Application Note

Bridge Design between CAN and SPI with MSPM0 MCUs

Yuhao Zhao

ABSTRACT

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

There are many communication methods between devices depending on the application. MCUs today usually support more than one communication method. For example, MSPM0 can support UART, SPI, CAN, and so on on a specific device. When devices need to transfer data over different communication interfaces, a bridge is constructed.

For CAN and SPI, a CAN-SPI bridge acts as a translator between the two interfaces. A CAN-SPI 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-SPI bridge. The MSPM0G3507 microcontroller (MCU) can be used by providing CAN and SPI communication interfaces. The accompanying demo uses the MSPM0G3507 with 2Mbps CANFD and 500k bit rate SPI to demonstrate transceiving data between channels.

1.1 Bridge between CAN and SPI

The CAN-SPI bridge connects the CAN and SPI interfaces. The bridge supports the SPI to function in either slave mode or master mode. The example in this document uses the CAN analyzer to observe the CAN data. Users can also send messages from the CAN analyzer over the CAN-SPI bridge to the SPI side. For SPI data, users can use a logic analyzer or use two LaunchPADs to form a loop to observe, such as the accompanying demo in Figure 4-1.

The example in this article 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 SPI message format. Users can modify the format according to application requirements. When receiving the message from the SPI, the message format is < 55 AA ID1 ID2 ID3 ID4 Length Data1 Data2 ...> Users can send data through SPI with the same format. 55 AA is the header. The 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 the SPI slave to detect one message. Data from the SPI is filled into the data area of the CAN (same in reverse). The CAN ID is the default value.

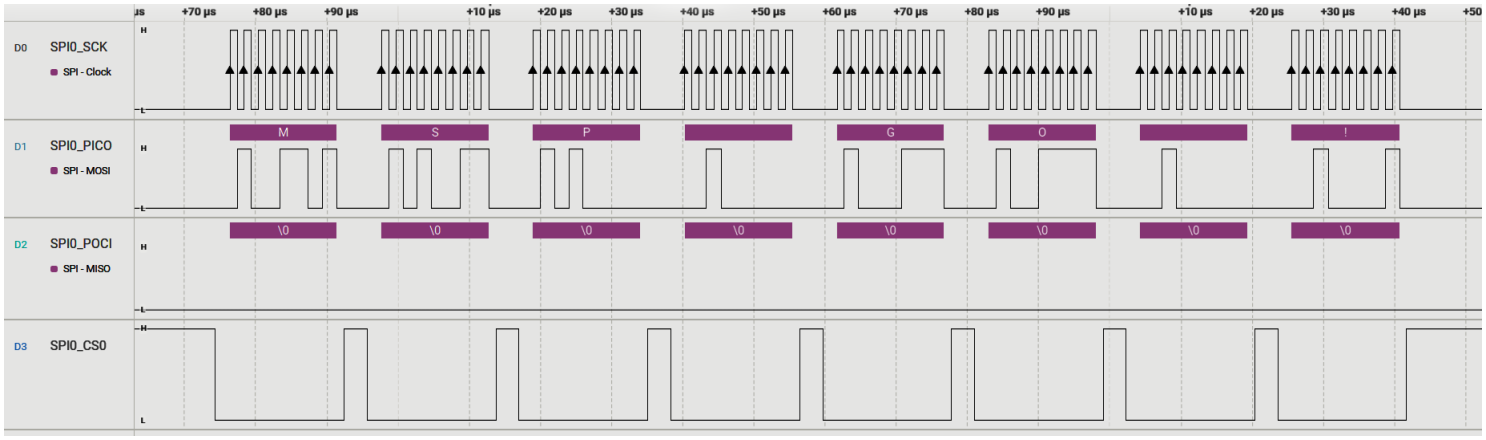

Figure 1-1. Logic Analyzer for SPI Transparent Transmission

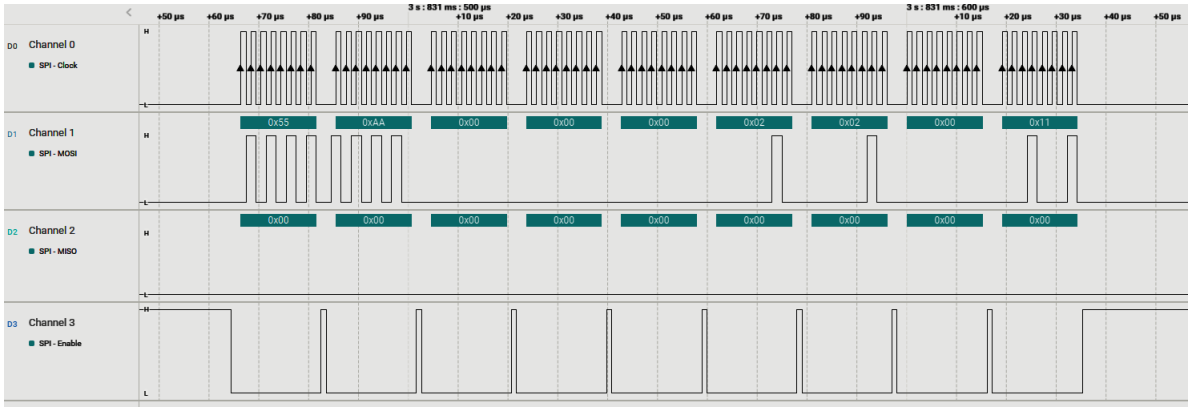

Figure 1-2. Logic Analyzer for SPI Protocol Transmission

2 Implementation

2.1 Principle

In this document, the CAN-SPI bridge uses both the CAN receive and transmit and the SPI receive and transmit. So both the CAN module and the SPI module must be configured. Since the message formats of different communications are different, the CAN-SPI 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 with ISO 11898-1:2015. For more information, see related documentation. For the SPI, the interface can be used transfer data between a MSPM0 device and another device with serial asynchronous communication protocols. For more information, see related documentation. Since the receiving and transmitting of the SPI slave are controlled by the SPI master, the SPI slave cannot initiate transmission to the SPI 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 that must be sent.

