Application Note

Bridge Design between CAN and UART with MSPM0 MCUs

Yuhao Zhao

ABSTRACT

This application note introduces a CAN to UART bridge. The document describes the structure and behavior of a CAN to UART bridge. This document details software implementation, hardware connection, and application usage. Users can configure the bridge by modifying the predefine. Relevant code is also provided.

Table of Contents

List of Figures

List of Tables

Trademarks

All trademarks are the property of their respective owners.

1 Introduction

Based on different applications, there are many communication methods between devices. MCUs today usually support more than one communication method. For example, MSPM0 can support UART, SPI, CAN, and so on for a specific device. When devices transfer data over different communication interfaces, a bridge is constructed.

For CAN and UART, a CAN-UART bridge acts as a translator between the two interfaces. CAN-UART bridge allows a device to send and receive information on one interface and receive and send the information on the other interface.

This application note describes the software and hardware designs used in creating and using the CAN-UART bridge. The MSPM0G3507 microcontroller (MCU) can be used as a solution by providing CAN and UART communication interfaces. The accompanying demo uses the MSPM0G3507 with 2Mbps CANFD and 9600 baud rate UART to demonstrate transceiving data between channels.

1.1 Bridge Between CAN and UART

The CAN-UART bridge connects the CAN and UART interfaces. The example in this article can rely on the XDS110 on the launchpad to observe the UART data with a PC. User can also send messages from a PC over the CAN-UART bridge to the CAN bus. For CAN bus data, users can use a CAN analyzer or two LaunchPADs to form a loop, as shown in Basic Structure of Accompanying Demo.

The example in this article supports both transparent transmission and protocol transmission. Figure 1-1 shows the PC terminal program for transparent transmission. Figure 1-2 shows the PC terminal program for protocol transmission.

For protocol transmission, this example specifies the UART message format. Users can also modify the format according to application requirements. When receiving the message from UART, 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 in the same format. 55 AA is the header. ID area is four bytes. The length area is one byte, which indicates the data length.

For transparent transmission, a configurable timeout is used for UART to detect one message. Data from UART is filled into the data area of CAN (same in reverse). CAN ID is the default value.

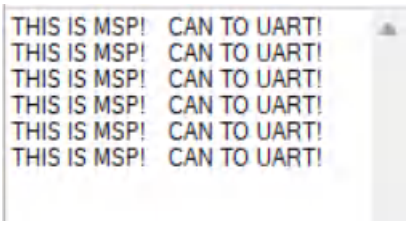

Figure 1-1. PC Terminal Program for Transparent Transmission

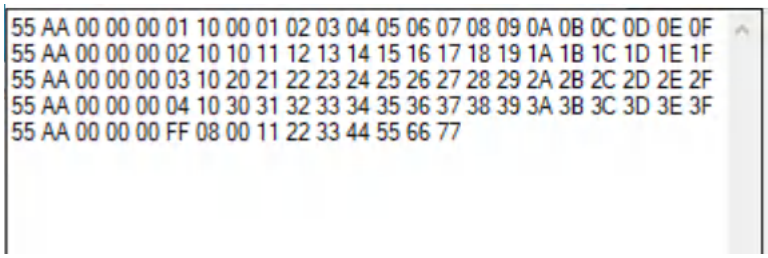

Figure 1-2. PC Terminal Program for Protocol Transmission

2 Implementation

2.1 Principle

In the design of this article, the CAN-UART bridge uses both CAN receive and transmit and UART receive and transmit. So both the CAN module and the UART module must be configured. Since the message formats of different communications are different, the CAN-UART bridge also must convert the message format.

For CAN, the CAN module supports both classic CAN and CAN FD (CAN with flexible data-rate) protocols. The CAN module is compliant to ISO 11898-1:2015. For more information, see to the related document. For UART, the interface can be used to transfer data between a MSPM0 device and another device with serial asynchronous communication protocols. For more information, see the related document.

Figure 2-1 shows the basic principle of the CAN-UART bridge. Typically, the communication rate of CAN is much higher than that of UART. For CAN FD the baud rate can be up to 5Mbps, while the UART operates at 9600bps as in the example code. As a result, it is possible that the data received by CAN is not sent by the UART in time. To match the rate, this scheme uses a buffer to transfer data between CAN and UART. This buffer not only implements data caching, but also implements data format conversion. This is equivalent to adding a barrier between the two communication interfaces. Users can add overload control actions for the overload case.

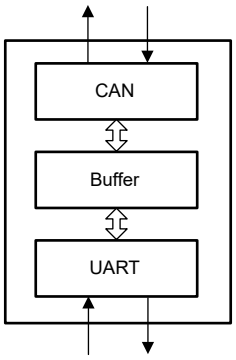

Figure 2-1. Basic Principle of CAN-UART Bridge

2.2 Structure

The structure of CAN-UART bridge with protocol transmission can be seen in Figure 2-3. 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 2-4 shows the structure of a CAN-UART bridge with transparent transmission. A timer interrupt is added to detect the timeout as the end of one packet.

Both UART and CAN reception are set to interrupt trigger so that messages can be received in time. When entering an interrupt, the message is first fetched through getXXXRxMsg().

