SPI to CAN Bridging Transmission

Example Code Introduction for 32-bit NuMicro® Family

Document Information

Application	This example code demonstrates the bridging transmission between SPI and CAN using the M0A23 series microcontroller.
BSP Version	M0A23_Series_BSP_CMSIS_V3.03.000
Hardware	NuMaker-M0A23OC V1.1

The information described in this document is the exclusive intellectual property of Nuvoton Technology Corporation and shall not be reproduced without permission from Nuvoton.

Nuvoton is providing this document only for reference purposes of NuMicro microcontroller and microprocessor based system design. Nuvoton assumes no responsibility for errors or omissions.

All data and specifications are subject to change without notice.

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

1. Overview

This example code demonstrates how to create an SPI to CAN bridge, allowing a device to receive data through the SPI interface and forward it via the CAN interface to another device. This enables communication between devices for sending and receiving messages. The bridge functions as a communication link between SPI interface and CAN interface, allowing data transfer between the two protocols.

1.1 Principle

Once SPI receives the data and verifies that it has been correctly parsed, it encapsulates the data into a CAN frame and transmits it using the CAN_Transmit() function. This process ensures that only valid and properly formatted data is forwarded to the CAN bus, maintaining data integrity across communication interfaces. By converting SPI received data into CAN frames, the system can effectively bridge communication between SPI-based devices and CAN, enabling interoperability in multi-protocol embedded systems.

The transmitted CAN data will be formatted as shown in Table 1-1.

Frame Type	ID Type	ID	DLC	Data
CAN_Data_FRAME	CAN_STD_ID	0x7FF	8	g_au32DestinationData

Table 1-1 CAN Package Format

The process and decision flow is illustrated in Figure 1-1. The following briefly describes the program flow of the CAN master and slave sides:

CAN Master

Step 1: Wait for the CAN slave to send data through the USI_SPI communication interface.

Step 2: After receiving the USI_SPI data, transmit the data to the CAN slave via the CAN communication interface.

Step 3: Wait for a response packet from the CAN slave. Once received, send the next CAN packet.

Step 4: Continue until all data has been transmitted. Then wait for the CAN slave to send USI_SPI data and return to Step 1.

CAN Slave

Step 1: Send data to the CAN master through the USI_SPI communication interface.

Step 2: After completing USI_SPI data transmission, wait to receive packets from the CAN master.

Step 3: Send a response packet to the CAN master, then wait for the next packet from the CAN master and return to Step 1.