Figure 2-1 shows the basic principle of the CAN-SPI bridge. Typically, the communication rate of CAN is different from that of the SPI. For the CAN FD, the baud rate can be up to 5Mbps, while the SPI operates at 500k bit rate as shown in the example code. As a result, it is possible that the data received by one interface is not sent by another interface in time. To match the rate, this scheme uses a buffer to transfer data between the CAN and the SPI. 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.

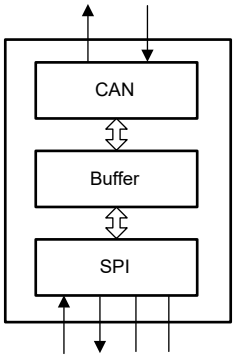

Figure 2-1. Basic Principle of the CAN-SPI Bridge

2.2 Structure

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

The structure of the CAN- SPI bridge with transparent transmission is shown in Figure 2-3 and Figure 2-5. Figure 2-3 is for the SPI master and Figure 2-5 is for the SPI slave. A timer interrupt is added to the CAN-SPI (SPI target) bridge to detect the timeout as the end of one packet.

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

For the CAN, a CAN frame is a fixed format. MSPM0 supports classic CAN or CANFD. Figure 2-2 shows the CANFD frame. The example in this document can define 0/1/4 bytes of additional ID in the data area for protocol transmission, which is listed in Table 2-1.

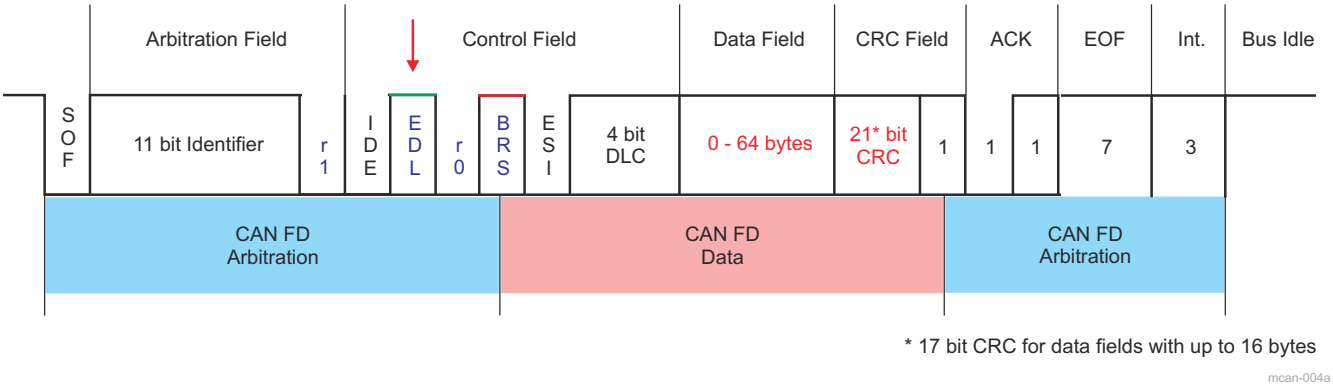

Figure 2-2. CAN FD Frame

Table 2-1. CAN Packet Form

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

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

Table 2-2. SPI 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 - (SPI_TRANSPARENT_LENGTH) bytes

The header is a fixed hex number combined with *0x55 0xAA*, which means the start of the group. The ID area occupies four bytes as default to match the CAN ID, which can be configured as one byte or doesn't 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 the SPI is a communication method where the SPI master controls the transmission and reception. In general, the SPI slave cannot initiate communication. For SPI slave-to-master communication, SPI slave pulls down the IO when messages must be sent, as shown in Figure 2-4 . The SPI master initiates the SPI read command in IO interrupt when IO is detected low, as shown in Figure 2-3.

For SPI transparent transmission, messages are identified when timeout occurs (on the SPI slave), as shown in Figure 2-4. All bytes are regards as pure data. 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. The format of FIFO element can be seen in Figure 2-6. In the format of the FIFO element, there are *origin_id*, *destination_id*, *data length* and *data* functions. 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 SPI transmission are performed in the main function. When 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 as shown in Table 2-1. For SPI, messages are sent in the format listed in SPI Packet Form.

Note that SPI is a communication method that sends and receives at the same time. When the master initiates sending a byte, the master expects to receive a byte. In the design of this document, SPI RX interrupt is not only used for the SPI to receive, but also used to fill the TX data into the SPI TX FIFO. If SPI functions in master mode, SPI communication starts immediately after SPI TX FIFO is stored by data. If the SPI works in slave mode, SPI uses IO to trigger the master to initiate communication after the data is stored. In this demo, users can select the SPI mode.