For CAN, the CAN frame is a fixed format. MSPM0 supports classic CAN or CANFD. The frame for CANFD is shown in Figure 2-2. The example in this article can define 0, one, and four bytes of additional ID in the data area for protocol transmission.

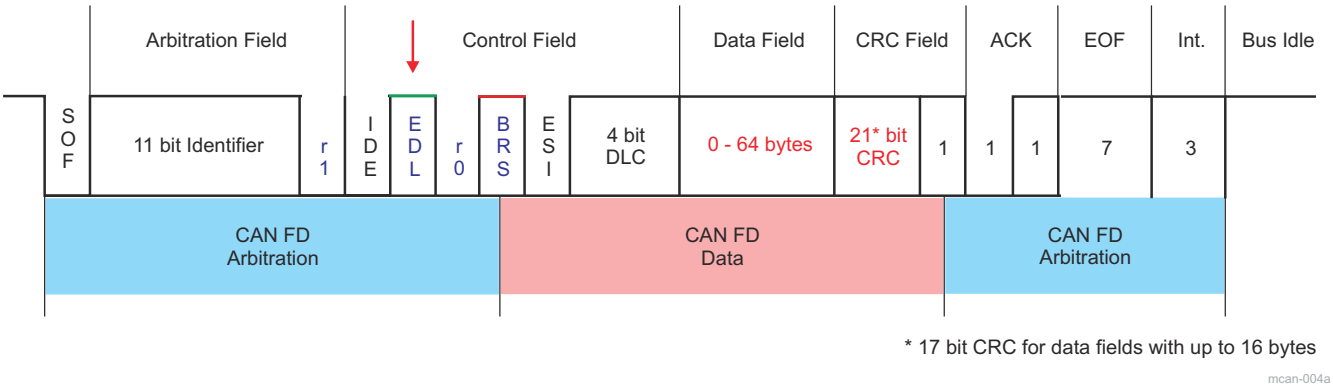

Figure 2-2. CAN FD Frame

Table 2-1. CAN Packet Form

	ID Area	Data
Protocol Transmission	4/1/0 bytes	(Data Length) Bytes

For UART protocol transmission, messages are identified based on serial frame information. The UART message format is listed in UART Packet Form.

Table 2-2. UART Packet Form

	Header	ID Area	Data Length	Data
Protocol Transmission	0×55 0×AA	4/1/0 bytes	1 byte	(Data Length) bytes
Transparent Transmission	—	—	—	(Data Length) bytes

The header is a fixed hex number combined *0x55 0xAA*, which means the start of the group. ID area occupies four bytes by default to match CAN ID, which can be configured as one byte or the ID area does not exist. The data length area occupies one byte. After Data Length area, a certain length of data is followed. This format is provided as an example. Users can modify the format according to application requirements.

For UART transparent transmission, messages are identified when timeout occurs. All bytes are regarded as pure data. The default value is the load for packet information. (For example, ID).

After receiving the message, processXXXRxMsg() converts the format of the message and stores the message in the FIFO as a new element. The format of the FIFO element is shown in Figure 2-5. In the format of the FIFO element, there are There are four categories in the FIFO element: *Origin_ID, Destination ID, Data Length* and *Data*. Users can also change the message items according to application requirements. In addition, this scheme also checks whether the FIFO is full for overload control. Users can add overload control actions according to application requirements.

Both CAN and UART transmissions are performed in the main function. When it is detected that the FIFO is not empty, the FIFO element is fetched. The message is formatted and sent. For CAN, the CAN frame is a fixed format. For UART, messages are sent in the format as described in CAN Packet Form. In the design of this article, UART TX interrupt is used to fill the data into the UART TX buffer.

