the first message to the bridge.
2. The bridge forwards this message to an I²C target via the I²C bus.
3. The I²C target returns the requested data to the bridge.
4. Finally, the LIN commander requests this data, and the bridge delivers the response.

At the conclusion of each complete cycle, the RED_LED toggles (which is active low) providing a visual indication that the transfer has been successfully processed.

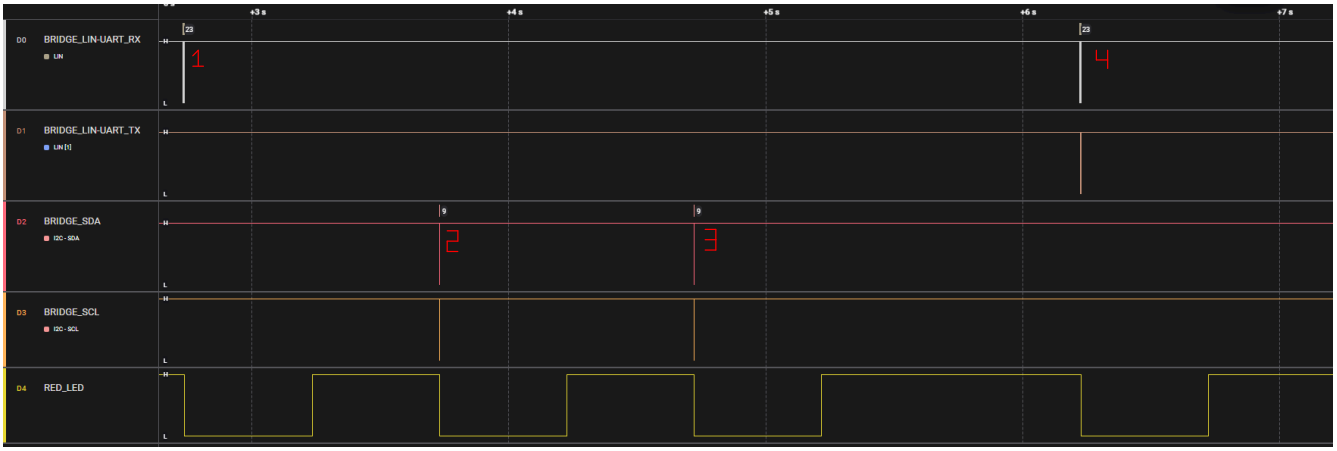

Figure 8-1. LIN to I²C transaction capture

9 Trademarks

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