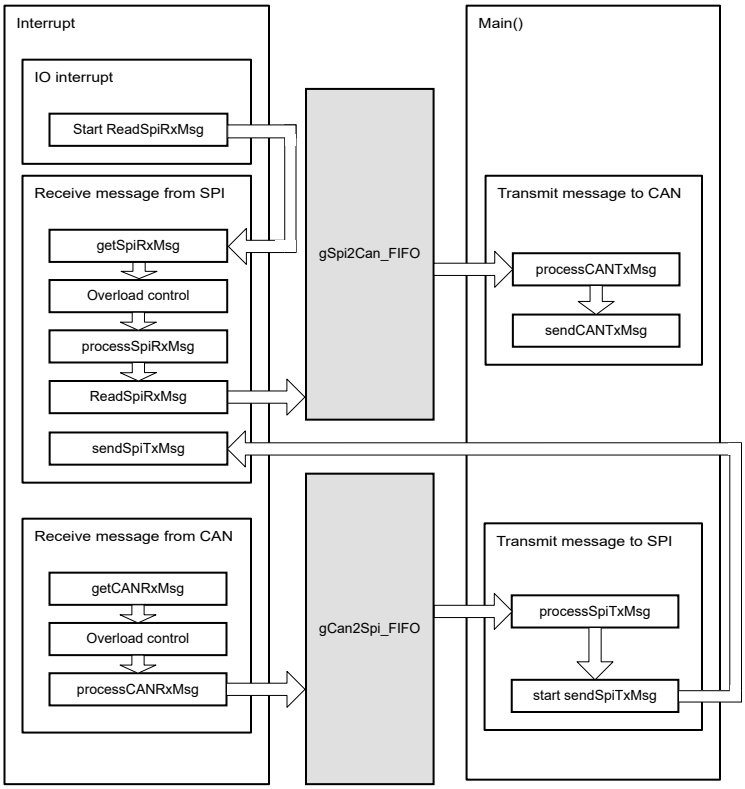

Figure 2-3. Structure of CAN-SPI (SPI Controller) Bridge: Protocol and Transparent

