Subsystem Design

CAN to UART Bridge

Design Description

This subsystem demonstrates how to build a CAN-UART bridge. CAN-UART bridge allows a device to send or receive information on one interface and receive or send the information on the other interface Download the code for this example.

Figure 1-1 shows a functional diagram of this subsystem.

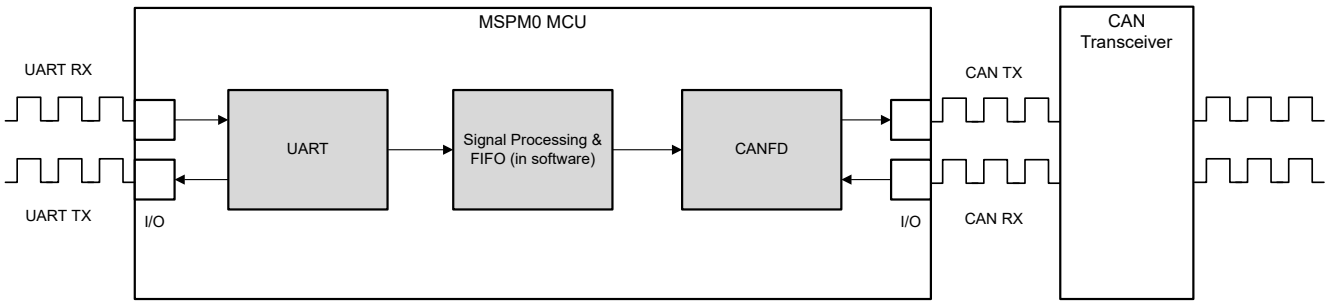

Figure 1-1. Subsystem Functional Block Diagram

Required Peripherals

This application requires CANFD and UART.

Table 1-1. Required Peripherals

Sub-block Functionality	Peripheral Use	Notes
CAN interface	(1x) CANFD	Called *MCAN0_INST* in code
UART interface	(1x) UART	Called *UART_0_INST* in code

Design Steps

1. Determine the basic setting of CAN interface, including CAN mode, bit timing, message RAM configuation and so on. Consider which setting is fixed and which setting is changed in the application. In example code, CANFD is used with 250kbit/s arbitration rate and 2Mbit/s data rate.
   a. Key features of the CAN-FD peripheral include:
      i. Dedicated 1KB message SRAM with ECC
      ii. Configurable transmit FIFO, transmit queue and event FIFO (up to 32 elements)
      iii. Up to 32 dedicated transmit buffers and 64 dedicated receive buffers. Two configurable receive FIFOs (up to 64 elements each)
      iv. Up to 128 filter elements
   b. If CANFD mode is enabled:
      i. Full support for 64-byte CAN-FD frames
      ii. Up to 8Mbit/s bit rate
   c. If CANFD mode is disabled:
      i. Full support for 8-byte classical CAN frames
      ii. Up to 1Mbit/s bit rate

2. Determine the CAN frame, including data length, bit rate switching, identifier, data and so on. Consider which part is fixed and which part need to be changed in the application. In example code, identifier, data length and data can change in different frames, while others are fixed. Note that users need to modify the code if protocol communication is required.

```
/**
 *  @brief  Structure for MCAN Rx Buffer element.
 */
typedef struct {
    /*! Identifier */
    uint32_t id;
    /*! Remote Transmission Request
     *   0 = Received frame is a data frame
     *   1 = Received frame is a remote frame
     */
    uint32_t rtr;
    /*! Extended Identifier
     *   0 = 11-bit standard identifier
     *   1 = 29-bit extended identifier
     */
    uint32_t xtd;
    /*! Error State Indicator
     *   0 = Transmitting node is error active
     *   1 = Transmitting node is error passive
     */
    uint32_t esi;
    /*! Rx Timestamp */
    uint32_t rxts;
    /*! Data Length Code
     *   0-8  = CAN + CAN FD: received frame has 0-8 data bytes
     *   9-15 = CAN: received frame has 8 data bytes
     *   9-15 = CAN FD: received frame has 12/16/20/24/32/48/64 data bytes
     */
    uint32_t dlc;
    /*! Bit Rat Switching
     *   0 = Frame received without bit rate switching
     *   1 = Frame received with bit rate switching
     */
    uint32_t brs;
    /*! FD Format
     *   0 = Standard frame format
     *   1 = CAN FD frame format (new DLC-coding and CRC)
     */
    uint32_t fdf;
    /*! Filter Index */
    uint32_t fidx;
    /*! Accepted Non-matching Frame
     *   0 = Received frame matching filter index FIDX
     *   1 = Received frame did not match any Rx filter element
     */
    uint32_t anmf;
    /*! Data bytes.
     *   Only first dlc number of bytes are valid.
     */
    uint16_t data[DL_MCAN_MAX_PAYLOAD_BYTES];
} DL_MCAN_RxBufElement;
```