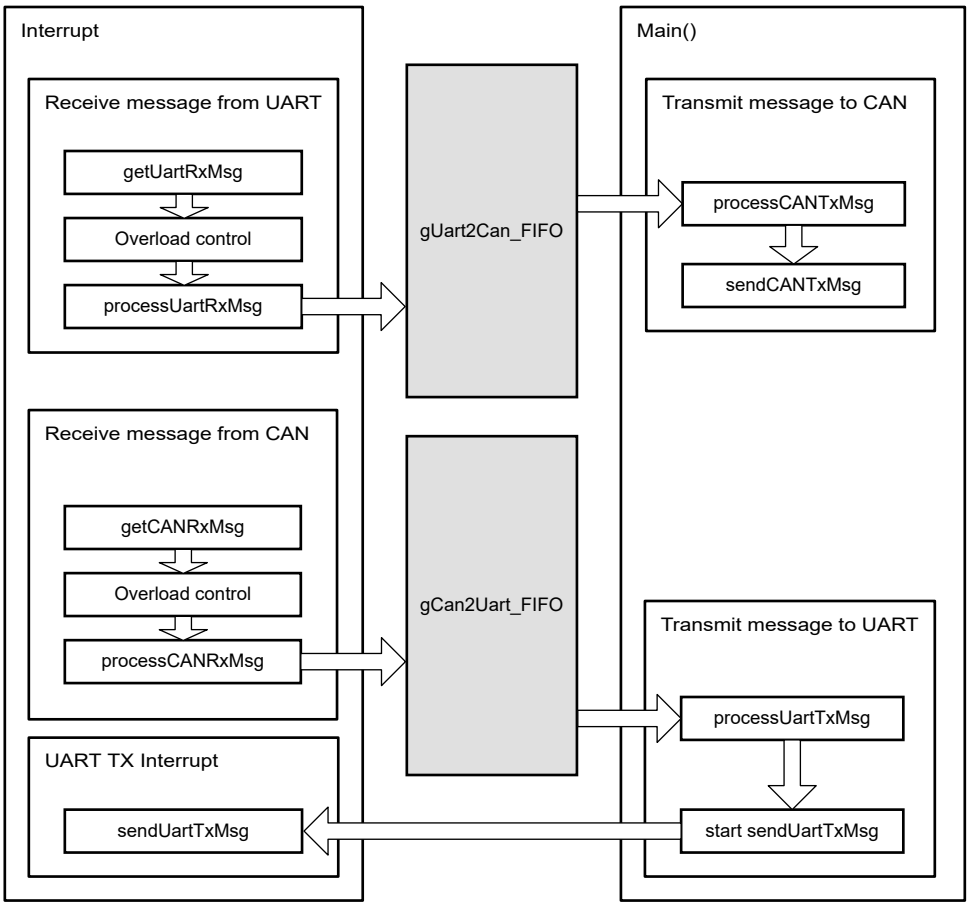

Figure 2-3. Structure of CAN-UART Bridge: Protocol

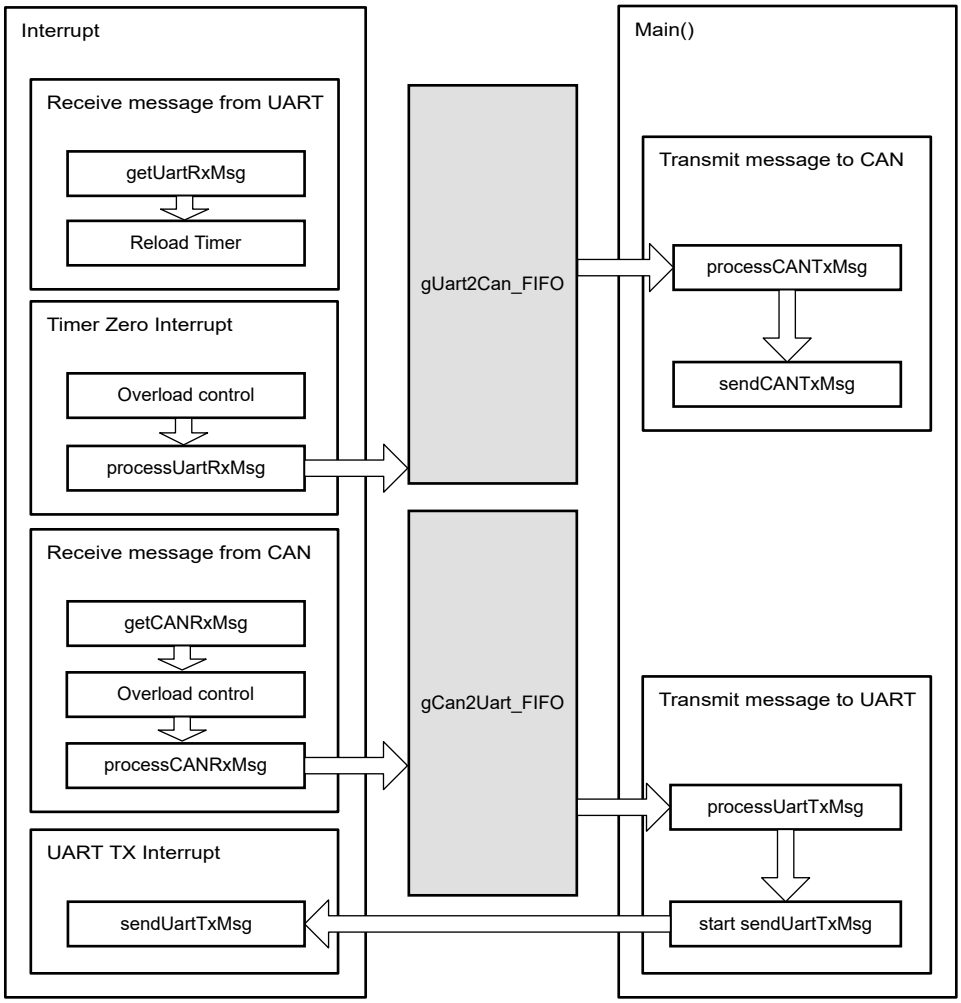

Figure 2-4. Structure of CAN-UART Bridge: Transparent

The structure of FIFO can be seen in Figure 2-5. Each FIFO uses three global variables to indicate the FIFO status. For *gUart2Can_FIFO, gUart2Can_FIFO.fifo_in* indicates the write position, *gUart2Can_FIFO.fifo_out* indicates the read position,and gUart2Can_FIFO.fifo_Count indicates the number of elements in the *gUart2Can_FIFO*.

If the *gUart2Can_FIFO* is empty, *gUart2Can_FIFO.fifo_in* equals the *gUart2Can_FIFO.fifo_out*, and the *gUart2Can_FIFO.fifo_count* is zero.