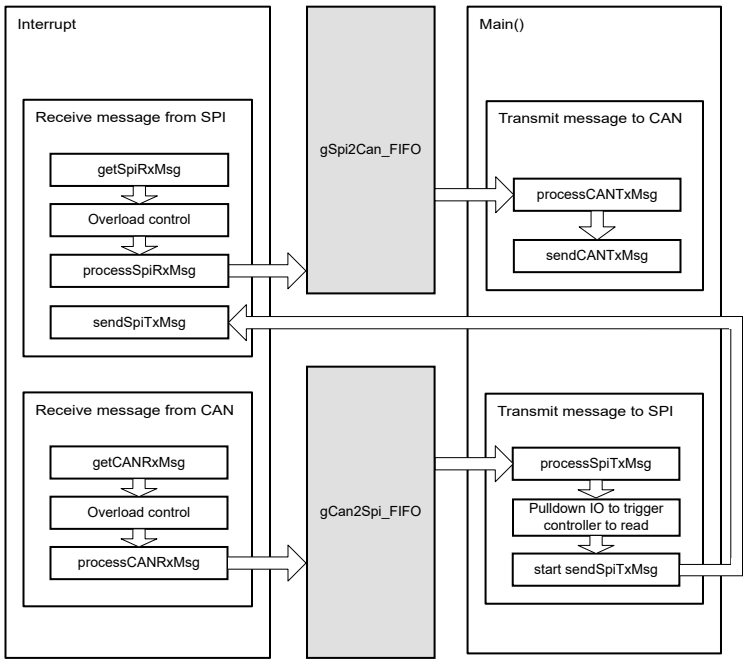

Figure 2-4. Structure of CAN-SPI (SPI Target) Bridge: Protocol

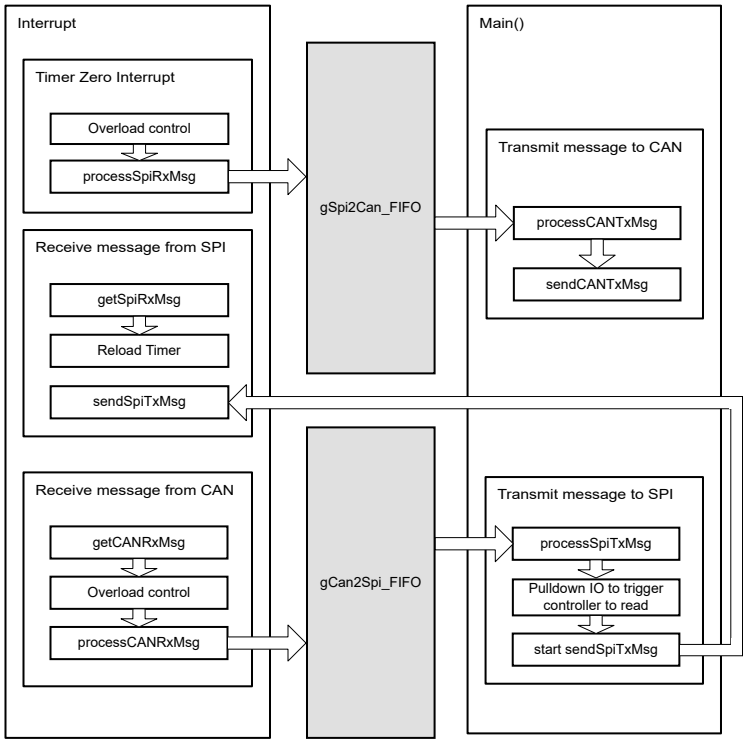

Figure 2-5. Structure of CAN-SPI (SPI Target) Bridge: Transparent

Figure 2-6 shows the FIFO structure. Each FIFO uses three global variables to indicate the FIFO status. For *gSpi2Can_FIFO*, *gSpi2Can_FIFO.fifo_in* indicates the write position, *gSpi2Can_FIFO.fifo_out* indicates the read position,and *gSpi2Can_FIFO.fifo_Count* indicates the number of elements in the *gSpi2Can_FIFO*.

If the *gSpi2Can_FIFO* is empty, *gSpi2Can_FIFO.fifo_in* equals *gSpi2Can_FIFO.fifo_out*, and *gSpi2Can_FIFO.fifo_count* is zero.

When performing *processSpiRxMsg()*, a new message from SPI is stored to *gSpi2Can_FIFO*. So the *gSpi2Can_FIFO.fifo_in* moves to the next position, and *gSpi2Can_FIFO.fifo_count* is incremented by 1.

When transmitting a message from *gSpi2Can_FIFO* to CAN, *gSpi2Can_FIFO.fifo_out* moves to the next position, and *gSpi2Can_FIFO.fifo_count* minus 1. *gCan2Spi_FIFO* is similar to *gSpi2Can_FIFO*.

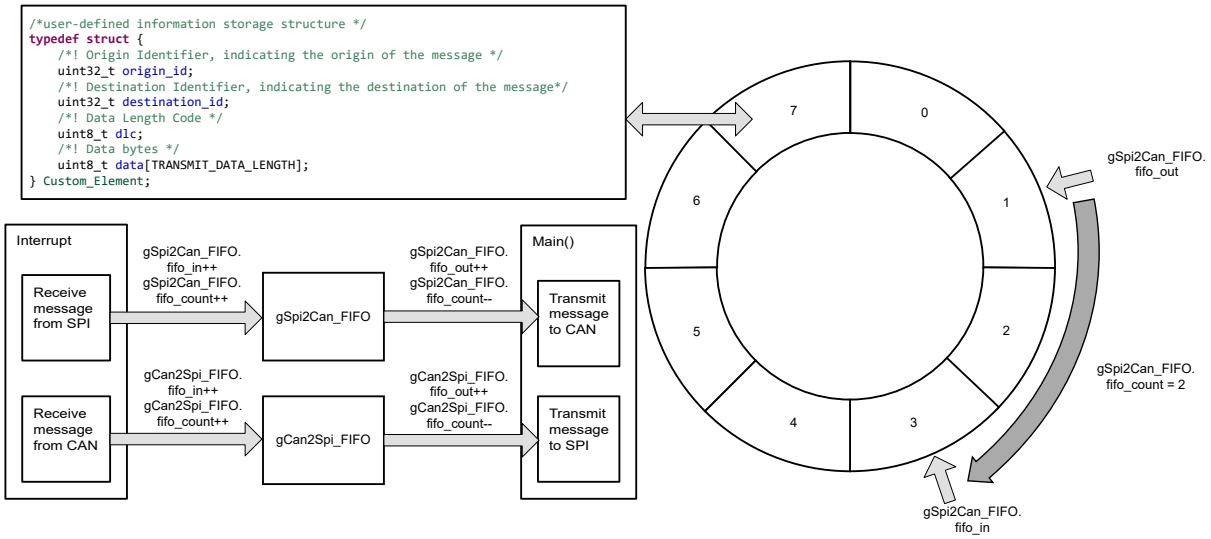

Figure 2-6. Structure of FIFO

3 Software Description

3.1 Software Functionality

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

Table 3-1. Functions and Descriptions

Tasks	Functions	Description	Location
SPI receive	readSpiRxMsg()	Send bytes to receive message (SPI master only)	bridge_spi.c bridge_spi.h
	getSpiRxMsg()	Obtain the received SPI message (protocol)	
	getSpiRxMsg_transparent()	Obtaub the received SPI message (transparent)	
	processSpiRxMsg()	Convert the received SPI message format (protocol) and store the message in gSPI_RX_Element	
	processSpiRxMsg_transparent()	Convert the received SPI message format (transparent) and store it in gSPI_RX_Element	
SPI transmit	processSpiTxMsg()	Convert the gSPI_TX_Element format (protocol) to be sent through SPI	
	processSpiTxMsg()	Convert the gSPI_TX_Element format (transparent) to be sent through SPI	
	sendSpiTxMsg()	Send message through SPI	
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 in 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 SPI, both transparent transmission and protocol transmission are supported in this example, switching by defining SPI_TRANSPARENT or SPI_PROTOCOL.