3. Determine the basic setting of UART interface, including UART mode, baud rate, word length, FIFO and so on. Consider which setting is fixed and which setting is changed in the application. In example code, UART is used with 9600 baud rate.
   a. Key features of the UART peripheral include:
      i. Standard asynchronous communication bits for start, stop, and parity
      ii. Fully programmable serial interface
      iii. Separated transmit and receive FIFOs support DAM data transfer
      iv. Support transmit and receive loopback mode operation

4. Determine the UART frame. Typically UART is transmitted in bytes. To achieve high-level communication, users can implement frame communication through software. If necessary, users can also introduce specific communication protocols. In example code, the message format is < 55 AA ID1 ID2 ID3 ID4 Length Data1 Data2 ...>. Users can send data to the CAN bus from the terminal by entering data as the same format. 55 AA is the header. ID area is 4 bytes. Length area is 1 byte, indicating the data length. Note that if users need to modify the UART frame, the code for frame acquisition and parsing also need to be modified.

Table 1-2. UART Frame Form

Header	Address	Data Length	Data
0x55 0xAA	4 bytes	1 byte	(Data Length) bytes

5. Determine the bridge structure, including what messages need to be converted, how to convert messages and so on.
   a. Consider whether the bridge is one-way or two-way. Typically each interface has two functions: receiving and sending. Consider whether only some functions need to be included (such as UART reception and CAN transmission). In example code, CAN-UART bridge is a two-way structure.
   b. Consider what information to convert and the corresponding carrier(variable, FIFO). In example code, identifier, data and data length are convert from one interface to the other interface. There are two FIFOs defined in code as shown below.

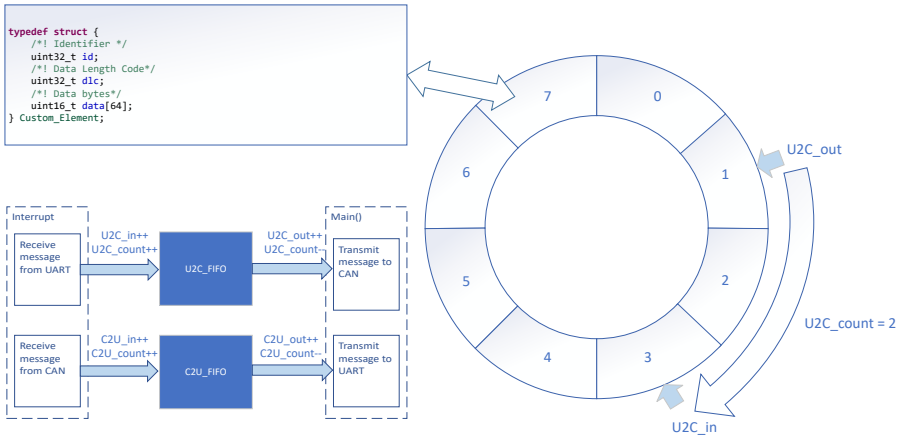

Figure 1-2. Bridge structure

6. (Optionally) Consider priority design, congestion situation, error handling, and so on.