When performing *processUartRxMsg()*, a new message from UART is stored to *gUart2Can_FIFO*. So the *gUart2Can_FIFO.fifo_in* moves to the next position, and the *gUart2Can_FIFO.fifo_count* is incremented by one.

When transmitting a message from *gUart2Can_FIFO* to CAN, *gUart2Can_FIFO.fifo_out* moves to the next position, and the *gUart2Can_FIFO.fifo_count* subtracts one. *gCan2Uart_FIFO* is similar to *gUart2Can_FIFO*.

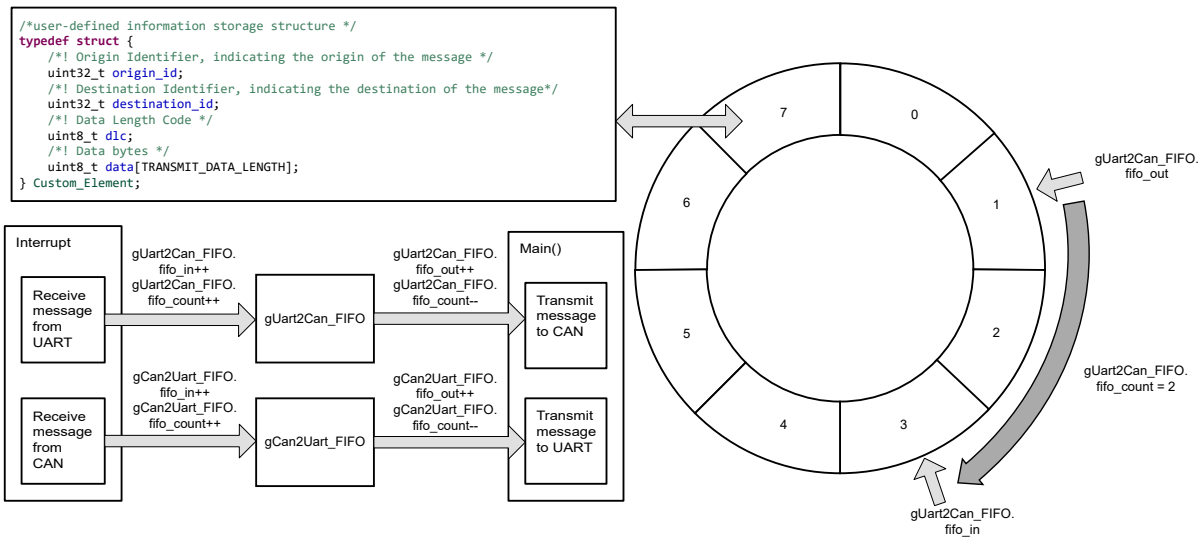

Figure 2-5. Structure of FIFO

3 Software Description

3.1 Software Functionality

Figure 2-3 shows the function design. The functions are listed in Table 3-1 .

Table 3-1. Functions and Descriptions

Tasks	Functions	Description	Location
UART receive	getUartRxMsg()	Receive 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()	Receive 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	

3.2 Configurable Parameters

All the configurable parameters are defined in user_define.h, which are listed in Configurable Parameters.

For UART, both transparent transmission and protocol transmission are supported in this example. These functions can switch by defining UART_TRANSPARENT or UART_PROTOCOL.

In transparent transmission, users can configure timeout for detecting one UART message receiving done.

In protocol transmission, users can configure the ID length for different formats. Please note there is a fixed 2-byte header (0×55 0×AA) and 1 byte data length. To modify the format more, users can require modifying the code directly.

```
#define UART_TRANSPARENT
#ifdef UART_TRANSPARENT
/* The format of Uart:
     * Transparent transmission - Data1 Data2 ...*/
#define UART_TIMEOUT  (0x4000)      //timeout 250ms
#else
#define UART_PROTOCOL
/* The format of Uart:
     * if UART_ID_LENGTH = 4, format is 55 AA ID1 ID2 ID3 ID4 Length Data1 Data2 ...
     * if UART_ID_LENGTH = 1, format is 55 AA ID Length Data1 Data2 ...
     * if UART_ID_LENGTH = 0, format is 55 AA Length Data1 Data2 ...*/
//#define UART_ID_LENGTH  (0)
//#define UART_ID_LENGTH  (1)
#define UART_ID_LENGTH  (4)
#endif
```

For CAN, ID and data length are included in CAN frame. Users can add another ID in the data area by changing the CAN_ID_LENGTH. (Default value is 0).

```
/* The format of CAN:
     * if CAN_ID_LENGTH = 4, format is ID1 ID2 ID3 ID4 Data1 Data2 ...
     * if CAN_ID_LENGTH = 1, format is ID Data1 Data2 ...
     * if CAN_ID_LENGTH = 0, format is Data1 Data2 ...*/
#define CAN_ID_LENGTH  (0)
//#define CAN_ID_LENGTH  (1)
//#define CAN_ID_LENGTH  (4)
```