In transparent transmission, users can configure the timeout for the SPI slave to detect one SPI message receiving done. Users can also configure the default data length(SPI_TRANSPARENT_LENGTH) for the message from the SPI slave to the SPI 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. Note that there is a fixed 2-byte header(0x55 0xAA) and one byte of data-length. To modify the format more, users can modify the code directly.

```
#define SPI_TRANSPARENT
#ifdef SPI_TRANSPARENT
/* The format of SPI:
     * Transparent transmission - Data1 Data2 ...*/
/* data length for SPI master receiving or SPI slave transmitting*/
#define SPI_TRANSPARENT_LENGTH  (8)     //need be <= TRANSMIT_DATA_LENGTH
#define SPI_TIMEOUT     (0x4000)      //timeout 250ms
#else
/* The format of SPI:
     * if SPI_ID_LENGTH = 4, format is 55 AA ID1 ID2 ID3 ID4 Length Data1 Data2 ...
     * if SPI_ID_LENGTH = 1, format is 55 AA ID Length Data1 Data2 ...
     * if SPI_ID_LENGTH = 0, format is 55 AA Length Data1 Data2 ...*/
//#define SPI_ID_LENGTH  (0)
//#define SPI_ID_LENGTH  (1)
#define SPI_ID_LENGTH  (4)
#endif
```

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

Parameter	SPI Interface: Master		SPI 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+SPI_ID_LENGT+1+Length) bytes	(2+SPI_ID_LENGT+1+Length) bytes	(2+SPI_ID_LENGT+1+Length) bytes	(2+SPI_ID_LENGT+1+Length) bytes
Transparent Transmission	(SPI_TRANSPARENT_LENGTH) bytes	(Length) bytes	Timeout identify the end of message	(SPI_TRANSPARENT_LENGTH) bytes

For CAN, ID or 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 3-3. Configurable Parameters

Parameter	Optional value	Description
#define SPI_TRANSPARENT	Define / Not define	Enables the SPI transparent transmission.
#define SPI_PROTOCOL	Define / Not define	Enables the SPI protocol transmission.
#defineSPI_TRANSPARENT_LENGTH (8)		The default data length for message from the SPI slave to the SPI master. Only available when SPI_TRANSPARENT is defined. In this case, default value is eight bytes.
#define SPI_TIMEOUT (0x4000)	Timeout = SPI_TIMEOUT / 32768 s	Timeout to indicate one SPI message receiving done.Only available when SPI_TRANSPARENT is defined. In this case, default value is 250ms.
#define SPI_ID_LENGTH (4)	0/1/4	Optional SPI ID length, which is related to the ID area in protocol. Only available when SPI_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, the default value is 0 byte
#define TRANSMIT_DATA_LENGTH (12)	<=64	Size of data area. If the received message contains more data than this value, it can result in data loss
#define C2S_FIFO_SIZE (8)		Size of CAN to SPI FIFO. Note the usage of SRAM.
#define S2C_FIFO_SIZE (8)		Size of SPI to CAN FIFO. Note the usage of SRAM.
#define DEFAULT_SPI_ORIGIN_ID (0x00)		Default value for SPI origin ID
#define DEFAULT_SPI_DESTINATION_ID (0x00)		Default value for SPI 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 2-6.

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

- Example 1: CAN interface receive and transmit
  1. When the CAN-SPI bridge receives a CAN message, the ID from CAN frame is *Origin Identifier*, indicating where the message comes from.
  2. When the CAN-SPI bridge transmits a CAN message, *Origin Identifier* is ignored (CAN_ID_LENGTH is set to 0 default).
- Example 2: SPI Interface Receive and Transmit (SPI protocol transmission)
  1. When the CAN-SPI bridge receives an SPI message (SPI protocol transmission), DEFAULT_SPI_ORIGIN_ID is the *Origin Identifier* since the SPI does not have an ID.
  2. When the CAN-SPI bridge transmits the SPI message(SPI protocol transmission), *Origin Identifier* is a 4-byte ID in SPI data (SPI_ID_LENGTH is set to four as the default), indicating where the message came from.
- Example 3 - SPI interface receives and transmits (SPI transparent transmission)
  1. When the CAN-SPI bridge receives an SPI message (SPI transparent transmission) DEFAULT_SPI_ORIGIN_ID is the *Origin Identifier* since SPI does not have an ID.
  2. When the CAN-SPI bridge transmits the SPI message (SPI transparent transmission), *Origin Identifier* is ignored. (Transparent transmission does not have an ID area).

Destination Identifier indicates the destination of the message.

- Example 1 - CAN interface receive and transmit
  1. When a CAN-SPI bridge receives a CAN message, DEFAULT_CAN_DESTINATION_ID is the *Destination Identifier* since CAN_ID_LENGTH is set to 0 as the default. SPI transmit does not require an ID.
  2. When the CAN-SPI bridge transmits a CAN message, *Destination Identifier* is the CAN ID in CAN frame. In this example, 11 bit or 29 bit are both supported.
- Example 2 - SPI interface receive and transmit (SPI protocol transmission)
  1. When the CAN-SPI bridge receives an SPI message (SPI protocol transmission), 4-byte ID from the SPI data is Destination Identifier(SPI _ID_LENGTH is set to 4 default). CAN transmit requires ID information.
  2. When the CAN-SPI bridge transmits an SPI message (SPI protocol transmission), *Destination Identifier* is ignored since SPI transmit does not require an ID.
- Example 3 - SPI interface receive and transmit (SPI transparent transmission)
  1. When the CAN-SPI bridge receives an SPI message (SPI transparent transmission), DEFAULT_SPI_DESTINATION_ID is the *Destination Identifier* (Transparent transmission does not have ID area). CAN transmit requires ID information.
  2. When the CAN-SPI bridge transmits a SPI message (SPI transparent transmission), *Destination Identifier* is ignored since SPI 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*. The definition is also shown in Figure 2-6.