Design Considerations

1. Consider the information flow in the application to determine the information to be received or sent by each interface, the protocols to be followed, and design appropriate information transfer carriers to connect different interfaces.
2. The recommendation is to test the interface separately first, and then implement the overall bridge function. In addition, consider the handling of abnormal situations, such as communication failure, overload, frame format error, and so on.
3. The recommendation is to implement interface functions through interrupts to make sure of timely communication. In example code, interface functions are usually implemented in the interrupt, and the transfer of information is completed in the main() function.

Software Flowchart

The following figure shows the code flow diagram for *CAN-UART bridge* which explains how the messages received in one interface and sent in the other interface. The *CAN-UART bridge* can be divided into four independent tasks: receive from UART, receive from CAN, transmit through CAN, transmit through UART. Two FIFOs implement bidirectional message transfer and message caching.

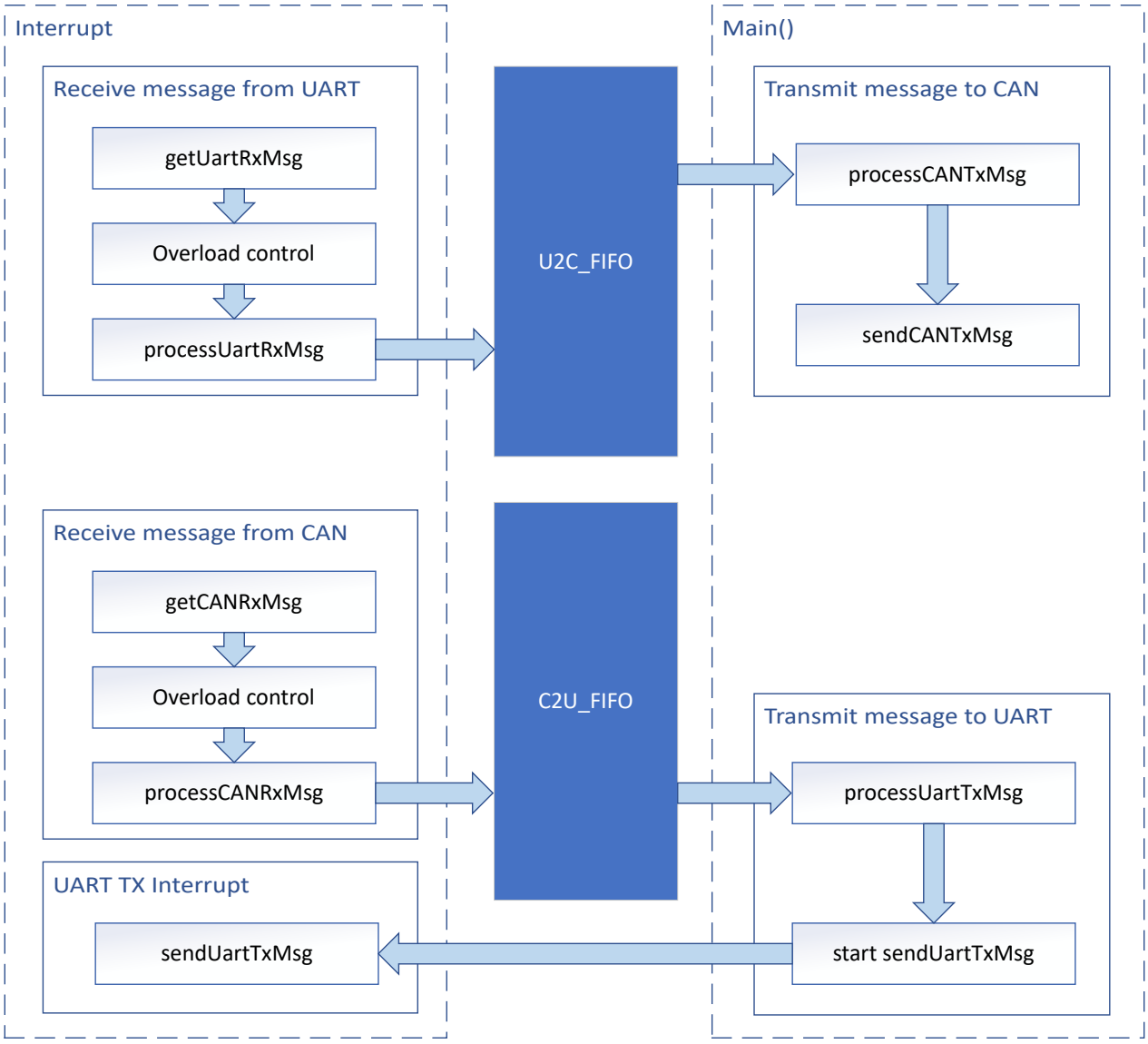

Figure 1-3. Application Software Flowchart

Device Configuration

This application makes use of TI System Configuration Tool (SysConfig) graphical interface to generate the configuration code for the CAN and UART. Using a graphical interface to configure the device peripherals streamlines the application prototyping process.