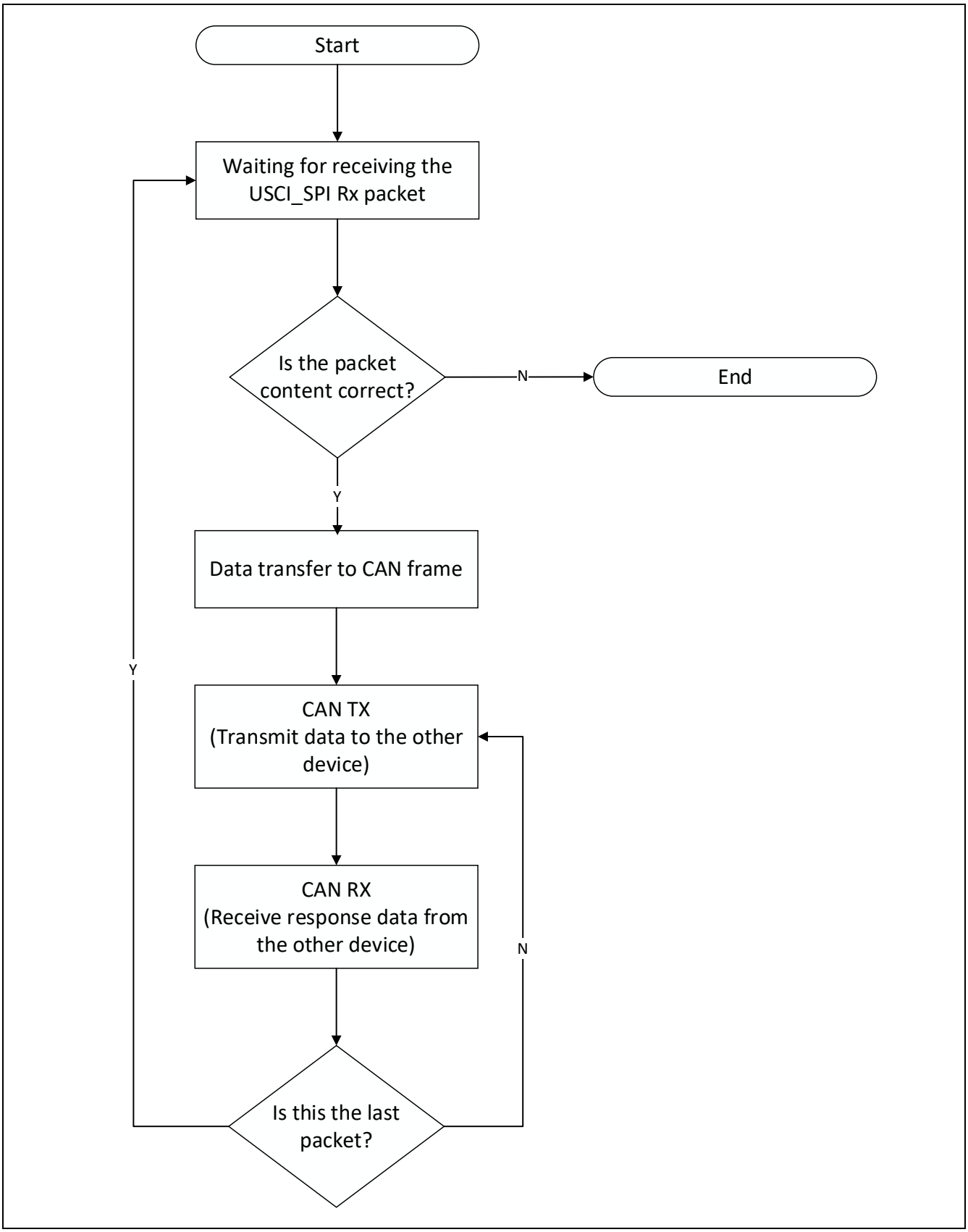


Figure 1-1 Process of Converting USCI_SPI Reception to CAN Transmission

1.2 Demo Result

Start the computer terminal software, set UART baud rate to 115200 bps, no parity bit, the data length is 8 bits, and the stop bit is 1 bit. The packet contents received by the CAN master and slave are shown in Figure 1-2 and Figure 1-3, respectively.

```
CAN master is waiting for reciving USCI_SPI package...
Press any key to continue ...

USCI_SPI data compCAN master is waiting for reciving USCI_SPI package...
Press any key to continue ...
CAN master is waiting for reciving USCI_SPI package...
Press any key to continue ...

USCI_SPI data comparison successful.

Pkt=0 Read ID=     788, Type=STD, DLC=2,Data=4F,4B,

Pkt=8 Read ID=     788, Type=STD, DLC=2,Data=4F,4B,

Pkt=16 Read ID=     788, Type=STD, DLC=2,Data=4F,4B,

Pkt=24 Read ID=     788, Type=STD, DLC=2,Data=4F,4B,

Pkt=32 Read ID=     788, Type=STD, DLC=2,Data=4F,4B,

Pkt=40 Read ID=     788, Type=STD, DLC=2,Data=4F,4B,

Pkt=48 Read ID=     788, Type=STD, DLC=2,Data=4F,4B,

Pkt=56 Read ID=     788, Type=STD, DLC=2,Data=4F,4B,

Transfor done

CAN master is waiting for reciving USCI_SPI package...
Press any key to continue ...
```

Figure 1-2 CAN Master Screenshot of the Terminal

```
Press any key to continue ...

Send Msg:
0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0a 0x0b 0x0c 0x0d 0x0e 0x0f
0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17 0x18 0x19 0x1a 0x1b 0x1c 0x1d 0x1e 0x1f
0x20 0x21 0x22 0x23 0x24 0x25 0x26 0x27 0x28 0x29 0x2a 0x2b 0x2c 0x2d 0x2e 0x2f
0x30 0x31 0x32 0x33 0x34 0x35 0x36 0x37 0x38 0x39 0x3a 0x3b 0x3c 0x3d 0x3e 0x3f
Read ID=     7FF, Type=STD, DLC=8,Data=00,01,02,03,04,05,06,07,

Read ID=     7FF, Type=STD, DLC=8,Data=08,09,0A,0B,0C,0D,0E,0F,

Read ID=     7FF, Type=STD, DLC=8,Data=10,11,12,13,14,15,16,17,

Read ID=     7FF, Type=STD, DLC=8,Data=18,19,1A,1B,1C,1D,1E,1F,

Read ID=     7FF, Type=STD, DLC=8,Data=20,21,22,23,24,25,26,27,

Read ID=     7FF, Type=STD, DLC=8,Data=28,29,2A,2B,2C,2D,2E,2F,

Read ID=     7FF, Type=STD, DLC=8,Data=30,31,32,33,34,35,36,37,

Read ID=     7FF, Type=STD, DLC=8,Data=38,39,3A,3B,3C,3D,3E,3F,

Press any key to continue ...
```

Figure 1-3 CAN Slave Screenshot of the Terminal

2. Code Description

Section 2.1 and 2.2 will describe the initialization settings, transmission and reception function for USCI_SPI and CAN, respectively.