Table 3-2. Configurable Parameters

Parameter	Optional Value	Description
#define UART_TRANSPARENT	Define / Not defined	Enable the UART transparent transmission.
#define UART_PROTOCOL	Define / Not defined	Enable the UART protocol transmission.
#define UART_TIMEOUT (0x4000)	Timeout = UART_TIMEOUT / 32768 s	Timeout to indicate one UART message receiving done.Only available when UART_TRANSPARENT is defined. In this case, default value is 250ms.
#define UART_ID_LENGTH (4)	0/1/4	Optional UART ID length, which is related to the ID area in protocol. Only available when UART_PROTOCOL is defined. In this case, the default value is four bytes.
#define CAN_ID_LENGTH (0)	0/1/4	Optional CAN ID length, which is related to the ID area in protocol. In this case, default value is 0 bytes.
#define TRANSMIT_DATA_LENGTH (12)	<=64	Size of data area. If the received message contains more data than this value, data loss can occur.
#define C2U_FIFO_SIZE (8)		Size of CAN to Uart FIFO. Note the usage of SRAM.
#define U2C_FIFO_SIZE (8)		Size of Uart to CAN FIFO. Note the usage of SRAM.
#define DEFAULT_UART_ORIGIN_ID (0x00)		Default value for UART origin ID
#define DEFAULT_UART_DESTINATION_ID (0x00)		Default value for UART destination ID
#define DEFAULT_CAN_ORIGIN_ID (0x00)		Default value for CAN origin ID
#define DEFAULT_CAN_DESTINATION_ID (0x00)		Default value for CAN destination ID

3.3 Structure of Custom Element

Custom_Element is the structure defined in *user_define.h*. *Custom_Element* is also shown in Figure 2-5.

Origin Identifier indicates the origin of the message. The following are the examples(CAN_ID_LENGTH =0,UART_ID_LENGTH =4).

- Example 1 - CAN interface receive and transmit
  - When CAN-UART bridge receives a CAN message, the ID from the CAN frame is the *Origin Identifier*, which indicates where the message came from.
  - When CAN-UART bridge transmits a CAN message, Origin Identifier will be ignored(CAN_ID_LENGTH is set to 0 default).
- Example 2 - UART interface receive and transmit (UART protocol transmission)
  - When CAN-UART bridge receives the UART message(UART protocol transmission), *DEFAULT_UART_ORIGIN_ID* is the *Origin Identifier* since the UART does not have an ID.
  - When CAN-UART bridge transmits the UART message(UART protocol transmission), *Origin Identifier* will be 4-byte ID in UART data(UART_ID_LENGTH is set to 4 default), indicating where the message came from.
- Example 3 - UART interface receive and transmit (UART transparent transmission)
  - When the CAN-UART bridge receives the UART message (UART transparent transmission), *DEFAULT_UART_ORIGIN_ID* is the *Origin Identifier* since the UART does not have an ID.
  - When the CAN-UART bridge transmits the UART message (UART transparent transmission), *Origin Identifier* will be ignored (Transparent transmission does not have an ID area).

Destination Identifier indicates the destination of the message. The following are the examples(CAN_ID_LENGTH =0,UART_ID_LENGTH =4).

- Example 1 - CAN interface receive and transmit
  - When the CAN-UART bridge receives a CAN message, the DEFAULT_CAN_DESTINATION_ID is the *Destination Identifier* since the CAN_ID_LENGTH is set to 0 by default. UART transmit does not require an ID.
  - When the CAN-UART bridge transmits a CAN message, *Destination Identifier* will be CAN ID in CAN frame. In this example, 11 bit or 29 bit are both supported.
- Example 2 - UART interface receive and transmit (UART protocol transmission)
  - When the CAN-UART bridge receives a UART message (UART protocol transmission), the 4-byte ID from UART data is the *Destination Identifier* (UART_ID_LENGTH is set to 4 default). The CAN transmit requires ID information.
  - When the CAN-UART bridge transmits a UART message (UART protocol transmission), *Destination Identifier* will be ignored since UART transmit does not require an ID.
- Example 3 - UART interface receive and transmit (UART transparent transmission)
  - When CAN-UART bridge receives UART message (UART transparent transmission), DEFAULT_UART_DESTINATION_ID is the *Destination Identifier*. (Transparent transmission does not have an ID area). CAN transmit requires ID information.
  - When the CAN-UART bridge transmits a UART message (UART transparent transmission), *Destination Identifier* will be ignored since UART transmit does not require an ID.

```
/*user-defined information storage structure */
typedef struct {
    /*! Origin Identifier, indicating the origin of the message */
    uint32_t origin_id;
    /*! Destination Identifier, indicating the destination of the message */
    uint32_t destination_id;
    /*! Data Length Code */
    uint8_t dlc;
    /*! Data bytes */
    uint8_t data[TRANSMIT_DATA_LENGTH];
} Custom_Element;
```