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

gCan2Spi_FIFO and *gSpi2Can_FIFO* are defined in *main.c*. Note the usage of SRAM, which is related to *C2S_FIFO_SIZE*, *S2C_FIFO_SIZE* and the size for *Custom_Element*.

```
/* Variables for C2S_FIFO
 * C2S_FIFO is used to temporarily store message from CAN to SPI */
Custom_Element gC2S_FIFO[C2S_FIFO_SIZE];
Custom_FIFO gCan2Spi_FIFO = {0, 0, 0, gC2S_FIFO};

/* Variables for S2C_FIFO
 * S2C_FIFO is used to temporarily store message from SPI to CAN */
Custom_Element gS2C_FIFO[S2C_FIFO_SIZE];
Custom_FIFO gSpi2Can_FIFO = {0, 0, 0, gS2C_FIFO};
```

3.5 SPI Receive and Transmit (Transparent Transmission)

In general, the SPI master controls the SPI communication, and the SPI 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 SPI receive, there are three global variables defined in *bridge_spi.c*.

```
uint8_t gSpiReceiveGroup[SPI_RX_SIZE];
Custom_Element gSPI_RX_Element;
uint16_t gGetSpiRxMsg_Count;
```

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

1. In IO interrupt, call *readSpiRxMsg()* to send (SPI_TRANSPARENT_LENGTH) bytes to receive message from the SPI slave (SPI send and receive together).
2. Call *getSpiRxMsg_transparent()* to store a message into *gSpiReceiveGroup*. Message receiving is finished when (SPI_TRANSPARENT_LENGTH) bytes are received.
3. Call *processSpiRxMsg_transparent()* to extract data from *gSpiReceiveGroup* and store the data in *gSPI_RX_Element*.
4. Put *gSPI_RX_Element* into *gSpi2Can_FIFO*.

The following is the process for the SPI slave to receive data

1. Call *getSpiRxMsg_transparent()* to store message into *gSpiReceiveGroup*. Message receiving is finished when timeout occurs.
2. Call *processSpiRxMsg_transparent()* to extract data from *gSpiReceiveGroup* and store the data into *gSPI_RX_Element*.
3. Put *gSPI_RX_Element* into *gSpi2Can_FIFO*.

For SPI transmit, there are two global variables defined in bridge_spi.c.

```
uint8_t gSpiTransmitGroup[SPI_TX_SIZE];
Custom_Element gSPI_TX_Element;
```

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

1. Obtain *gSPI_TX_Element* from *gCan2Spi_FIFO*.
2. Call *processSpiTxMsg_transparent()* to obtain data from *gSPI_TX_Element* and store the data into *gSpiTransmitGroup*.
3. Call *sendSpiTxMsg()* to transmit *gSpiTransmitGroup* through SPI. For the SPI slave, only *(SPI_TRANSPARENT_LENGTH)* bytes are sent.
4. (SPI slave only) Use IO to trigger the master to read from the slave.

3.6 SPI Receive and Transmit (Protocol Transmission)

In general, SPI master controls the SPI communication, and SPI 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 SPI receive, there are two global variables defined in *bridge_spi.c*.

```
uint8_t gSpiReceiveGroup[SPI_RX_SIZE];
Custom_Element gSPI_RX_Element;
```

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

1. In IO interrupt, call *readSpiRxMsg()* to send bytes to receive message from the SPI slave (SPIs send and receive together)
2. Call *getSpiRxMsg()* to detect header and to store complete message into *gSpiReceiveGroup*.
3. Call *processSpiRxMsg ()* to extract information from *gSpiReceiveGroup* and store the data in *gSPI_RX_Element*.
4. Place *gSPI_RX_Element* into *gSpi2Can_FIFO*.

The following is the process for SPI slave reception.

1. Call *getSpiRxMsg()* to store message into *gSpiReceiveGroup*. Message reception is finished when timeout occurs.
2. Call *processSpiRxMsg()* to extract data from *gSpiReceiveGroup* and store the data in *gSPI_RX_Element*.
3. Put *gSPI_RX_Element* into *gSpi2Can_FIFO*.

For SPI transmission, there are two global variables defined in *bridge_spi.c*.

```
uint8_t gSpiTransmitGroup[SPI_TX_SIZE];
Custom_Element gSPI_TX_Element;
```

The following is the process for SPI master and slave transmission

1. Get *gSPI_TX_Element* from *gCan2Spi_FIFO*.
2. Call *processSpiTxMsg()* to obtain information from *gSPI_TX_Element* and store the data in *gSpiTransmitGroup*.
3. Call sendSpiTxMsg() to transmit *gSpiTransmitGroup* through SPI.
4. (SPI slave only) Use an IO to trigger the master to read from the slave.

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. Place *gCAN_RX_Element* into *gCan2Spi_FIFO*.

For CAN transmit, there are 2 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. Obtain *gCAN_TX_Element* from *gSpi2Can_FIFO*.
2. Call *processCANTxMsg()* to receive information from *gCAN_TX_Element* and store it into *txMsg0*.

