Application Note

How to Program TXE81XX SPI I/O Expander Family

Tyler Townsend

ABSTRACT

This application note provides insight into programming the TXE81XX family of I/O expanders.

Table of Contents

Trademarks

1 Introduction

The TXE81XX device family is TI's first SPI to GPIO expander. This application note provides details on how to program these devices with an explanation of the 24-bit SPI word structure. This document also looks at the register map and helps to explain the purposes of some of the common registers within the I/O expander.

2 Setup and Configuration

TXE81XX SPI to GPIO expander family is controlled through the 4-wire SPI interface: MISO (master-in-slave-out), MOSI (master-out-slave-in), SCLK (clock), and CS (chip select) lines. To this day, the terminology has since been redefined to the following.

MISO → POCI (peripheral-out-controller-in)

MOSI → PICO (peripheral-in-controller-out)

SCLK is the same

CS is the same

The TXE81XX follows the terminology as follows.

SDI (serial-data-input) → MOSI / PICO

SDO (serial-data-output) → MISO / POCI

SCLK → Clock

CS → Chip Select

In this example, the M0 launchpad LP-MSPM0C1104 is used to program the TXE8124 with the following connections.

VCC = 3.3V → (pin 2)

PA11 → SCLK (pin 29)

PA16 → MISO/POCI/SDO (pin 1)

PA18 → MOSI/PICO/SDI (pin 30)

PA2 → /CS (pin 28)

GND → (pin 3)

The SPI settings are set for 500kHz SPI clock, CPOL (clock idle polarity) = 0 (LOW), CPHA (clock-phase) = 0 (rising-edge / leading edge).

Figure 2-1 is the complete block diagram used for the physical connections between the TXE8124 and the MSPM0.

