Application Note

Bridge Design between CAN and I2C with MSPM0 MCUs

TEXAS INSTRUMENTS

ABSTRACT

This application note introduces the example for CAN to I2C bridge. The document describes the structure and behavior of CAN to I2C bridge. Then the document introduces the software implementation, hardware connection and application usage. Users can configure the bridge by modifying the predefinition. In addition, the relevant code is provided to the users.

Table of Contents

List of Figures

List of Tables

Trademarks

LaunchPad™ is a trademark of Texas Instruments.
All trademarks are the property of their respective owners.

1 Introduction

Based on different applications, now there are many communication methods between devices. MCUs today usually support more than one communication method. For example, MSPM0 can support UART, I2C, CAN, and so on on a specific device. When devices must transfer data over different communication interfaces, a bridge is constructed.

For CAN and I2C, a CAN-I2C bridge acts as a translator between the two interfaces. A CAN-I2C 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-I2C bridge. The MSPM0G3507 microcontroller (MCU) can be used as a design by providing CAN and I2C communication interfaces. The accompanying demo uses the MSPM0G3507 with 2Mbps CANFD and 400kHz bus speed I2C to demonstrate transceiving data between channels.

1.1 Bridge Between CAN and I2C

The CAN-I2C bridge connects the CAN and I2C interfaces. The bridge supports I2C to work in slave mode or master mode. The example in this document uses a CAN analyzer to observe the CAN data. A user can also send messages from CAN analyzer over the CAN-I2C bridge to the I2C side. For I2C data, users can use a logic analyzer, or use two LaunchPad™s to form a loop to observe, such as in the accompanying demo in Figure 4-1.

The example in this document support both transparent transmission and protocol transmission. Figure 1-1 shows the logic analyzer observation for transparent transmission. Figure 1-2 shows the logic analyzer observation for protocol transmission.

For protocol transmission, this example specifies the I2C message format. Users can also modify the format according to application requirements. When receiving the message from I2C, the message format is < 55 AA ID1 ID2 ID3 ID4 Length Data1 Data2 ...>. Users can send data through the I2C as the same format. 55 AA is the header. ID area is four bytes. Length area is one byte, which indicates the data length.

For transparent transmission, I2C stop interrupt is used to detect one message. Data from I2C is filled into the data area of CAN (same in reverse). CAN ID is the default value.

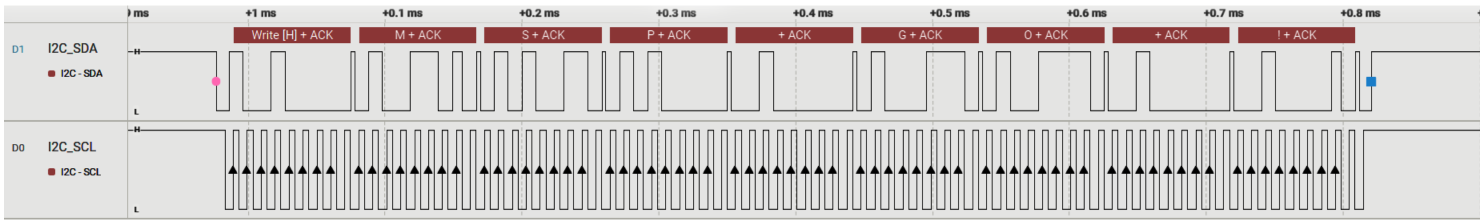


Figure 1-1. Logic Analyzer for I2C Transparent Transmission

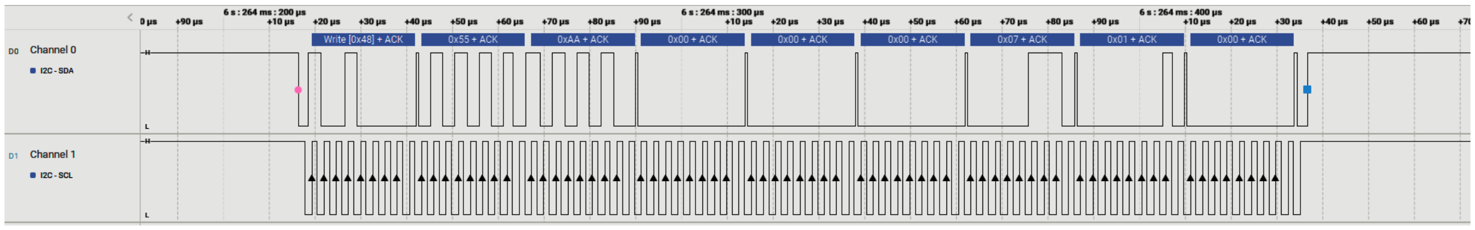


Figure 1-2. Logic Analyzer for I2C Protocol Transmission

2 Implementation

2.1 Principle

In the design of this document, the CAN-I2C bridge uses both CAN receive and transmit and I2C receive and transmit. So both the CAN module and the I2C module must be configured. Since the message formats of different communications are different, the CAN-I2C 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 related documentation. For I2C, the interface can be used as slave or master to transfer data between a MSPM0 device and another I2C device. For more information, see related documentation. Since the receiving and transmitting of the I2C slave are controlled by the I2C master, the I2C slave cannot initiate transmission to the I2C master. To achieve communication from the slave to the master, a line is added to this design. The IO pull-down of the slave notifies the master that there is information to be sent.

Figure 2-1 shows the basic principle of CAN-I2C bridge. Typically, the communication rate of CAN is different from that of I2C. For CAN FD the baud rate can be up to 5Mbps, while the I2C operates at 400kHz bus speed as in the example code. As a result, it is possible that the data received by one interface is not be sent by another interface in time. To match the rate, this scheme uses a buffer to transfer data between CAN and I2C. 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.

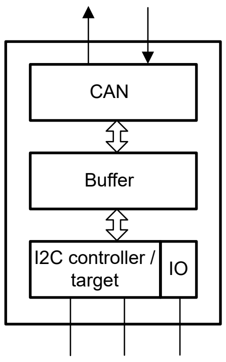


Figure 2-1. Basic Principle of CAN-I2C Bridge

2.2 Structure

The structure of CAN-I2C bridge with protocol and transparent transmission is shown in Figure 2-3 and Figure 2-4. Figure 2-3 is for the I2C master and Figure 2-4 is for the I2C slave. The CAN- I2C bridge can be divided into four independent tasks: receive from I2C, receive from CAN, transmit through CAN, transmit through I2C. Two FIFOs implement bidirectional message transfer and message caching.