3.4 Structure of FIFO

Custom_FIFO is the structure defined in *user_define.h*. This is also shown in Figure 2-5.

```
typedef struct {
    uint16_t fifo_in;
    uint16_t fifo_out;
    uint16_t fifo_count;
    Custom_Element *fifo_pointer;
} Custom_FIFO;
```

gCan2Uart_FIFO and gUart2Can_FIFO are defined in main.c. Please note the usage of SRAM, which is related to C2U_FIFO_SIZE, U2C_FIFO_SIZE and the size for Custom_Element.

```
/* Variables for C2U_FIFO
 * C2U_FIFO is used to temporarily store message from CAN to UART */
Custom_Element gC2U_FIFO[C2U_FIFO_SIZE];
Custom_FIFO gCan2Uart_FIFO = {0, 0, 0, gC2U_FIFO};

/* Variables for U2C_FIFO
 * U2C_FIFO is used to temporarily store message from UART to CAN */
Custom_Element gU2C_FIFO[U2C_FIFO_SIZE];
Custom_FIFO gUart2Can_FIFO = {0, 0, 0, gU2C_FIFO};
```

3.5 UART Receive and Transmit (Transparent Transmission)

For *UART Receive*, there are three global variables defined in *bridge_uart.c*.

```
uint8_t gUartReceiveGroup[UART_RX_SIZE];
Custom_Element gUART_RX_Element;
uint16_t gGetUartRxMsg_Count;
```

The following is the process for *UART Receive*

1. Call *getUartRxMsg_transparent()* to store message into *gUartReceiveGroup*. Message receiving is done when timeout occurs or when the group is full (data up to *TRANSMIT_DATA_LENGTH* bytes)
2. Call *processUartRxMsg_transparent()* to extract data from *gUartReceiveGroup* and store the data into *gUART_RX_Element*.
3. Put *gUART_RX_Element* into *gUart2Can_FIFO*.

For UART transmit, there are two global variables defined in bridge_uart.c.

```
uint8_t gUartTransmitGroup[UART_TX_SIZE];
Custom_Element gUART_TX_Element;
```

The following is the process for *UART Transmit*.

1. Receive *gUART_TX_Element* from *gCan2Uart_FIFO*.
2. Call *processUartTxMsg_transparent()* to get data from *gUART_TX_Element* and store it into *gUartTransmitGroup*.
3. Call *sendUartTxMsg()* to transmit *gUartTransmitGroup* through UART.

3.6 UART Receive and Transmit (Protocol Transmission)

For *UART receive*, there are two global variables defined in *bridge_uart.c*.

```
uint8_t gUartReceiveGroup[UART_RX_SIZE];
Custom_Element gUART_RX_Element;
```

The following is the process for UART receive.

1. Call *getUartRxMsg()* to detect header to store the complete message into *gUartReceiveGroup*.
2. Call *processUartRxMsg()* to extract information from *gUartReceiveGroup* and store the information in *gUART_RX_Element*.
3. Put *gUART_RX_Element* into *gUart2Can_FIFO*.

For *UART transmit*, there are two global variables defined in *bridge_uart.c*.

```
uint8_t gUartTransmitGroup[UART_TX_SIZE];
Custom_Element gUART_TX_Element;
```

The following is the process for UART transmit.

1. Get *gUART_TX_Element* from *gCan2Uart_FIFO*.
2. Call *processUartTxMsg()* to get information from *gUART_TX_Element* and store the information into *gUartTransmitGroup*.
3. Call *sendUartTxMsg()* to transmit *gUartTransmitGroup* through UART.

3.7 CAN Receive and Transmit

For *CAN receive*, there are two global variables defined in *bridge_can.c*.

```
DL_MCAN_RxBufElement rxMsg;
Custom_Element gCAN_RX_Element;
```

The following is the process for *CAN receive*.

1. Call *getCANRxMsg()* to get complete message from *CAN message RAM* to *rxMsg*.
2. Call *processCANRxMsg()* to extract information from *rxMsg* and store it into *gCAN_RX_Element*.
3. Put *gCAN_RX_Element* into *gCan2Uart_FIFO*.

For *CAN transmit*, there are two global variables defined in *bridge_can.c*.

```
DL_MCAN_TxBufElement txMsg0;
Custom_Element gCAN_TX_Element;
```

The following is the process for CAN transmit.

1. Get *gCAN_TX_Element* from *gUart2Can_FIFO*.
2. Call *processCANTxMsg()* to get information from *gCAN_TX_Element* and store it into *txMsg0*.
3. Call *sendCANTxMsg()* to transmit *txMsg0* through CAN.

3.8 Application Integration

Functions in Table 3-1 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*. The structure of the FIFO element is defined in *user_define.h*.

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

For the basic configuration of peripherals, this project integrates the SysConfig configuration file. Users can easily modify the basic configuration of peripherals by using SysConfig.

Applications requiring this functionality must include the CAN module API and UART module API. All API files are included with the SDK download.

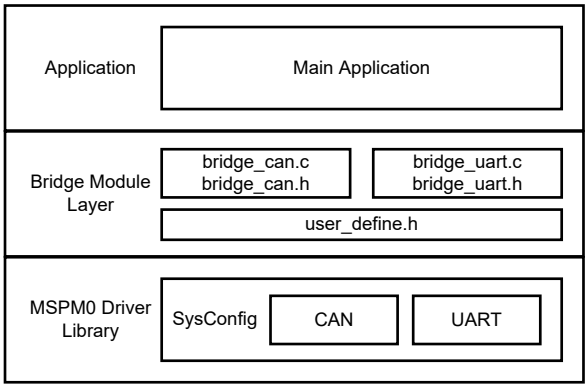

Figure 3-1. Files Required by the Software

Table 3-3 lists the footprint of the CAN-UART bridge design in terms of flash size and RAM size. Figure 3-1 and Table 3-3 were made using Code Composer Studio (Version: 12.7.1.00001) with optimization level 2.