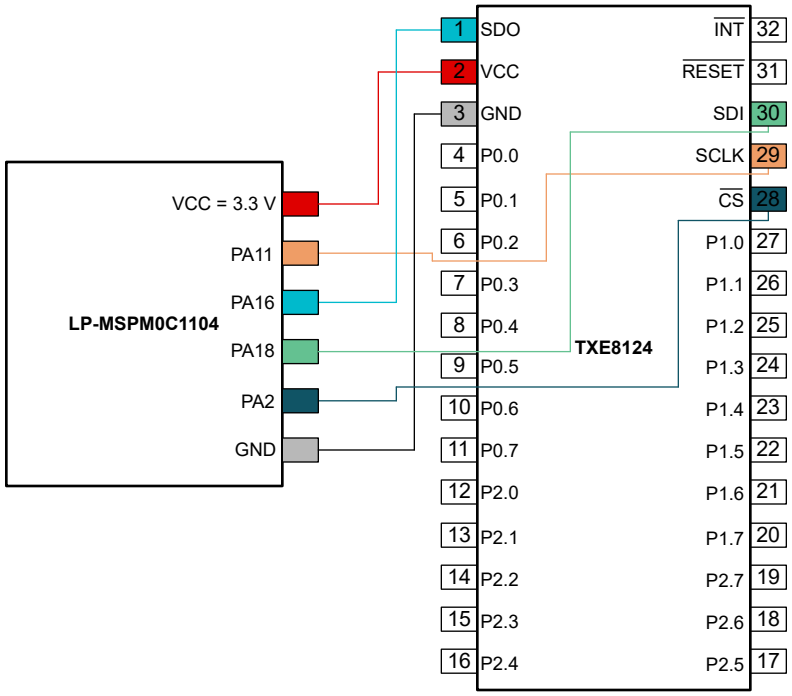

Figure 2-1. Connection Scheme for LP-MSPM0C1104 and TXE8124

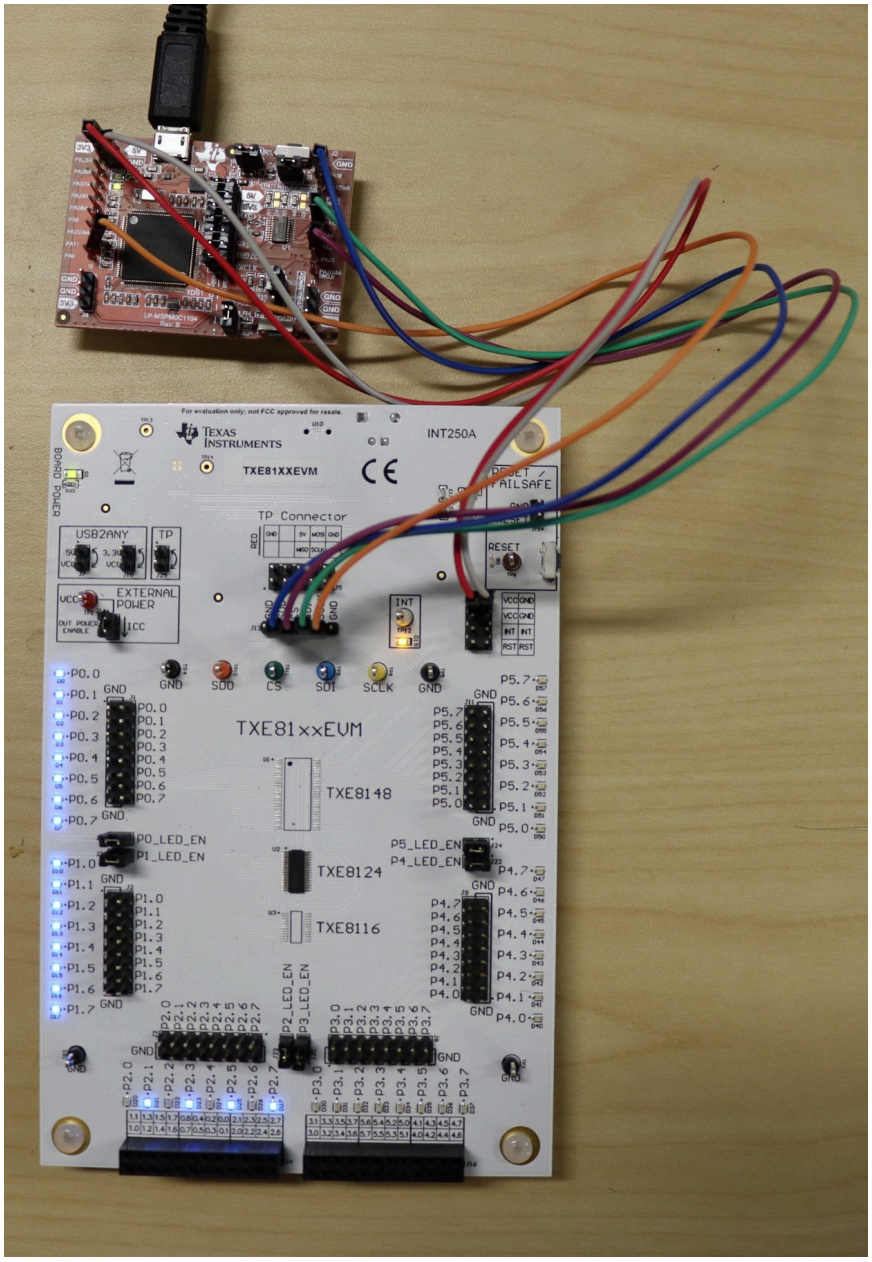

Figure 2-2. LP-MSPM0C1104 Connection to TXE81XXEVM Example

3 TXE81XX 24-Bit SPI Word Definition

TXE81XX uses the following 24-bit word structure to initiate SPI writes and reads to and from the device.

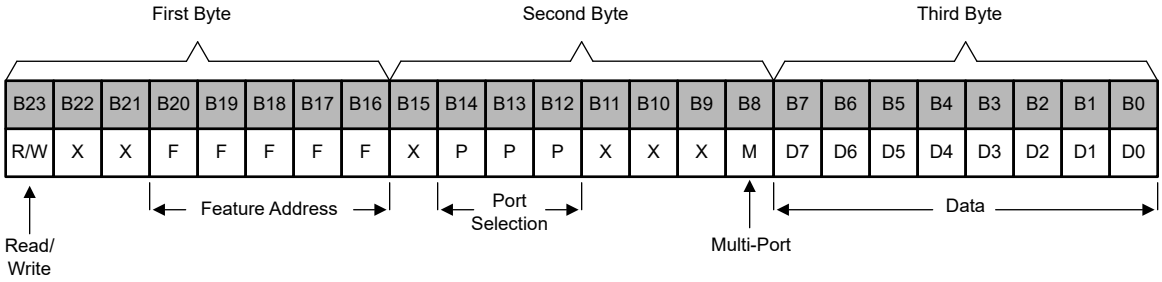

Figure 3-1. TXE81XX 24-Bit SPI Word

The SPI word is 24 bits long and requires data to be shifted in MSB first on SDI.