Both I2C and CAN reception are set to interrupt triggers so that messages are received in time. When entering an interrupt, the message is first obtained through *getXXXRxMsg()*.

For CAN, the CAN frame is a fixed format. MSPM0 supports classic CAN or CANFD. The frame for CANFD can be seen in Figure 2-2. The example in this article can define 0/1/4 additional bytes (default length is one byte for I2C address) in data area for protocol transmission, which is listed in Table 2-1.

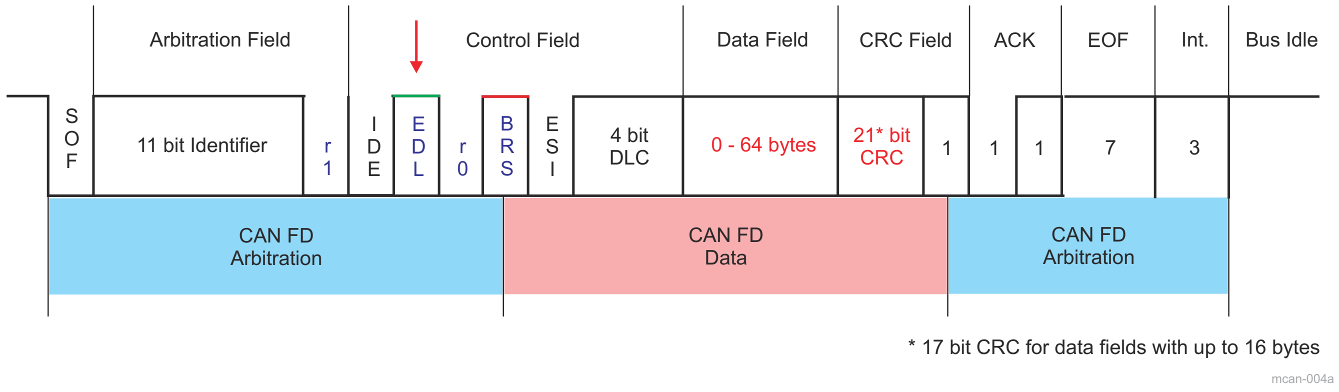


Figure 2-2. CAN FD Frame

Table 2-1. CAN packet form

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

For I2C protocol transmission, messages are identified based on serial frame information. The I2C message format is listed in Table 2-2.

Table 2-2. I2C Packet Form

	Header	ID Area	Data length	Data
Protocol Transmission	0x55 0xAA	4/1/0 bytes	1 byte	(Data Length) bytes
Transparent Transmission	—	—	—	Master to slave - (Data Length) bytes Slave to master - (I2C_TRANSPARENT_LENGTH) bytes

The header is a fixed hex number combine *0x55 0xAA*, which means the start of the group. ID area occupies four bytes default to match CAN ID, which can be configured to be one byte or does not exist. The data length area occupies one byte. After the 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.

Note that I2C is a communication method where the I2C master controls the transmission and reception. In general, an I2C slave cannot initiate communication. For I2C slave-to-master communication, I2C slave pulls down the IO when messages must be sent, as shown in Figure 2-4. The I2C master initiates the I2C read command in the IO interrupt when the IO is detected low, as shown in Figure 2-3.

For I2C transparent transmission, messages are identified by I2C stop interrupt, as shown in Figure 2-4. All bytes are regards as pure data. The default value is loaded 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. Figure 2-5 shows the format of a FIFO element. In the format of the FIFO element, there are *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 as requirements change.

Both CAN and I2C transmission 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, CAN frame is a fixed format as described in Table 2-1. For the I2C, messages are sent in the format listed in Table 2-2.

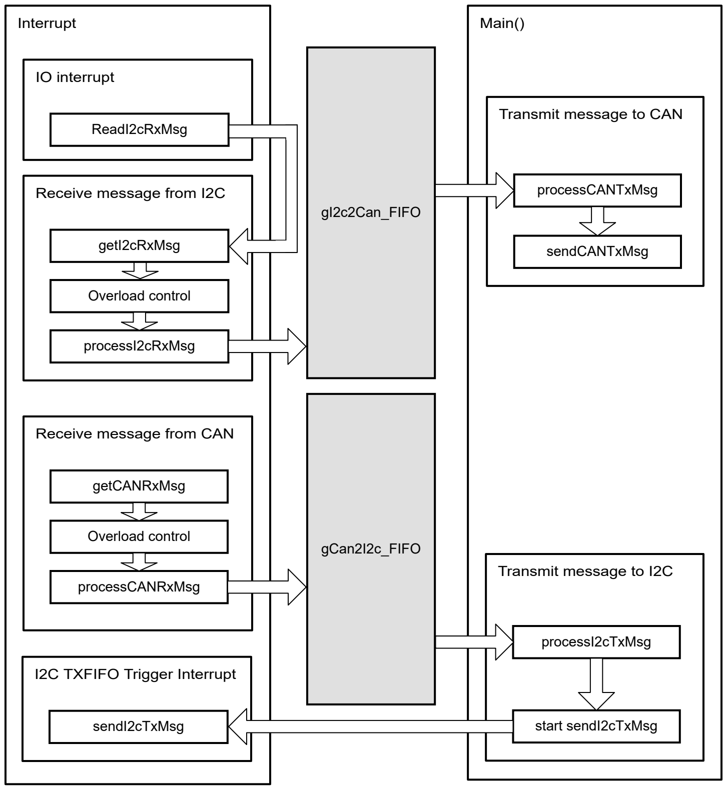


Figure 2-3. Structure of CAN-I2C (I2C controller) Bridge: Protocol and Transparent