The code for what is described in Figure 1-3 can be found in the files from example code as shown in Figure 1-4.

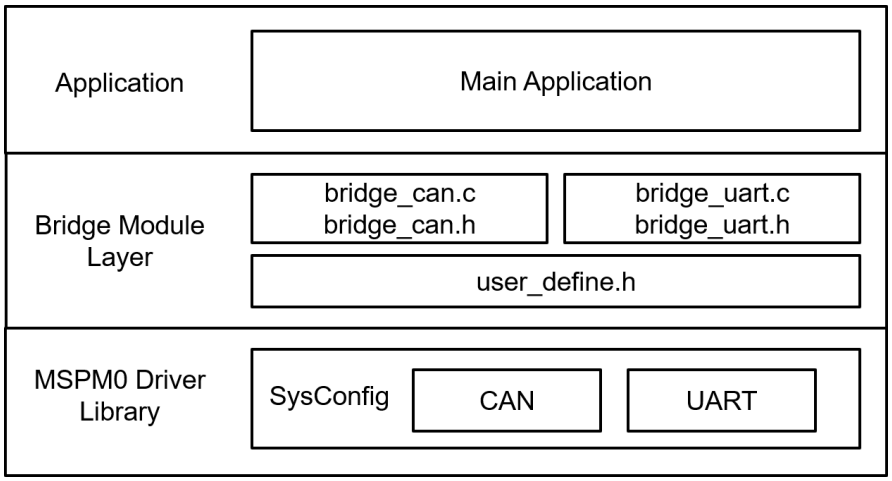

Figure 1-4. File Structure

Application Code

The following code snippet shows where to modify the interface function. Functions in table are categorized into different files. Functions for UART receive and transmit are included in bridge_uart.c and bridge_uart.h. Functions for CAN receive and transmit are included in bridge_can.c and bridge_can.h. Structure of FIFO element is defined in user_define.h.

Users can easily separate functions by file. For example, if only UART functions are needed, users can reserve bridge_uart.c and bridge_uart.h to call the functions.

See the MSPM0 SDK and DriverLib documentation for the basic configuration of peripherals.

Table 1-3. Functions and Descriptions

Tasks	Functions	Description	Location
UART receive	getUartRxMsg()	Get the received UART message	bridge_uart.c bridge_uart.h
	processUartRxMsg()	Convert the received UART message format and store the message into gUART_RX_Element	
UART transmit	processUartTxMsg()	Convert the gUART_TX_Element format to be sent through UART	
	sendUartTxMsg()	Send message through UART	
CAN receive	getCANRxMsg()	Get the received CAN message	bridge_can.c bridge_can.h
	processCANRxMsg()	Convert the received CAN message format and store the message into gCAN_RX_Element	
CAN transmit	processCANTxMsg()	Convert the gCAN_TX_Element format to be sent through CAN	
	sendCANTxMsg()	Send message through CAN	