B23 → R/W = read or write bit (R = 1, W = 0)

B22 - B21 = (X) don’t care

B20 - B16 = feature address (5 bits)

B15 = (X) do not care

B14 - B12 = port selection (Port 0 = 000, Port 1 = 001, Port 2 = 010)

B11 - B9 = (X) don’t care

B8 = multi-port bit (multi-port enabled = 1, multi-port disabled = 0)

B7 - B0 = data byte

4 SPI Write Steps

A SPI write command involves writing data to a designated register while also reading the previous register data contents on SDO (full-duplex).

1. Drive /CS LOW. This enables the internal shift register
2. Shift 24 bits of data into the device MSB first on SDI. Data must be stable during the rising edge of the clock (SCLK).
3. The MSB bit (B23) must be a “0” to indicate this is a write operation.
4. 16 bits of status is sent out on SDO. The first 2 bits are 2’b11 (indicating this to be a status segment). The next 6 bits B21-B16 are bits D5 to D0 of the Fault Status Register. The last 8 bits B15-B8 are all 0’s.
5. The previous data in the register is read on SDO (B7-B0) while a data byte is written into the register on SDI (B7-B0).
6. After the last bit of data is transferred, drive SCLK low if there is no more data to be transferred.
7. De-assert /CS (drive this high) to end the write cycle

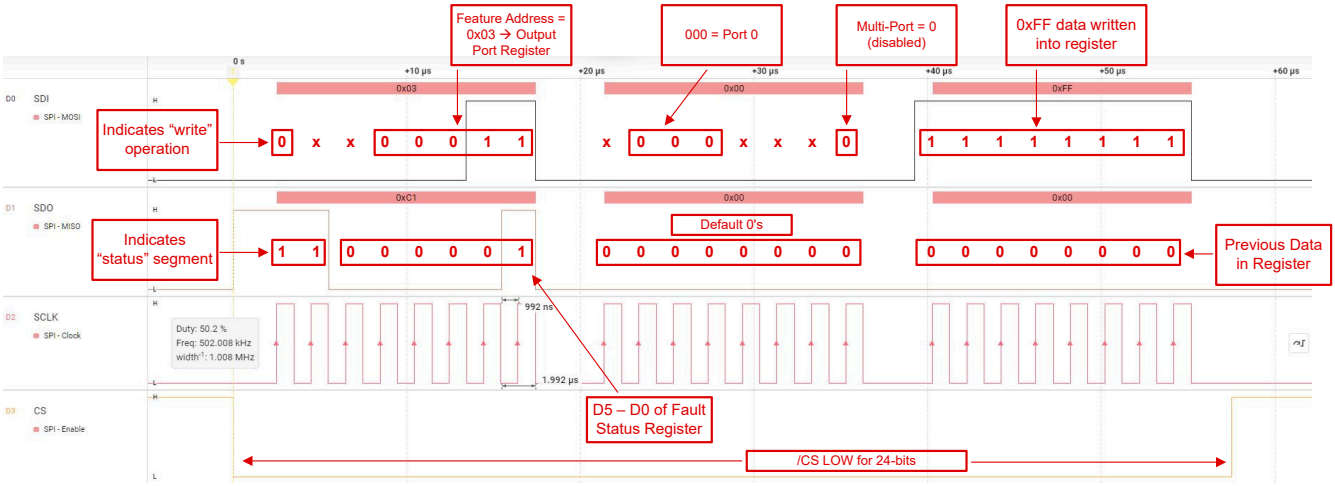

Figure 4-1. SPI Write Example Using Saleae Logic Analyzer

5 Coding Example

The current version of code composer studio can be downloaded from TI Developer Zone below the *Common Actions* section.