The user can adjust the size of the FIFO. A larger FIFO means more cache capacity, but also takes up more RAM space. For details, see the relevant content in Section 5. In addition, the size of the data field in this code is set to a maximum of 64 bytes by default. The user can configure the data field size according to the actual data length. Using an 12 byte data field can significantly reduce RAM usage, as listed in Table 3-3

Table 3-3. Memory Footprint of the CAN-UART Bridge

Minimum Required Code Size (bytes)	Flash	SRAM
CAN-UART bridge (Protocol Transmission) U2C_FIFO_SIZE=8 C2U_FIFO_SIZE = 8 Data size = 12 bytes)	6328	910
CAN-UART bridge (Protocol Transmission) U2C_FIFO_SIZE=8 C2U_FIFO_SIZE=8 Data size = 64 bytes)	6416	2054
CAN-UART bridge (Protocol Transmission U2C_FIFO_SIZE=30 C2U_FIFO_SIZE=30 Data size = 12 bytes)	6432	1966

4 Hardware

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, the PC receives UART messages from the other LaunchPad™. Figure 4-1 shows the basic structure. Note that CAN transceivers are required to construct a CAN bus.

The accompanying demo uses two LaunchPads: a TCAN1046EVM and a PC. A TCAN1046EVM is a high-speed dual channel CAN transceiver evaluation module. Figure 4-2 shows the connection of the demo. For LaunchPad, a PA12 is used for CAN transmit and a PA13 is used for CAN receive. PA12 and PA13 should be connected to the TX pin and the RX pin of TCAN1046EVM. PA20 is used for UART transmit, and PA21 is used for UART receive. Note that back-channel UART interface on eZ-FET of the LaunchPad can be used on UART communication with PC.

For TCAN1046EVM, VCC must be connected to 5V and VIO must be connected to 3.3V since TCAN1046 supports level shifting. To build up a CAN bus, CANH1 and CANL1 must be connected to CANH2 and CANL2. Besides, the termination on the CAN bus (CANH and CANL) must be configured with the J2 (or J3) and J6 (or J8) jumpers. Each jumper adds 120Ω termination to the respective bus. For more information, see related documentation.

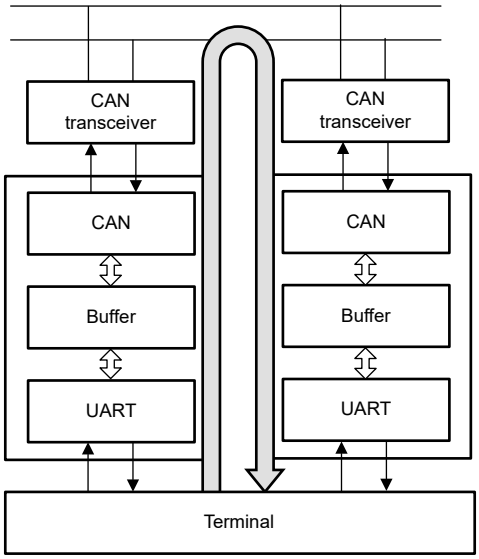

Figure 4-1. Basic Structure of Accompanying Demo

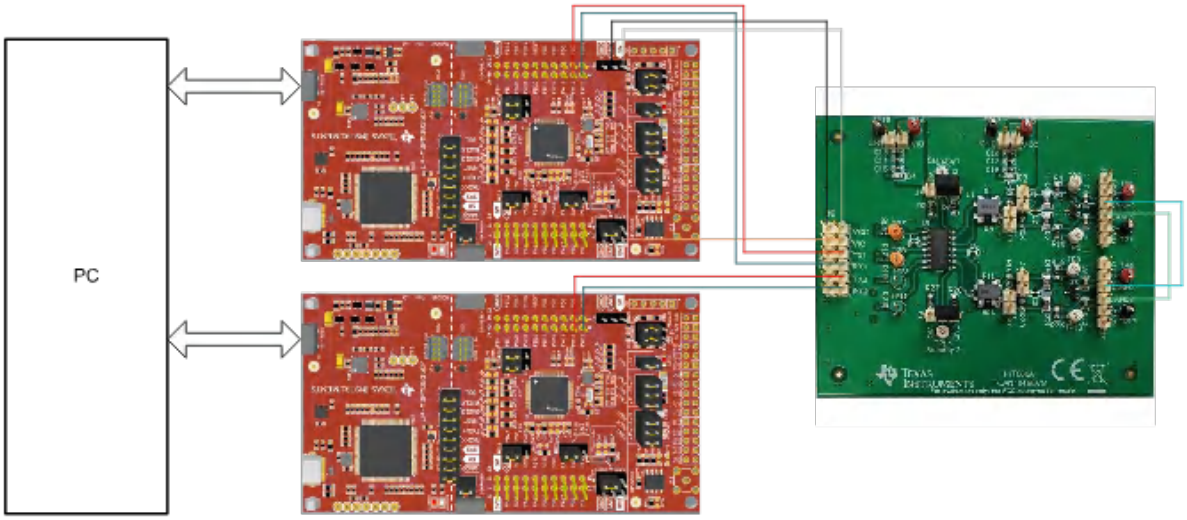

Figure 4-2. Hardware Connection of the Demo

5 Application Aspects

This section describes the application-level features of the CAN-UART bridge design and how to configure the design to meet application requirements.

