AN13059

Enable I^{2}C Slave Bus Communication Bridge to SPI Master based on LPC802 MCU

Rev. 0 — 11/2020 Application Note

Contents

1 Introduction & Requirements

This application is to develop a firmware on LPC802 MCU, enabling an I^{2}C slave bus communication bridge to SPI master and GPIO control function as the SC18IS602B chip.

The SC18IS602B is designed to serve as an interface between a standard I^{2}C-bus of a microcontroller and an SPI bus. It allows the microcontroller to directly communicate with SPI devices through its I^{2}C-bus. The SC18IS602B operates as an I^{2}C-bus slave-transmitter or slave-receiver and an SPI master. The SC18IS602B controls all the SPI bus-specific sequences, protocol, and timing. The SC18IS602B has its own internal oscillator, and it supports four SPI chip select outputs that may be configured as GPIO when not used.

- I^{2}C-bus slave interface operating up to 400 kHz
- SPI master operating up to 1.8 Mbit/s
- 200-byte data buffer
- Up to four slave select outputs
- Up to four programmable I/O pins, shared with SPI slave select output pins
- Operating supply voltage: 2.4 V to 3.6 V
- Low power mode
- Internal oscillator option, without external oscillators
- Active LOW interrupt output
- ESD protection exceeds 2000 V HBM per JESD22-A114, 200 V MM per JESD22-A115, and 1000 V CDM per JESD22-C101
- Latch-up testing is done to JEDEC Standard JESD78 that exceeds 100 mA
- Very small 16-pin TSSOP

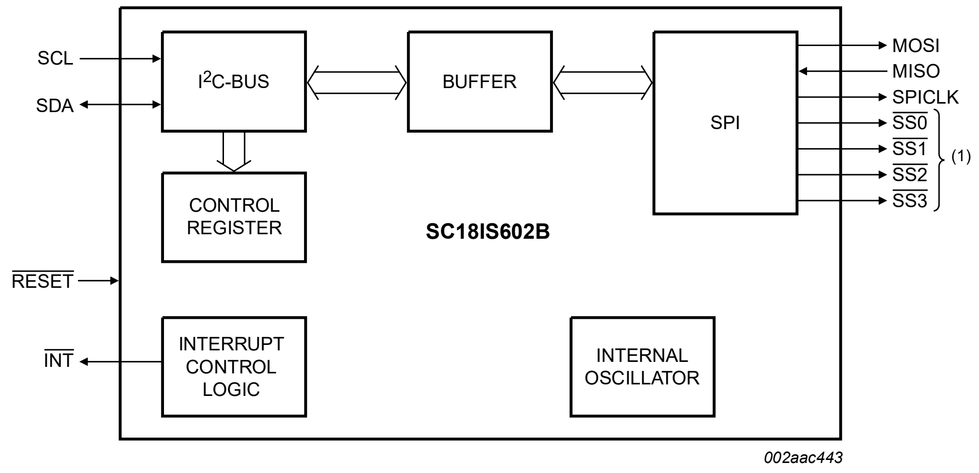


Figure 1. SC18IS602B block diagram

In the development based on the LPC802, I used two LPCXpresso802 boards, as described in OM40000. The on-board part was LPC802M001JDH20, which had an Arm® Cortex®-M0+ core at 15 MHz, 16 KB FLASH and 2 KB SRAM, with TSSOP16 package. Even this application coould not make the LPC802 working as SC18IS602B pin to pin the same, as the power pin and the reset pin were not in the same position. The most function would be compatible. Table 1 summarizes the commands supported by SC18IS602B and the difference between the LPC802 and SC18IS602B.

Table 1. SC18IS602B commands and LPC802's implementation

SC18IS602B command	LPC802 implementation
SPI write - Function ID 01h to 0Fh	Fully support
SPI read	Fully support
Configure SPI interface - Function ID F0h	Fully support
Clear interrupt - Function ID F1h	Fully support
Idle mode - Function ID F2h	Fully support
GPIO write - Function ID F4h	Fully support
GPIO read - Function ID F5h	Fully support
GPIO enable - Function ID F6h	Fully support
GPIO Configuration - Function ID F7h	Not supporting the **quasi-bidirectional** mode

For the detailed description about the format of frame for each command, see the **Functional description** chapter in *I 2C-bus to SPI bridge* (document SC18IS602B).

This application note describes the implementation of the functions in the source code. According to the guide here, engineers can make new development based on other packages, to support more pins, of LPC802 or other MCU platform for the similar function.

2 Hardware setup

Two LPCXpresso802 boards were used in this development: one as I^{2}C master and the other as I^{2}C slave. The two boards were connected with I^{2}C bus. The I^{2}C master board converted the user command, through the UART terminal, to I^{2}C data/command frame, acting like a communication bridge from UART to I^{2}C master. The I^{2}C slave board ran the functions of I^{2}C slave to SPI master like the SC18IS602B, accepting the I^{2}C command and outputting through the SPI bus and GPIO pins.