2.1 USCI_SPI

The initialization settings and reception function are as follows and the code is located in *USCI_SPI_transfer*.c.

- Configure USCI_SPI0 as a slave, clock rate as 2 MHz.

```
void USCI_SPI_Init(void)
{
    …

    /* Configure USCI_SPI0 as a master, clock idle low, 16-bit transaction,
         drive output on falling clock edge and latch input on rising edge. */
    /* Set USCI_SPI0 clock rate = 2MHz */
    USPI_Open(USPI0, USPI_SLAVE, USPI_MODE_0, 16, 2000000);

    …
}
```

The USCI_SPI slave compares the received data with the transmitted data. If the USCI_SPI packet contents match correctly, the CAN packet transmission and reception process will begin.

```
uint8_t USCI_SPI_Rx(void)
{

    /* Access TX and RX Buffer */
    while (u32RxDataCount < TEST_COUNT)
    {
        /* Check RX EMPTY flag */
        if (USPI_GET_RX_EMPTY_FLAG(USPI0) == 0){
            /* Read RX Buffer */
            g_au32DestinationData[u32RxDataCount++] = USPI_READ_RX(USPI0);
         }
    }

    /* Check the USPI0 received data */
    for (u32RxDataCount = 0; u32RxDataCount < TEST_COUNT; u32RxDataCount++)
    {
        if (g_au32DestinationData[u32RxDataCount] != u32RxDataCount)
        {
            u32Err = 1;
            break;
        }
    }
    …
}
```

2.2 CAN

The initialization settings, transmission and reception function are as follows and the code is located in *CAN_transfer*.c.

- Configure CAN function speed as 500 kbps and normal mode for forwarding USCI_SPI data to the other device.

```
void CAN_Init(CAN_T  *tCAN)
{
    /* Enable IP clock */
    CLK->APBCLK0 |= CLK_APBCLK0_CAN0CKEN_Msk;
}

void SelectCANSpeed(CAN_T  *tCAN)
{
    /* Set CAN speed = 500 Kbps */
    CAN_Open(tCAN,  500000, CAN_NORMAL_MODE);
}
```

For CAN related configuration values, please refer to Table 1-1.

```
void CAN_Master_NormalMode_Tx(CAN_T *tCAN)
{
    STR_CANMSG_T tMsg;
    static uint32_t i = 0;

    /* Send a 8 bytes message */
    tMsg.FrameType = CAN_DATA_FRAME;
    tMsg.IdType    = CAN_STD_ID;
    tMsg.Id        = 0x7FF;
    tMsg.DLC       = 8;

    if (u32TxLength < TEST_COUNT)
    {

        for (i = 0; i < 8; i++)
        {
            tMsg.Data[i]  = g_au32DestinationData[i + u32TxLength];
        }
         /* Configure Msg RAM and send the Msg in the RAM */
        if (CAN_Transmit(tCAN, MSG(0), &tMsg) == FALSE)
        {
            printf("Set Tx Msg Object failed\n");
            return;
        }

        u32TxLength += 8;

    }
}
```

After the CAN packet is transmitted, the system waits for a response message from the other device. Once the acknowledgment is verified, the next packet transmission begins.

```
unsigned char CAN_Master_NormalMode_Rx(CAN_T *tCAN)
{

    …
    /* Polling Mode */
    while (1)
    {
         /* Wait IDR is changed */
        while (tCAN->IIDR != 0x01);

        CAN_Receive(tCAN, tCAN->IIDR - 1, &rrMsg);

        if (rrMsg.Data[0] == 'O' && rrMsg.Data[1] == 'K')
        {
            CAN_ShowMsg(&rrMsg);
            u32RxLength++;
            break;
        }
    }
    …
     return 0;
}
```

3. Software and Hardware Requirements

3.1 Software Requirements

- BSP version
  - M0A21_Series_BSP_CMSIS_V3.03.000
- IDE version
  - Keil uVersion 5.38

3.2 Hardware Requirements

- Circuit components
  - NuMaker-M0A23OC V1.1
- Pin Connect
  - USCI0_DAT0 (PC.5) pin corresponds to the SPI_MOSI function pin
  - USCI0_DAT1 (PC.4) pin corresponds to the SPI_MISO function pin
  - USCI0_CLK (PC.6) pin corresponds to the SPI_CLK function pin
  - USCI0_CTL0 (PC.7) pin corresponds to the SPI_SS function pin