5.1 Flexible structure

There are various configurable parameters, which are shown in Section 3.2. Users can configure the CAN and UART packet frame, the size of the FIFO, or the maximum size of data area by modifying these parameters which are all defined in *user_define.h*.

Users can modify the definition of *Custom_Element* in *user_define.h*. Entries can be increased or decreased based on application and storage requirements.

```
/*user-defined information storage structure */
typedef struct {
    /*! Origin Identifier, indicating the origin of the message */
    uint32_t origin_id;
    /*! Destination Identifier, indicating the destination of the message */
    uint32_t destination_id;
    /*! Data Length Code */
    uint8_t dlc;
    /*! Data bytes */
    uint8_t data[TRANSMIT_DATA_LENGTH];
} Custom_Element;
```

The reception and transmission of the two communication interfaces are separated. Messages are delivered through the FIFO. Users can make changes to the structure (for example, make messages follow a specific format or even a specific communication protocol). Additionally, users can split the structure into a one-way transmission according to Figure 2-3.

5.2 Optional Configuration for CAN

The CAN module of MSPM0 conforms with CAN Protocol 2.0 A, B and ISO 11898-1:2015. Users can configure various functions of the CAN module. By using SysConfig, users can change the basic configuration of CAN. (For example, the data transmission rate).

The code provide with an optional configuration for the CAN ID. The sample code defaults to 11 bit ID (standard ID). The configuration can be changed by modifying *user_define.h*.

- Add *#define CAN_ID_EXTEND* to enable 29-bit ID (Extended ID).

In addition, this sample code supports carrying 64 bytes of data in a single frame. Users can configure the appropriate data size according to requirements, which can further reduce the RAM space occupied by the FIFO.

5.3 CAN Bus Multi-Node Communication Example

CAN communication is a bus communication. Users can use the CAN-UART bridge design to test the multi-node communication of the CAN bus. Figure 5-1 shows the basic structure. When the user sends a message to the CAN bus through any CAN-UART bridge, the message is read back from other nodes immediately.

At least three LaunchPads must be used. Each CAN communication on the LaunchPad requires a transceiver. The connection between the LaunchPad and transceiver is shown in Figure 4-2.

The CAN module of MSPM0 supports hardware filtering to select messages with specific IDs. Note that hardware filtering is not performed by default in this sample code. The user can configure hardware filtering. For specific configuration, see related documentation.

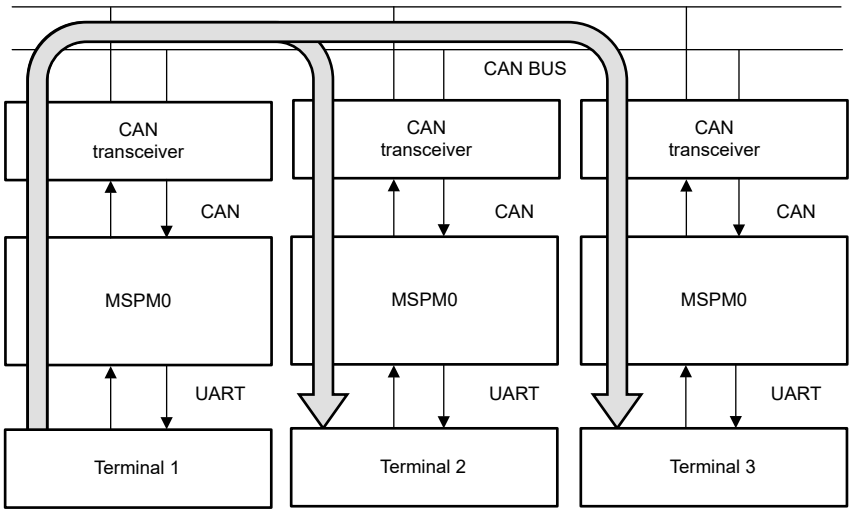

Figure 5-1. Basic Structure of Multi-Node Communication

6 Summary

This document introduce the implementation of CAN to UART bridge, including structure, function definition, interface usage and application aspects. With the example, MSPM0 can act like a translator between CAN and UART, allowing the user to send and receive information on one interface and receive and send the information on the other interface.

7 References

- Texas Instruments, *Bridge Solution between CAN and SPI with MSPM0 MCUs*, application note.
- Texas Instruments, *Bridge Solution between CAN and I2C with MSPM0 MCUs*, application note.
- Texas Instruments, *CAN to UART Bridge*, subsystem design.
- Texas Instruments, CAN to SPI Bridge, subsystem design.
- Texas Instruments, CAN to I2C Bridge, subsystem design.
- Texas Instruments, Download the MSPM0 SDK, resource explorer.
- Texas Instruments, Learn more about SysConfig, system configuration tool.

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