3. Call *sendCANTxMsg()* to transmit *txMsg0* through the CAN.

3.8 Application Integration

Functions listed in Table 3-1 are categorized into different files. Functions for SPI receive and transmit are included in *bridge_spi.c* and *bridge_spi.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 separate functions by file. For example, if only SPI functions are required, users can reserve *bridge_spi.c* and *bridge_spi.h* to call the functions.

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

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

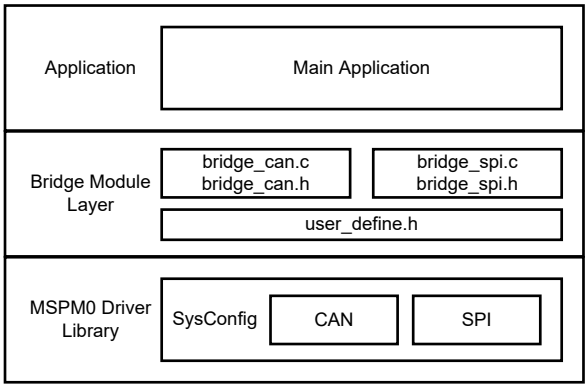

Figure 3-1. Files Required by the Software

Table 3-4 lists the footprint of the CAN-SPI bridge design in terms of flash size and RAM size. Table 3-4 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 requires more RAM space. For details, 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 data field with less bytes can significantly reduce RAM usage.

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

Minimum Required Code Size (Bytes)	Flash	SRAM
CAN-SPI master bridge (Protocol Transmission S2C_FIFO_SIZE=8 C2S_FIFO_SIZE=8 Data size = 12 bytes)	6128	1466
CAN-SPI slave bridge (Protocol Transmission S2C_FIFO_SIZE=8 C2S_FIFO_SIZE=8 Data size = 12 bytes)	6344	1462
CAN-SPI master bridge (Protocol Transmission S2C_FIFO_SIZE=8 C2S_FIFO_SIZE=8 Data size = 64 bytes)	6224	2610
CAN-SPI slave bridge (Protocol Transmission S2C_FIFO_SIZE=8 C2S_FIFO_SIZE=8 Data size = 64 bytes)	6440	2606
CAN-SPI master bridge (Protocol Transmission S2C_FIFO_SIZE=30 C2S_FIFO_SIZE=30 Data size = 12 bytes)	6232	2522
CAN-SPI slave bridge (Protocol Transmission S2C_FIFO_SIZE=30 C2S_FIFO_SIZE=30 Data size = 12 bytes)	6448	2518

4 Hardware

By using a CAN analyzer, users can send and receive messages on the CAN side. As a demonstration, two LaunchPad™s can be used as two CAN-SPI bridges(one SPI master and one SPI 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 4-1 shows 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. A PA12 LaunchPad is used for the CAN transmit and a PA13 LaunchPad is used for the CAN receive. PA12 and PA13 must be connected to the TX pin and the RX pin of TCAN1046EVM. PB9 is used for SPI SCLK (clock). PB8 is used for SPI MOSI (master out, slave in). PB7 is used for SPI MISO(master in, slave out). PB6 is used for SPI CS (chip select). PB20 is used for IO trigger from I2C slave to master.

For TCAN1046EVM, (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.

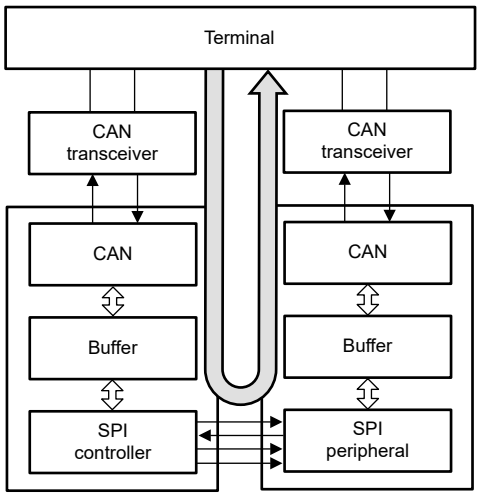

Figure 4-1. Basic Structure of Accompanying Demo

Index	Time	Device	Channel	Frame ID	Type	CANType	RT	Len	Data
					ALL	ALL	ALI		
0	0.000000	Device0	0	0x1	StandardFrame	CANFD Accelerate	Tx	1	00
1	0.000300	Device0	1	0x1	StandardFrame	CANFD Accelerate	Rx	1	00
2	18.323700	Device0	0	0x2	StandardFrame	CANFD Accelerate	Tx	2	00 11
3	18.324100	Device0	1	0x2	StandardFrame	CANFD Accelerate	Rx	2	00 11
4	33.411500	Device0	0	0x3	StandardFrame	CANFD Accelerate	Tx	4	00 11 22 33
5	33.411900	Device0	1	0x3	StandardFrame	CANFD Accelerate	Rx	4	00 11 22 33
6	50.216400	Device0	0	0x4	StandardFrame	CANFD Accelerate	Tx	8	00 11 22 33 44 55 66 77
7	50.216900	Device0	1	0x4	StandardFrame	CANFD Accelerate	Rx	8	00 11 22 33 44 55 66 77
8	67.378700	Device0	0	0x5	StandardFrame	CANFD Accelerate	Tx	12	00 11 22 33 44 55 66 77 88 99 AA BB
9	67.379400	Device0	1	0x5	StandardFrame	CANFD Accelerate	Rx	12	00 11 22 33 44 55 66 77 88 99 AA BB
10	344.182200	Device0	0	0x6	StandardFrame	CANFD Accelerate	Tx	32	00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF...
11	344.183400	Device0	1	0x6	StandardFrame	CANFD Accelerate	Rx	32	00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF...