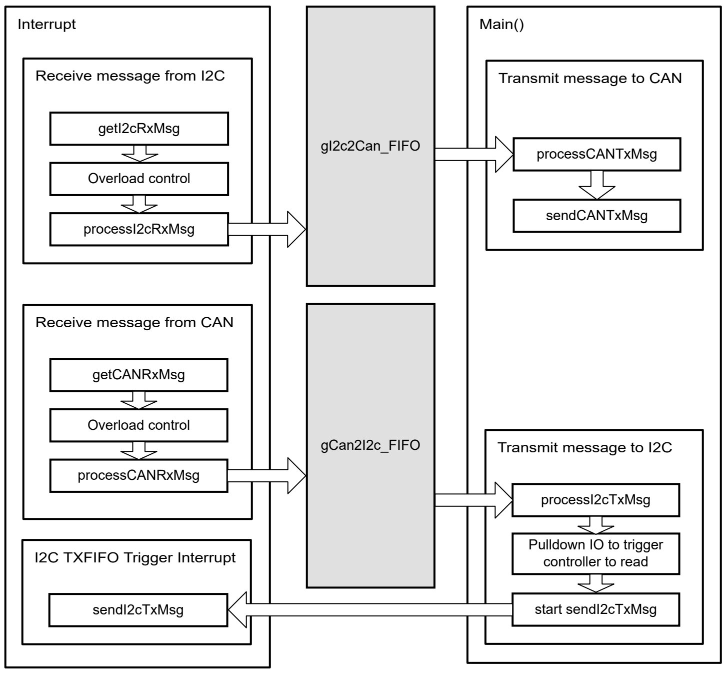


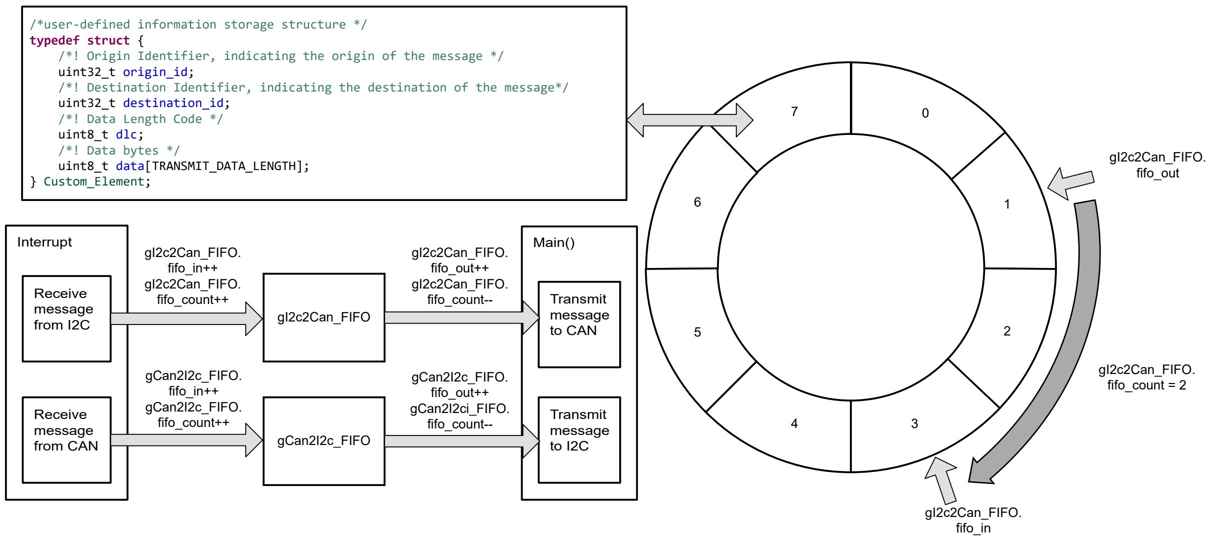
Figure 2-4. Structure of CAN-I2C (I2C Target) Bridge: Protocol and Transparent

Figure 2-5 shows the structure of the FIFO. Each FIFO uses three global variables to indicate the FIFO status. For *gI2c2Can_FIFO*, *gI2c2Can_FIFO.fifo_in* indicates the write position, *gI2c2Can_FIFO.fifo_out* indicates the read position,and *gI2c2Can_FIFO.fifo_Count* indicates the number of elements in the *gI2c2Can_FIFO*.

If the *gI2c2Can_FIFO* is empty, *gI2c2Can_FIFO.fifo_in* equals *gI2c2Can_FIFO.fifo_out,* and *gI2c2Can_FIFO.fifo_count* is zero.

When performing *processI2cRxMsg()*, a new message from I2C is stored to *gI2c2Can_FIFO*. So the *gI2c2Can_FIFO.fifo_in* moves to the next position, and *gI2c2Can_FIFO.fifo_count* is incremented by one.

When transmitting a message from *gI2c2Can_FIFO* to CAN, *gI2c2Can_FIFO.fifo_out* moves to next position, and *gI2c2Can_FIFO.fifo_count* minus 1. *gCan2I2c_FIFO* is similar to *gI2c2Can_FIFO*.



```
/*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;
```

Interrupt
Receive message from I2C
Receive message from CAN
gI2c2Can_FIFO. fifo_in++ gI2c2Can_FIFO. fifo_count++
gCan2I2c_FIFO. fifo_in++ gCan2I2c_FIFO. fifo_count++
gI2c2Can_FIFO
gCan2I2c_FIFO
gI2c2Can_FIFO. fifo_out++ gI2c2Can_FIFO. fifo_count--
gCan2I2c_FIFO. fifo_out++ gCan2I2ci_FIFO. fifo_count--
Main()
Transmit message to CAN
Transmit message to I2C
0
1
2
3
4
5
6
7
gI2c2Can_FIFO. fifo_out
gI2c2Can_FIFO. fifo_count = 2
gI2c2Can_FIFO. fifo_in

Figure 2-5. Structure of FIFO

3 Software Description

3.1 Software Functionality

The functions are designed according to Figure 2-3 and Figure 2-4. The functions are listed in Table 3-1.