After downloading and opening the Code Composer Studio IDE, the example code can be found by browsing the software examples provided in the Resource Explorer.

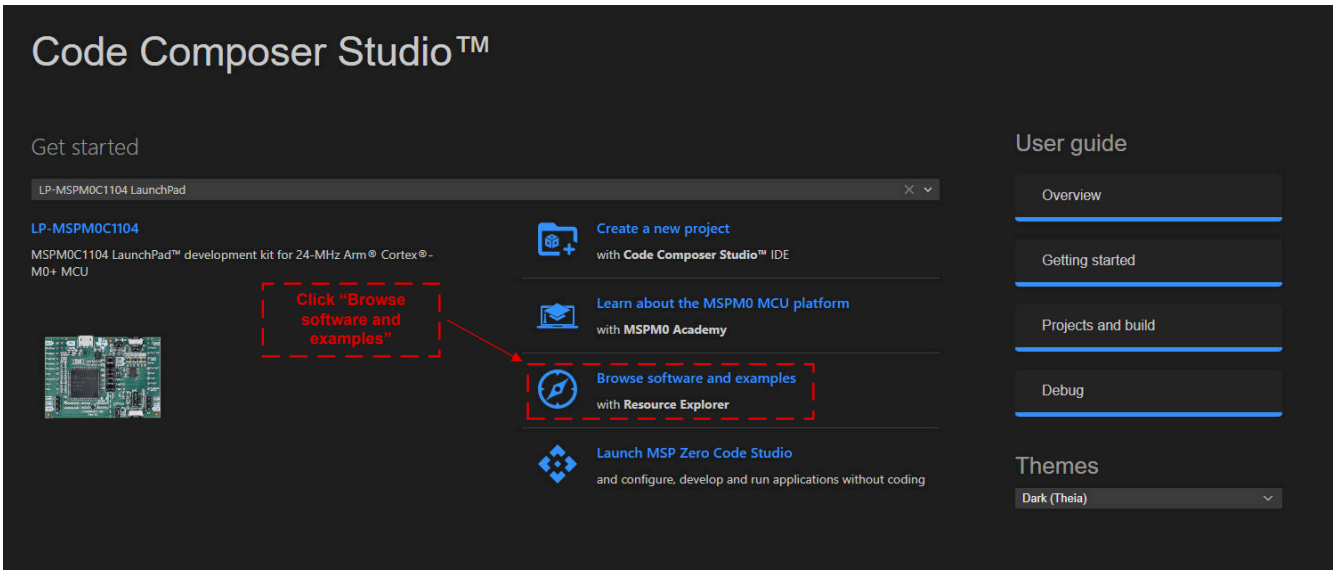

Figure 5-1. In Code Composer Studio IDE, Browse Software and Examples

Once the resource explorer is opened, follow the below directory to locate the sample code:

Arm^R-Based Microcontrollers → Embedded Software → MSPM0 SDK – 2.04.00.06 → Examples → Development Tools → LP-MSPM0C1104 LaunchPad → DriverLib → spi_controller_internal_loopback_poll → No RTOS → GCC Compiler → spi_controller_internal_loopback_poll

The sample project can be imported into the CCS IDE by clicking the 3 dots and selecting “Import to CCS IDE.”

Once imported, start by selecting the “spi_controller_internal_loopback_poll.syscfg” file.

The SPI settings in the example are to be modified through the SYSCONFIG GUI. Underneath SPI, select the following configurations to setup the code properly.