Custom_Element is the structure defined in user_define.h. Custom_Element is used as the structure of FIFO element, output element of UART/CAN transmit and input element of UART/CAN receive. Users can modify the structure according to the need.

```
typedef struct {
    /*! Identifier */
    uint32_t id;
    /*! Data Length Code*/
    uint32_t dlc;
    /*! Data bytes*/
    uint16_t data[64];
} Custom_Element;
```

For FIFO, there are 2 global variables used as FIFO. 6 global variables are used to trace the FIFO.

```
Custom_Element U2C_FIFO[U2C_FIFO_SIZE];
Custom_Element C2U_FIFO[C2U_FIFO_SIZE];
uint16_t U2C_in = 0;
uint16_t U2C_out = 0;
uint16_t U2C_count = 0;
uint16_t C2U_in = 0;
uint16_t C2U_out = 0;
uint16_t C2U_count = 0;
```

Results

By using the XDS110 on the launchpad, users can use the PC to send and receive messages on the UART side. As a demonstration, two launchpads can be used as two CAN-UART bridges to form a loop. When the PC sends UART messages through one of the launchpads, XDS110 can receive UART messages from the other launchpad.

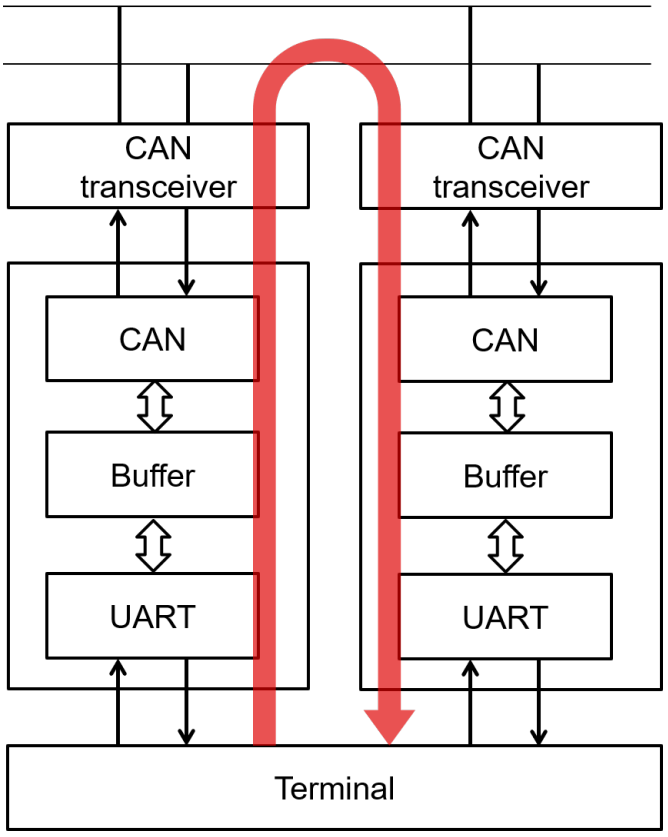

Figure 1-5. Demonstration

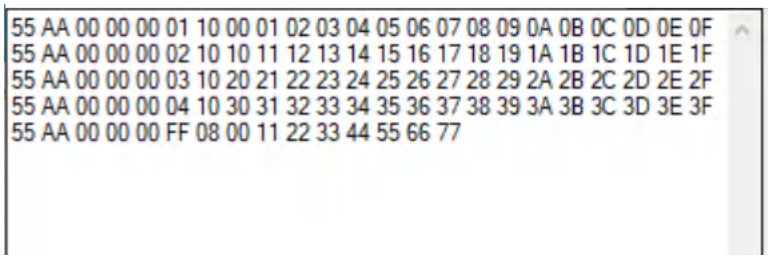

Figure 1-6. PC Terminal Program

Additional Resources

- Download the MSPM0 SDK
- Learn more about SysConfig
- MSPM0G Technical Reference Manual (TRM)
- MSPM0G LaunchPad development kit
- MSPM0 CAN academy
- 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

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