Table 3-1. Functions and Descriptions

Tasks	Functions	Description	Location
I2C receive	readI2CRxMsg_controller()	Send a read request to slave (I2C master only)	bridge_i2c.c bridge_i2c.h
	getI2CRxMsg_controller()	Obtain the received I2C message (I2C master only) (protocol)	
	getI2CRxMsg_controller_transparent()	Obtain the received I2C message (I2C master only) (transparent)	
	getI2CRxMsg_target()	Obtain the received I2C message (I2C slave only) (protocol)	
	getI2CRxMsg_target_transparent()	Obtain the received I2C message (I2C slave only) (transparent)	
	processI2CRxMsg_controller()	Convert the received I2C message format (protocol) and store it into gI2C_RX_Element (I2C master only)	
	processI2CRxMsg_controller_ transparent()	Convert the received I2C message format (transparent) and store it into gI2C_RX_Element (I2C master only)	
	processI2CRxMsg_target()	Convert the received I2C message format (protocol) and store it into gI2C_RX_Element (I2C slave only)	
	processI2CRxMsg_target_transparent()	Convert the received I2C message format (transparent) and store it into gI2C_RX_Element (I2C slave only)	
I2C transmit	processI2CTxMsg_controller()	Convert the gI2C_TX_Element format (protocol) to be sent through I2C (I2C master only)	
	processI2CTxMsg_controller_ transparent()	Convert the gI2C_TX_Element format (transparent) to be sent through I2C (I2C master only)	
	processI2CTxMsg_target()	Convert the gI2C_TX_Element format (protocol) to be sent through I2C (I2C slave only)	
	processI2CTxMsg_target_transparent()	Convert the gI2C_TX_Element format (transparent) to be sent through I2C (I2C slave only)	
	sendI2CTxMsg_controller()	Send message through I2C (I2C master only)	
	sendI2CTxMsg_target()	Send message through I2C (I2C slave only)	
CAN receive	getCANRxMsg()	Obtain 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 Table 3-3.

For I2C, both transparent transmission and protocol transmission are supported in this example, switching by defining I2C_TRANSPARENT or I2C_PROTOCOL.

In transparent transmission, users can configure the default data length(I2C_TRANSPARENT_LENGTH) for message from I2C slave to I2C master. Table 3-2 lists the number of bytes for receiving or sending with different modes.

In protocol transmission, users can configure the ID length for different formats. Please note there is a fixed 2-byte header (0x55 0xAA) and 1-byte data length. To modify the format more, users can modify the code directly.

```
#define I2C_TRANSPARENT
#ifdef I2C_TRANSPARENT
/* The format of I2C:
     * Transparent transmission - Data1 Data2 ...*/
/* data length for I2C master receiving or I2C slave transmitting*/
#define I2C_TRANSPARENT_LENGTH  (8)
#else
/* The format of I2C:
     * if I2C_ID_LENGTH = 4, format is 55 AA ID1 ID2 ID3 ID4 Length Data1 Data2 ...
     * if I2C_ID_LENGTH = 1, format is 55 AA ID Length Data1 Data2 ...
     * if I2C_ID_LENGTH = 0, format is 55 AA Length Data1 Data2 ...*/
//#define I2C_ID_LENGTH  (0)
//#define I2C_ID_LENGTH  (1)
#define I2C_ID_LENGTH  (4)
#endif

/* default address for I2C master receiving */
#define I2C_TARGET_ADDRESS (0x48)
```

Table 3-2. Number of Bytes for Receiving or Sending with Different I2C Modes

Parameter	I2C Interface: Master		I2C Interface: Slave	
	How many bytes are received?	**How many bytes are sent?**	**How many bytes are received?**	**How many bytes are sent?**
Protocol Transmission	(2+I2C_ID_LENGT+1+Length) bytes	(2+I2C_ID_LENGT+1+Length) bytes	(2+I2C_ID_LENGT+1+Length) bytes	(2+I2C_ID_LENGT+1+Length) bytes
Transparent Transmission	(I2C_TRANSPARENT_LENGTH) bytes	(Length) bytes	I2C stop interrupt identify the end of message	(I2C_TRANSPARENT_LENGTH) bytes

For CAN, an ID or data length are included in CAN frame. Users can add another ID in data area by changing CAN_ID_LENGTH(default value is 1). In this example, a one-byte ID is added for the I2C address.

```
/* 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 3-3. Configurable Parameters

Parameter	Optional Value	Description
#define I2C_TRANSPARENT	Define / Not define	Enables the I2C transparent transmission.
#define I2C_PROTOCOL	Define / Not define	Enables the I2C protocol transmission.
#defineI2C_TRANSPARENT_LENGTH (8)		Default data length for message from I2C slave to I2C master. Only available when I2C_TRANSPARENT is defined. In this case, default value is eight bytes.
#define I2C_TARGET_ADDRESS (0x48)		Default I2C slave address for message from I2C slave to I2C master. In this case, default value is 0x48
#define I2C_ID_LENGTH (4)	0/1/4	Optional I2C ID length, which is related to the ID area in protocol. Only available when I2C_PROTOCOL is defined. In this case, 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 one byte
#define TRANSMIT_DATA_LENGTH (12)	<=64	Size of data area. If the received message contains more data than this value, this can result in data loss
#define C2I_FIFO_SIZE (8)		Size of CAN to I2C FIFO. Note the usage of SRAM.
#define ItoC_FIFO_SIZE (8)		Size of I2C to CAN FIFO. Note the usage of SRAM.
#define DEFAULT_I2C_ORIGIN_ID (0x00)		Default value for I2C origin ID
#define DEFAULT_I2C_DESTINATION_ID (0x00)		Default value for I2C destination ID
#define DEFAULT_CAN_ORIGIN_ID (0x00)		Default value for CAN origin ID
#define DEFAULT_CAN_DESTINATION_ID (0x48)		Default value for CAN destination ID

3.3 Structure of Custom Element

Custom_Element is the structure defined in *user_define.h* Figure 2-5 shows this structure.

Origin Identifier indicates the origin of the message. The following are the examples (CAN_ID_LENGTH =1, I2C_ID_LENGTH =4).

- Example 1 - CAN interface receive and transmit
    1. When the CAN-I2C bridge receives a CAN message, the ID from CAN frame is the *Origin Identifier*, which indicates where the message comes from.
    2. When the CAN-I2C bridge transmits a CAN message, *Origin Identifier* is 1-byte ID in CAN(CAN_ID_LENGTH is set to 1 default), which indicates where the message comes from.

- Example 2 - I2C interface receive and transmit(I2C protocol transmission)
    1. When the CAN-I2C bridge receives an I2C message (I2C protocol transmission) and if I2C works as the master, I2C_TARGET_ADDRESS is the *Origin Identifier*, which indicates where the message comes from. If I2C works as a slave and the I2C master does not have an address, the DEFAULT_I2C_ORIGIN_ID is the *Origin Identifier*.
    2. When the CAN-I2C bridge transmits an I2C message (I2C protocol transmission), the *Origin Identifier* is a 4-byte ID in I2C data (I2C _ID_LENGTH is set to 4 default), which indicates where the message comes from.

- Example 3 - I2C interface receive and transmit(I2C transparent transmission)
    1. When the CAN-I2C bridge receives an I2C message (I2C transparent transmission), and if I2C works as a master, the I2C_TARGET_ADDRESS is the *Origin Identifier*, which indicates where the message comes from. If I2C works as a slave, and if the I2C master does not have an address, DEFAULT_I2C_ORIGIN_ID is the *Origin Identifier*.
    2. When the CAN-I2C bridge transmits an I2C message (I2C transparent transmission), *Origin Identifier* is 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 =1, I2C_ID_LENGTH =4).

- Example 1 - CAN interface receive and transmit
    1. When the CAN-I2C bridge receives a CAN message, a 1 byte ID from the CAN data area (CAN_ID_LENGTH is set to 1 default) is the *Destination Identifier*, which indicates the destination of the message(I2C address).
    2. When the CAN-I2C bridge transmits a CAN message, the *Destination Identifier* is the CAN ID in the CAN frame. In this example, 11 bits or 29 bits are both supported.

- Example 2 - I2C interface receive and transmit(I2C protocol transmission)
    1. When the CAN-I2C bridge receives an I2C message (I2C protocol transmission), a 4-byte ID from I2C data is the *Destination Identifier* (I2C_ID_LENGTH is set to 4 default). The CAN transmit requires ID information.
    2. When the CAN-I2C bridge transmits an I2C message (I2C protocol transmission), and if I2C works as a master, the *Destination Identifier* is the I2C address. If the I2C works as a slave, *Destination Identifier* is ignored. (Use the IO to trigger the master for a message.)

- Example 3 - I2C interface receive and transmit(I2C transparent transmission)
    1. When the CAN-I2C bridge receives an I2C message (I2C transparent transmission), the DEFAULT_I2C_DESTINATION_ID is the *Destination Identifier*. (Transparent transmission does not have ID area). The CAN transmit requires ID information.
    2. When the CAN-I2C bridge transmits an I2C message (I2C transparent transmission), and if I2C works as a master, *Destination Identifier* is the I2C address. If I2C works as a slave, the *Destination Identifier* is ignored (Use the IO to trigger the master for a message).