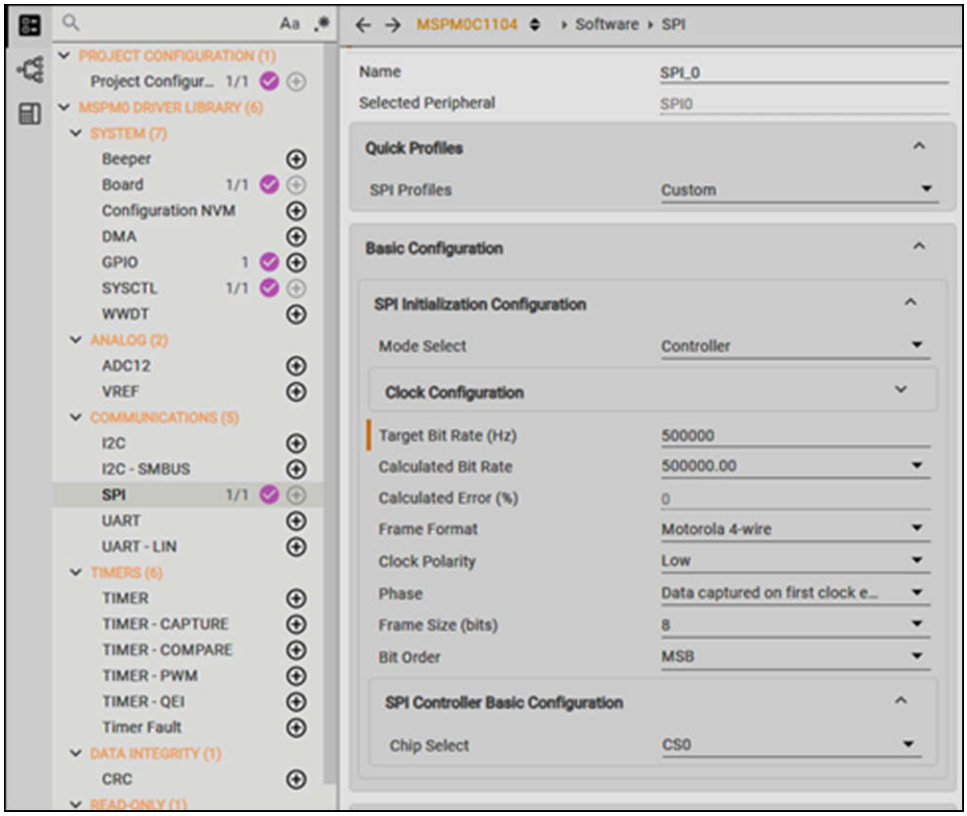

Figure 5-2. SYSCONFIG Setup

SPI_0 is the name of the SPI peripheral in use. A custom profile is created. The mode is selected to be *Controller* with a target bit rate of 500kHz. Frame formatting is Motorola 4-wire SPI. Clock polarity set to LOW. Clock phase – data captured on first clock edge which is rising edge. Frame size is set to 8 bits. Bit order is MSB first. While chip select is CS0 – PA11, this example uses the GPIO on PA2 to capture the 3-byte framing structure for the TXE8124 device. CS0 – PA11 is ignored for this example.

Select the following for the *Advanced Configuration* tab making sure the *Enable Internal Loopback* is unchecked.