Figure 4-2. Messages Sent and Received by CAN Analyzer for the Demo

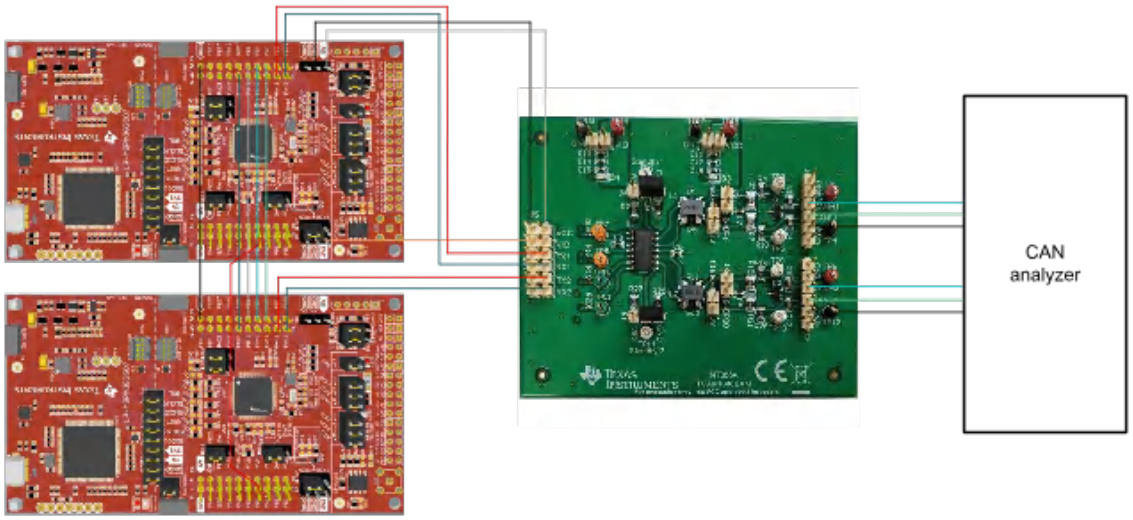

Figure 4-3. Hardware Connection of the Demo

5 Application Aspects

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

5.1 Flexible structure

There are various configurable parameters that are listed in Table 3-3. Users can configure a CAN or SPI packet frame, the size of the FIFO, or the maximum size of the data area by modifying these parameters which are all defined in *user_define.h*.

Users can also 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 FIFO. As a result, users can make changes to the structure. For example, making messages follow a specific format or even a specific communication protocol. In addition, the structure can be split into a one-way transmission according to Figure 2-3.

5.2 Optional Configuration for SPI

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

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

5.3 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 is provided 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 application requirements, which can further reduce the RAM space that is occupied by the FIFO.

5.4 CAN Bus Multinode Communication Example

CAN communication is a bus communication. Users can use this CAN-SPI bridge design to test the multinode communication of CAN bus. Figure 5-1 shows the basic structure. When the user sends a message to the CAN bus through any CAN-SPI 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. See Figure 4-3 to view the connection between the LaunchPad and the transceiver.

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 configurations, see related documentation.

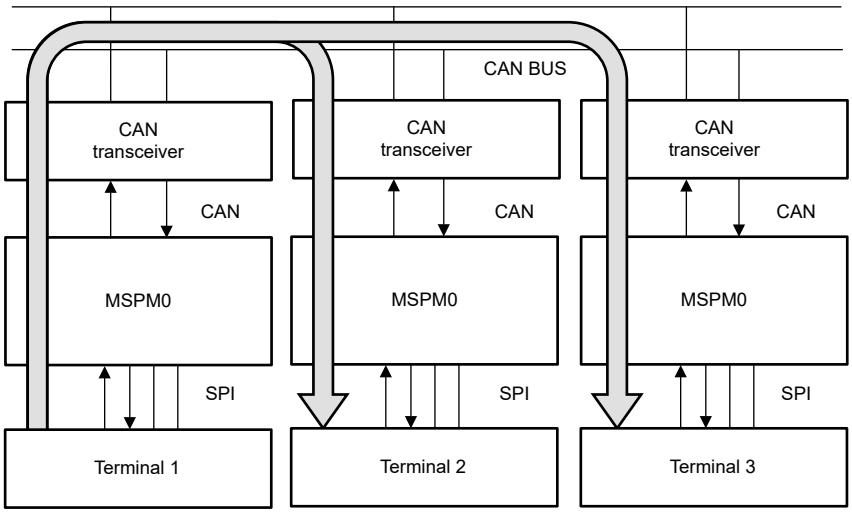

Figure 5-1. Basic Structure of Multinode Communication

6 Summary

This document introduces the implementation of a CAN to SPI bridge, including structure, function definition, interface usage and application aspects. The MSPM0 device can act as a translator between the CAN and the SPI, which allows the bridge to send and receive information on one interface and receive and send the information on the other interface.

7 References

- Texas Instruments, Bridge Design Between CAN and UART with MSPM0 MCUs, application note.
- Texas Instruments, Bridge Design 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, MSPM0 SDK, tool.
- Texas Instruments, SysConfig System Configuration Tool, webpage.

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