```
/*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*. Custom_FIFO 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;
```

gCan2I2c_FIFO and *gI2c2Can_FIFO* are defined in *main.c*. Note the usage of SRAM, which is related to *C2I_FIFO_SIZE*, *ItoC_FIFO_SIZE* and the size for *Custom_Element*.

```
/* Variables for ItoC_FIFO
 * ItoC_FIFO is used to temporarily store message from I2C to CAN */
Custom_Element gItoC_FIFO[ItoC_FIFO_SIZE];
Custom_FIFO gI2c2Can_FIFO = {0, 0, 0, gItoC_FIFO};

/* Variables for C2I_FIFO
 * C2I_FIFO is used to temporarily store message from CAN to I2C */
Custom_Element gC2I_FIFO[C2I_FIFO_SIZE];
Custom_FIFO gCan2I2c_FIFO = {0, 0, 0, gC2I_FIFO};
```

3.5 I2C Receive and Transmit (Transparent Transmission)

In general, the I2C master controls the I2C communication, and the I2C slave cannot trigger slave-to-master communication. In this design, another IO is used. The IO pull-down of the slave notifies the master that there is information to be sent. The user can modify the pin or remove the IO function as required.

For I2C receive, there are three global variables defined in *bridge_i2c.c*.

```
uint8_t gI2CReceiveGroup[I2C_RX_SIZE];
Custom_Element gI2C_RX_Element;
uint16_t gGetI2cRxMsg_Count;
```

The following is the process for the I2C master receive. An IO interrupt is used to detect the IO pull-down.

1. In IO interrupt, call *readI2CRxMsg_controller()* to send a read request to the I2C slave for I2C_TRANSPARENT_LENGTH) bytes.
2. Call *getI2CRxMsg_controller_transparent()* to store a message into *gI2cReceiveGroup*. Message reception is finished when (I2C_TRANSPARENT_LENGTH) bytes are received.
3. Call *processI2CRxMsg_controller_transparent()* to extract data from *gI2cReceiveGroup* and the store message into *gI2C_RX_Element*.
4. Place *gI2C_RX_Element* into *gI2c2Can_FIFO*.

The following is the process for I2C slave receive.

1. Call getI2CRxMsg_target_transparent() to store message into *gI2cReceiveGroup*. Message receiving is finished when I2C stop interrupt occurs(I2C STOP condition).
2. Call processI2CRxMsg_target_transparent() to extract data from *gI2cReceiveGroup* and store the data in *gI2C_RX_Element*.
3. Place *gI2C_RX_Element* into *gI2c2Can_FIFO*.

For the I2C transmit, there are four global variables defined in *bridge_i2c.c*.

```
uint8_t gI2cTransmitGroup[I2C_TX_SIZE];
Custom_Element gI2C_TX_Element;
uint32_t gTxLen, gTxCount;
```

The following is the process for I2C master/slave transmit.

1. Obtain *gI2C_TX_Element* from *gCan2I2c_FIFO*.
2. Call *processI2CTxMsg_controller_transparent()* and *processI2CTxMsg_target_ transparent()* to receive data from *gI2C_TX_Element* and store the message into *gI2cTransmitGroup*.
3. Call *sendI2CTxMsg_controller()* and *sendI2CTxMsg_target()* to transmit *gI2cTransmitGroup* through the I2C. For the I2C slave, an IO is used to trigger the master to read from the slave, and only (I2C_TRANSPARENT_LENGTH) bytes are sent.

The functions for master mode or slave mode are both included in *bridge_i2c.c*.

3.6 I2C Receive and Transmit (Protocol Transmission)

In general, the I2C master controls the I2C communication, and the I2C slave cannot trigger slave-to-master communication. In this design, another IO is used. The slave's IO pull-down notifies the master that there is information to be sent. Users can modify the pin or remove the IO function as required.

For I2C receive, there are two global variables defined in *bridge_i2c.c*.