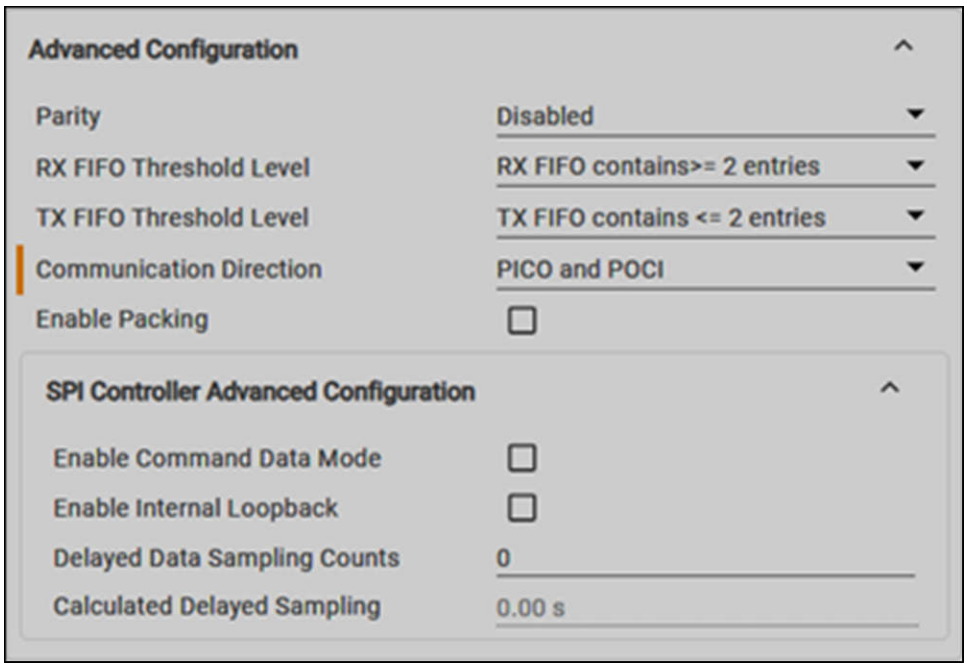

Figure 5-3. Below the Advanced Configuration Tab, the Enable Internal Loopback is Disabled

The SYSCONFIG GUI sets up the SPI configuration without having to write code within the M0 driver library. This sets up the SPI driver giving more time to focus on the example functions for the TXE81XX.