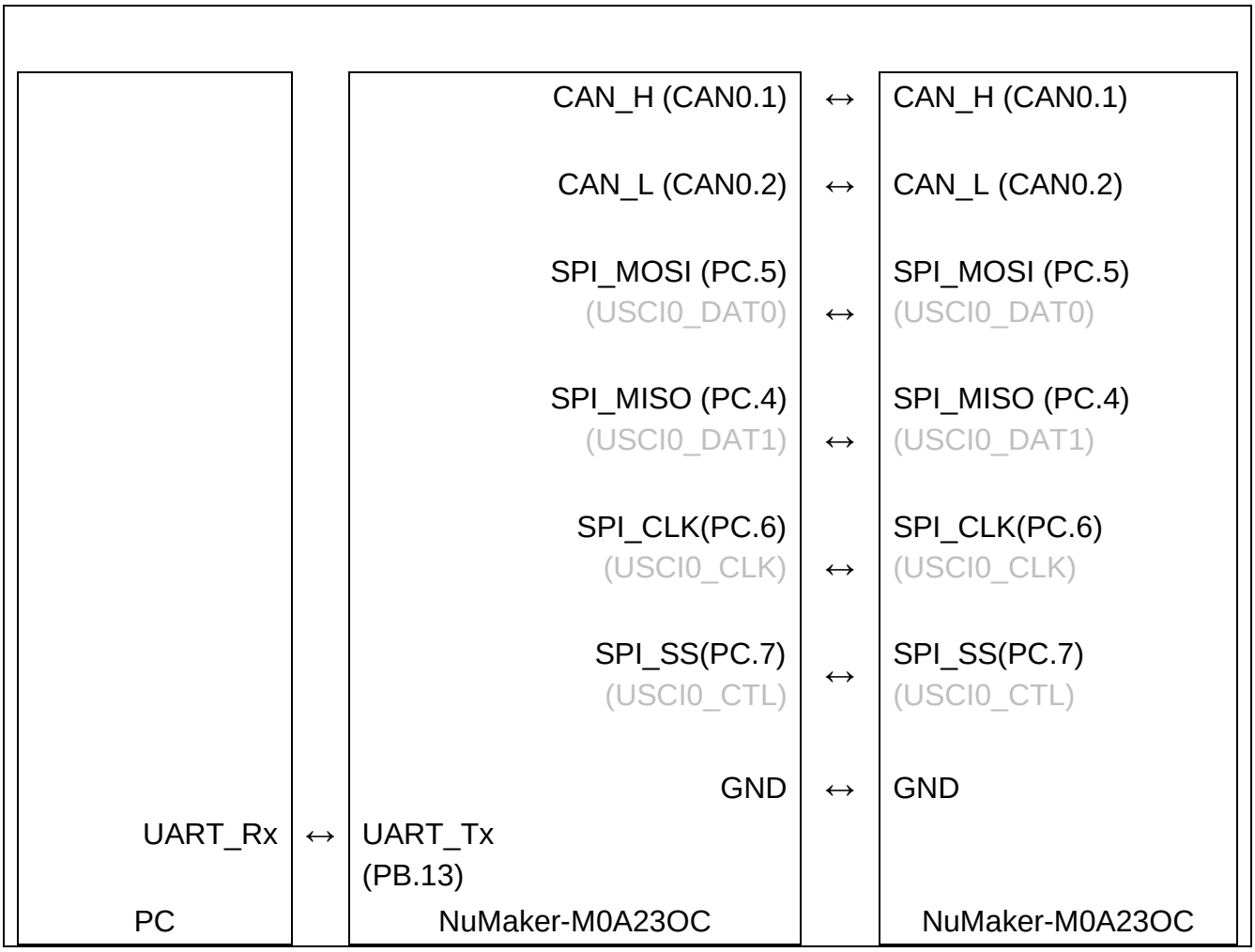


Figure 3-1 Pin Connect

4. Directory Information

The directory structure is shown below.

EC_M0A23_SPI_To_CAN_Bridging_Transmission_V1.00	
Library	Sample code header and source files
CMSIS	Cortex® Microcontroller Software Interface Standard (CMSIS) by Arm® Corp.
Device	CMSIS compliant device header file
StdDriver	All peripheral driver header and source files
SampleCode	
ExampleCode	
Project	Source file of example code

Figure 4-1 Directory Structure

5. Example Code Execution

1. Browse the sample code folder as described in the Directory Information section and double-click *M0A23_SPI_To_CAN_Bridging_Transmission.uvprojx*.
2. Enter Keil compile mode.
   - Build
   - Download
   - Start/Stop debug session
3. Enter debug mode.
   - Run

6. Revision History

Date	Revision	Description
2025.06.16	1.00	Initial version.

Important Notice

Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.

Please note that all data and specifications are subject to change without notice.
All the trademarks of products and companies mentioned in this datasheet belong to their respective owners.