```
uint8_t gI2CReceiveGroup[I2C_RX_SIZE];
Custom_Element gI2C_RX_Element;
```

The following is the process for I2C master receive. IO interrupt is used to detect the IO pull-down.

1. In IO interrupt, call *readI2CRxMsg_controller()* to send a read request to the I2C slave.
2. Call *getI2CRxMsg_controller()* to detect a header to store the complete message in *gI2cReceiveGroup*.
3. Call processI2CRxMsg_controller() to extract data from *gI2cReceiveGroup* and store the data in *gI2C_RX_Element.*
4. Place *gI2C_RX_Element* into *gI2c2Can_FIFO*.

The following is the process for the I2C slave reception.

1. Call *getI2CRxMsg_target()* to store the message into *gI2cReceiveGroup*.
2. Call processI2CRxMsg_target() to extract data from *gI2cReceiveGroup* and store the data into *gI2C_RX_Element*.
3. Place *gI2C_RX_Element* into *gI2c2Can_FIFO*.

For I2C transmit, there are four global variables defined in *bridge_i2c.c*.

```
uint8_t gI2cTransmitGroup[I2C_TX_SIZE];
Custom_Element gI2C_TX_Element;
uint32_t gTxLen, gTxCount;
```

The following is the process for I2C master and slave transmission.

1. Obtain *gI2C_TX_Element* from *gCan2I2c_FIFO*.
2. Call processI2CTxMsg_controller() / processI2CTxMsg_target() to obtain data from *gI2C_TX_Element* and store the message into *gI2cTransmitGroup*.
3. Call *sendI2CTxMsg_controller()* and *sendI2CTxMsg_target()* to transmit *gI2cTransmitGroup* through the I2C. For the I2C slave, an IO is used to trigger the master to read from the slave.

The functions for master mode or slave mode are both included in *bridge_i2c.c*.

3.7 CAN Receive and Transmit

For CAN receive, there are 2 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 obtain the complete message from *CAN message RAM* to *rxMsg*.
2. Call *processCANRxMsg()* to extract information from *rxMsg* and store it into *gCAN_RX_Element*.
3. Place *gCAN_RX_Element* into *gCan2I2c_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 *gI2c2Can_FIFO*.
2. Call *processCANTxMsg()* to receive 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 I2C reception and transmission are included in *bridge_i2c.c* and *bridge_i2c.h*. Functions for CAN reception and transmission are included in *bridge_can.c* and *bridge_can.h*. Structure of FIFO element is defined in *user_define.h*.

Users can separate functions by file. For example, if only I2C functions are required, users can reserve *bridge_i2c.c* and *bridge_i2c.h* to call the functions.

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

Applications requiring this functionality must include the CAN module API and the I2C module API. All API files are included with the SDK.

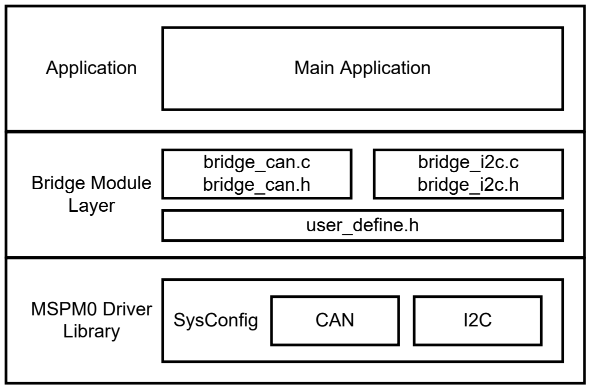


Figure 3-1. Files Required by the Software

Table 3-4 details the footprint of the CAN-I2C bridge solution in terms of Flash size and RAM size. The table and figure below have been determined using the 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, please see the relevant content in Application Aspects. The user can configure the data field size according to the actual data length. As listed in Table 3-4, using a less-byte data field can significantly reduce RAM usage.

Table 3-4. Memory Footprint of the CAN-I2C Bridge

Minimum required code size (bytes)	Flash	SRAM