6 Sample Code

The following sample code can be copied over the spi_controller_internal_loopback_poll.c example code from the MSPM0 SDK. This .c file was modified to include SPI write and read functions that work with the 24-bit word scheme of the TXE81XX.

```
/*SPI Connections:
VCC = 3.3V
CPOL = 0
CPHA = 0
Clock Frequency = 500kHz
SCLK = PA11
MISO = PA16
MOSI = PA18
/CS = PA2
*/
#include "ti_msp_dl_config.h" //MSP driver library
//Define the TXE81xx Register Map from the TXE81xx d
#define scratch_reg (0x00)
#define device_ID (0x01)
#define input_reg (0x02)
#define output_reg (0x03)
#define config_reg (0x04)
#define pol_reg (0x05)
#define pp_od_sel (0x06)
#define od_conf (0x07)
#define pu_pd_en (0x08)
#define pu_pd_sel (0x09)
#define bus_holder (0x0A)
#define smart_int (0x0B)
#define int_mask (0x0C)
#define glitch_filter_en (0x0D)
#define int_flag_status (0x0E)
#define int_port_status (0x0F)
#define fail_safe_en_reg_1 (0x12)
#define fail_safe_en_reg_2 (0x13)
#define fail_safe_dir_config1 (0x14)
#define fail_safe_dir_config2 (0x15)
#define fail_safe_out1 (0x16)
#define fail_safe_out2 (0x17)
```

```
#define fail_safe_redun (0x18)
#define fail_safe_fault (0x19)
#define software_reset (0x1A)
//define each port
//TXE8116 - Port 0 / Port 1
//TXE8124 - Port 0 / Port 1 / Port 2
#define port0 (0x00)
#define port1 (0x01)
#define port2 (0x02)
#define DELAY_500khz (50) //delay cycles to tune /CS for SPI comms at 500kHz
//Function Definitions
uint8_t SPI_Transfer(uint8_t feat_addr, uint8_t port, uint8_t data_byte, bool multi_port);
uint8_t SPI_read (uint8_t feat_addr, uint8_t port);
int main(void)
{
volatile uint8_t read_val = 0x00;
SYSCFG_DL_init(); //initalize SPI driver

read_val = SPI_Transfer(config_reg, port0, 0xFF, false); //write to configuration register 0 - set all output
//write outputs every second on port 0
while(1){
read_val = SPI_Transfer(output_reg, port0, 0xFF, false); //0xFF - LED's off
read_val = SPI_read(output_reg, port0);
delay_cycles(24000000);

read_val = SPI_Transfer(output_reg, port0, 0xAA, false); //0xAA - alternate LED's
read_val = SPI_read(output_reg, port0);
delay_cycles(24000000);
read_val = SPI_Transfer(output_reg, port0, 0x55, false); //0x55 - alternate the other way
read_val = SPI_read(output_reg, port0);
delay_cycles(24000000);

read_val = SPI_Transfer(output_reg, port0, 0x00, false); //0x00 - LED's ON
read_val = SPI_read(output_reg, port0);
delay_cycles(24000000);
}
}
//The SPI_Transfer both writes and returns the previous byte set in the 8-bit register
uint8_t SPI_Transfer(uint8_t feat_addr, uint8_t port, uint8_t data_byte, bool multi_port){
```

```
uint8_t write_byte = feat_addr & 0x1F; //only keep B20 - B16
uint8_t port_byte = (port << 4) & 0x70; //only keep B14 - B12, first shift port number to upper nibble
uint8_t m_bit;
uint8_t status_byte1 = 0x00; //first 2 bits are 2'b11 (indicating status segment), next 6 bits are D5 to D0 bits of the fault flag register
uint8_t status_byte2 = 0x00; //8 bits are all 0
uint8_t reg_data_out = 0x00; //the current register data
if (multi_port == true){
m_bit = 0x01; // multi-port = 1 = enabled
} else {
m_bit = 0x00; // multi-port = 0 = disabled
}
DL_GPIO_clearPins(GPIO_LEDS_PORT, GPIO_LEDS_USER_LED_1_PIN |
GPIO_LEDS_USER_TEST_PIN); //set /CS LOW
DL_SPI_transmitData8(SPI_0_INST, write_byte); //Transmit B23 - B16
DL_SPI_transmitData8(SPI_0_INST, port_byte + m_bit); //Transmit B15 - B8
DL_SPI_transmitData8(SPI_0_INST, data_byte); //Transmit B7 - B0
status_byte1 = DL_SPI_receiveDataBlocking8 (SPI_0_INST); //reading data on SDO 1st byte
status_byte2 = DL_SPI_receiveDataBlocking8 (SPI_0_INST); //reading data on SDO 2nd byte
reg_data_out = DL_SPI_receiveDataBlocking8 (SPI_0_INST); //reading data on SDO 3rd byte - this is the register data returned (previously set byte in register)
delay_cycles(DELAY_500khz); //delay to time chip select
DL_GPIO_setPins(GPIO_LEDS_PORT, GPIO_LEDS_USER_LED_1_PIN |
GPIO_LEDS_USER_TEST_PIN); //set /CS HIGH
return reg_data_out; //return the previous register byte
}
//create a function to directly read a register and the status bits for the fault flag register
uint8_t SPI_read (uint8_t feat_addr, uint8_t port){
uint8_t read_byte = (feat_addr & 0x1F) + 0x80; //only keep B20 - B16 and make B23 = 1 for a "READ"
uint8_t port_byte = (port << 4) & 0x70; //only keep B14-B12, first shift port number to upper nibble
uint8_t status_byte1 = 0x00; //first 2 bits are 2'b11 (indicating status segment), next 6 bits are D5 to D0 bits of the fault flag register
uint8_t status_byte2 = 0x00; //8 bits are all 0
uint8_t reg_data_out = 0x00; //the current register data

DL_GPIO_clearPins(GPIO_LEDS_PORT, GPIO_LEDS_USER_LED_1_PIN |
GPIO_LEDS_USER_TEST_PIN); //set CS LOW
DL_SPI_transmitData8(SPI_0_INST, read_byte); //send read command byte
DL_SPI_transmitData8(SPI_0_INST, port_byte); //specify the port selection
DL_SPI_transmitData8(SPI_0_INST, 0x00); //0x00 dummy data to shift out 8 bits of register data on MISO
```

```
status_byte1 = DL_SPI_receiveDataBlocking8 (SPI_0_INST); //first 8 bits of status
status_byte2 = DL_SPI_receiveDataBlocking8 (SPI_0_INST); //second 8 bits of status
reg_data_out = DL_SPI_receiveDataBlocking8 (SPI_0_INST); //register data byte
delay_cycles(DELAY_500khz); //delay to time chip select
DL_GPIO_setPins(GPIO_LEDS_PORT, GPIO_LEDS_USER_LED_1_PIN |
GPIO_LEDS_USER_TEST_PIN); //set CS HIGH
return reg_data_out; //return register byte
}
```

7 Summary

The TXE family of SPI to GPIO expanders uses a 24-bit word format. Writing to the TXE device is a full-duplex operation where data can be written and read from a register on SDI and SDO respectively. A direct read of the TXE results in data output from a register on SDO. The example code provided is a non-optimized design for simple read or write commands to the TXE, and shows the general format on how to operate the IC.

8 References

- Texas Instruments, *MSPM0 SDK*, resource explorer.

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