CAN-I2C master bridge (protocol transmission ItoC_FIFO_SIZE=8 C2S_FIFO_SIZE=8 Data size = 12 bytes)	6352	1428
CAN-I2C slave bridge (protocol transmission ItoC_FIFO_SIZE=8 C2I_FIFO_SIZE=8 Data size = 12 bytes)	6264	1428
CAN-I2C master bridge (protocol transmission ItoC_FIFO_SIZE=8 C2I_FIFO_SIZE=8 Data size = 64 bytes)	6440	2572
CAN-I2C slave bridge (protocol transmission ItoC_FIFO_SIZE=8 C2I_FIFO_SIZE=8 Data size = 64 bytes)	6360	2572
CAN-I2C master bridge (protocol transmission ItoC_FIFO_SIZE=30 C2I_FIFO_SIZE=30 Data size = 12 bytes)	6456	2484
CAN-I2C slave bridge (protocol transmission ItoC_FIFO_SIZE=30 C2I_FIFO_SIZE=30 Data size = 12 bytes)	6368	2484

4 Hardware

By using a CAN analyzer, users can send and receive messages on the CAN side. As a demonstration, two LaunchPads can be used as two CAN-I2C bridges (one I2C master and one I2C slave) to form a loop. When the CAN analyzer sends CAN messages through the master LaunchPad™, the analyzer receives CAN messages from the slave LaunchPad. Figure 4-1shows the basic structure. Note that CAN transceivers are required to construct a CAN bus. Figure 4-2 shows the messages sent and received by CAN analyzer for the demo.

The accompanying demo uses two LaunchPads, a TCAN1046EVM and a CAN analyzer. TCAN1046EVM is a high-speed dual channel CAN transceiver evaluation module. Figure 4-3 shows the connection of the demo. For the LaunchPad, PA12 is used for the CAN transmit and PA13 is used for the CAN receive. PA12 and PA13 must be connected to the TX pin and the RX pin of the TCAN1046EVM. PB2 is used for I2C SCL (Serial Clock line). PB3 is used for I2C SDA (Serial Data line). A PB20 is used for IO trigger from a I2C slave to the master.

Since TCAN1046 supports level shifting, VCC must be connected to 5V and VIO must be connected to 3.3V. 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.

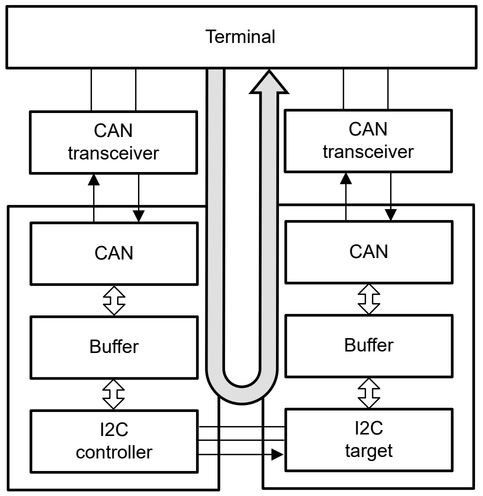
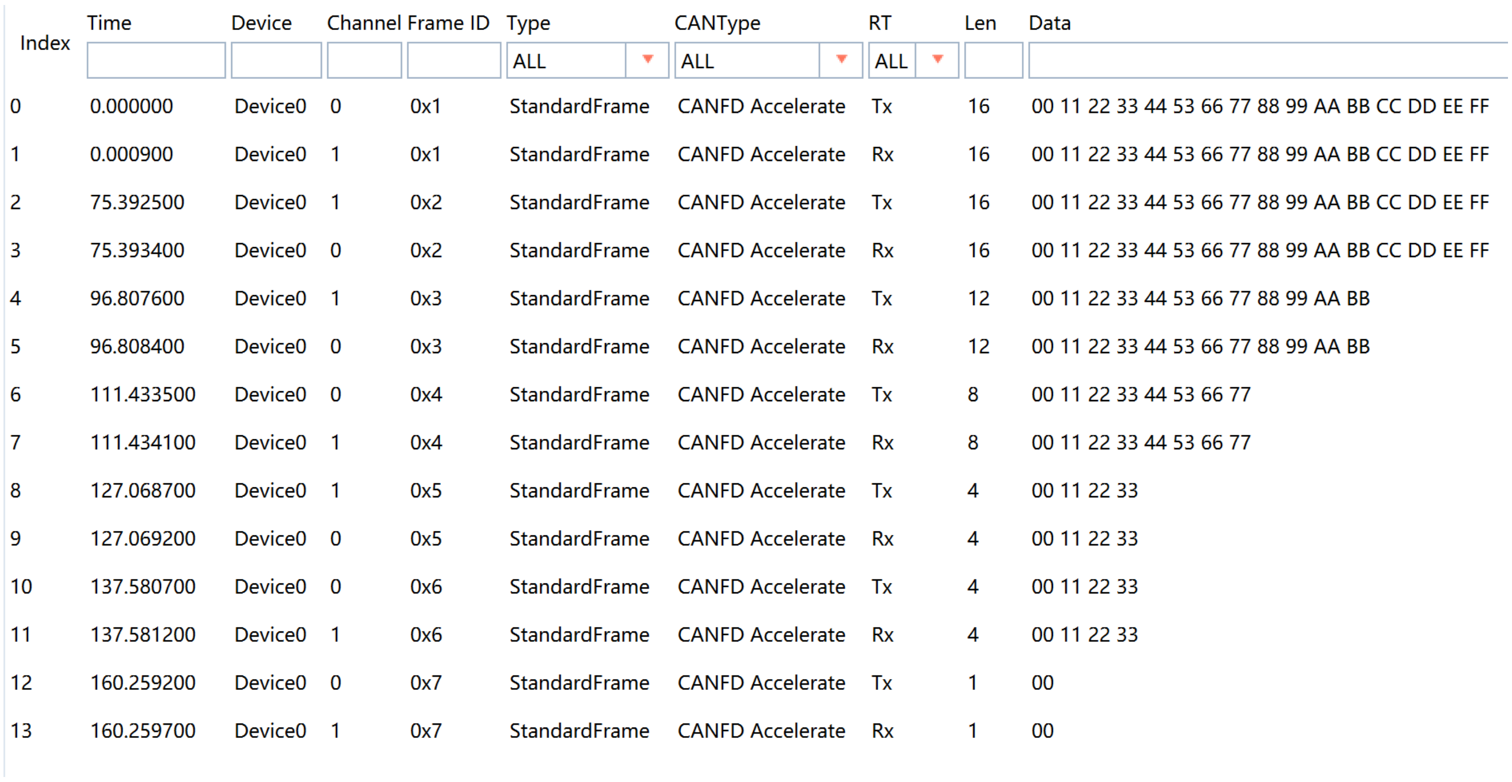


Figure 4-1. Basic Structure of Accompanying Demo



Index	Time	Device	Channel	Frame ID	Type	CANType	RT	Len	Data
					ALL	ALL	ALL		
0	0.000000	Device0	0	0x1	StandardFrame	CANFD Accelerate	Tx	16	00 11 22 33 44 53 66 77 88 99 AA BB CC DD EE FF
1	0.000900	Device0	1	0x1	StandardFrame	CANFD Accelerate	Rx	16	00 11 22 33 44 53 66 77 88 99 AA BB CC DD EE FF
2	75.392500	Device0	1	0x2	StandardFrame	CANFD Accelerate	Tx	16	00 11 22 33 44 53 66 77 88 99 AA BB CC DD EE FF
3	75.393400	Device0	0	0x2	StandardFrame	CANFD Accelerate	Rx	16	00 11 22 33 44 53 66 77 88 99 AA BB CC DD EE FF
4	96.807600	Device0	1	0x3	StandardFrame	CANFD Accelerate	Tx	12	00 11 22 33 44 53 66 77 88 99 AA BB
5	96.808400	Device0	0	0x3	StandardFrame	CANFD Accelerate	Rx	12	00 11 22 33 44 53 66 77 88 99 AA BB
6	111.433500	Device0	0	0x4	StandardFrame	CANFD Accelerate	Tx	8	00 11 22 33 44 53 66 77
7	111.434100	Device0	1	0x4	StandardFrame	CANFD Accelerate	Rx	8	00 11 22 33 44 53 66 77
8	127.068700	Device0	1	0x5	StandardFrame	CANFD Accelerate	Tx	4	00 11 22 33
9	127.069200	Device0	0	0x5	StandardFrame	CANFD Accelerate	Rx	4	00 11 22 33
10	137.580700	Device0	0	0x6	StandardFrame	CANFD Accelerate	Tx	4	00 11 22 33
11	137.581200	Device0	1	0x6	StandardFrame	CANFD Accelerate	Rx	4	00 11 22 33
12	160.259200	Device0	0	0x7	StandardFrame	CANFD Accelerate	Tx	1	00
13	160.259700	Device0	1	0x7	StandardFrame	CANFD Accelerate	Rx	1	00

Figure 4-2. Messages Sent and Received by CAN Analyzer for the Demo(CAN_ID_LENGTH = 0)

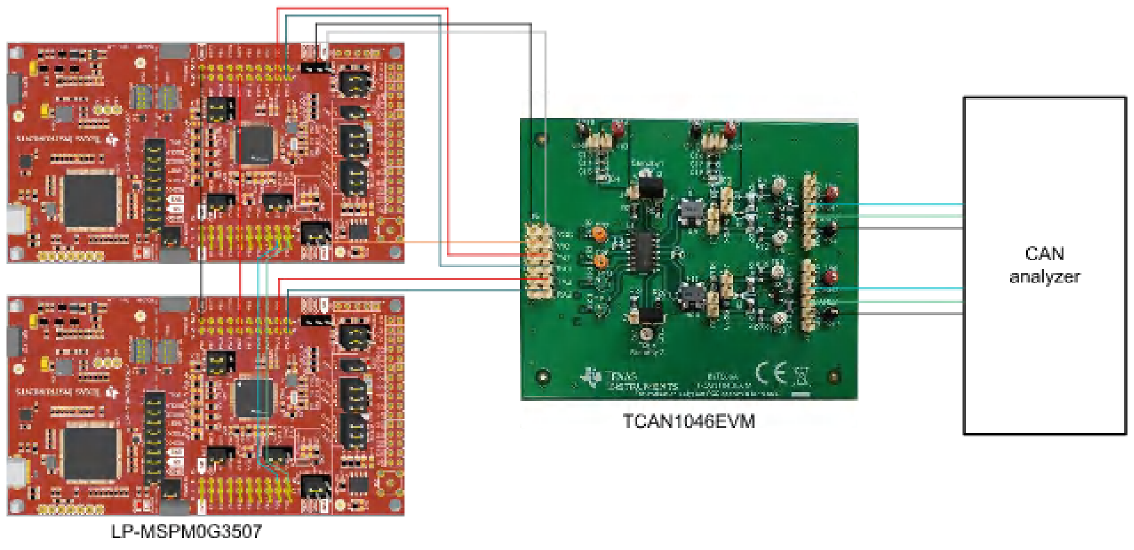


Figure 4-3. Hardware Connection of the Demo

5 Application Aspects

This section describes the application-level features of the CAN-I2C bridge design and how to configure the design to meet application requirements.

5.1 Flexible Structure

There are various configurable parameters, which have been mentioned in Section 3.2. Users can configure CAN and I2C packet frame, the size of the FIFO, and the maximum size of data area by modifying these parameters, which are all defined in *user_define.h*.

Users can also modify the define 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 FIFO. Users can make changes to the structure. For example, users can make messages follow a specific format or even a specific communication protocol. The structure can be divided into a one-way transmission according to Figure 2-3.

5.2 Optional Configuration for I2C

The I2C module acts as a master or slave interface for synchronous serial communication with peripheral devices and other controllers. The design provides with one code for CAN-I2C bridge (I2C master) and another code for CAN-I2C (I2C slave).

Besides, users can configure various functions of the I2C module. By using SysConfig, users can change the basic configuration of I2C. For more configuration, see related documentation.

5.3 Optional Configuration for CAN

The CAN module of the 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 provided has an optional configuration for the CAN ID. The sample code defaults to an 11-bit ID (standard ID). The configuration can be changed by modifying *user_define.h*.

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

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

5.4 CAN Bus Multinode Communication Example

CAN communication is a bus communication. Users can use this CAN-I2C bridge design to test the multinode 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-I2C bridge, the message is read back from other nodes immediately.

At least three LaunchPads need to be used. Each CAN communication on the LaunchPad requires a transceiver. The connection between the LaunchPad and the transceiver is shown in Figure 4-3.

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

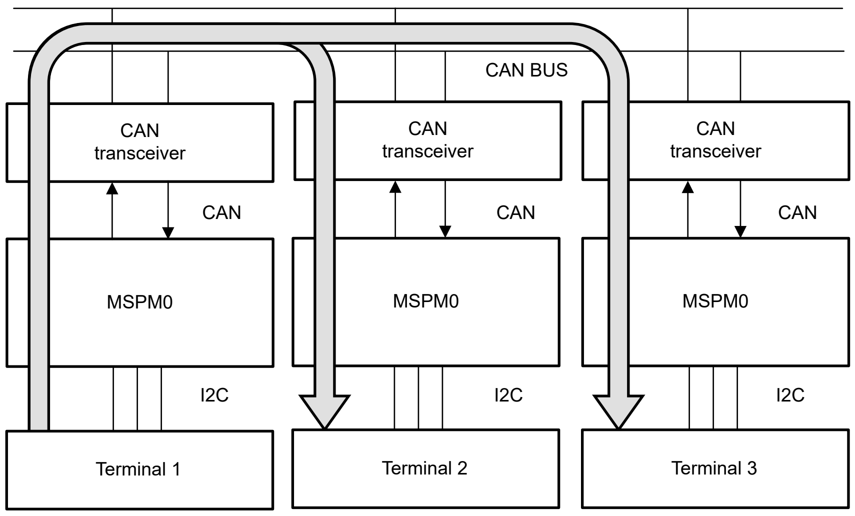


Figure 5-1. Basic Structure of Multinode Communication

6 Summary

This document introduces the implementation of the CAN to I2C bridge, including structure, function definition, interface usage and application aspects. MSPM0 can act as a translator between the CAN and the I2C, allowing for transmission and reception of information on one interface and to receive and send the information on the other interface.

7 References

- Bridge Solution between CAN and UART with MSPM0 MCUs
- Bridge Solution between CAN and SPI with MSPM0 MCUs
- Texas Instruments, CAN to UART Bridge, subsystem design.
- Texas Instruments, CAN to SPI Bridge, subsystem design.
- Texas Instruments, CAN to I2C Bridge, subsystem design.
- Download the MSPM0 SDK
- Texas Instruments, SysConfig tool, configuration tool.