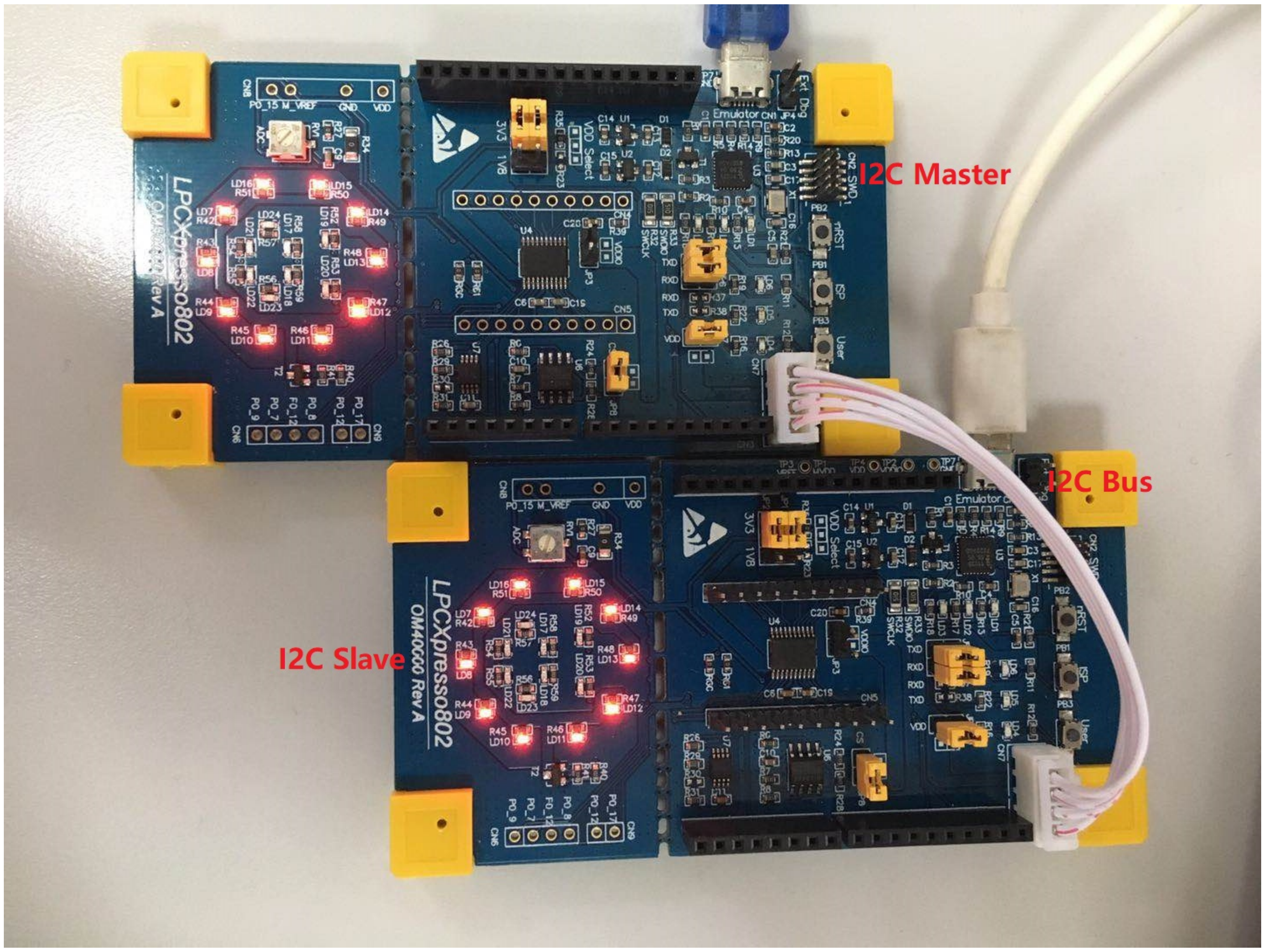


Figure 2. Two LPC802 board connected with I^2 bus

NOTE

I chose the LPC802 board as both the master board and the slave board because it could make the development easier. During the development, I could reuse the driver for both master board and slave board, and also I could use either board to debug. Only when running the final demo with two board working separately, I downloaded the different firmware to each board according to their own role.

3 Software enablement

3.1 Enable the interrupt-based SPI master transfer

This part of function was used to transfer the data through the SPI bus. I used the interrupt mode here because the SPI transfer should be launched by the execution of command when received from I^{2}C bus. However, the I^{2}C slave must work in interrupt mode, so that it can monitor the commands from the master at any time. The I^{2}C slave ISR would recognize command frame byte-by-byte , and deal with the byte stream after receiving each byte. I would not like to stay inside the I^{2}C slave's ISR too much time when executing the SPI transaction and ignoring any further I^{2}C data. So I need that the SPI driver can deal with the transaction automatically outside the I^{2}C slave ISR. It means the command dispatcher running inside I^{2}C slave ISR would just start the SPI transaction when necessary and return to wait for the next I^{2}C command, while the SPI ISR would handle the data's transaction simultaneously.

The `spi_comm.h/.c` file contains the implementation of the driver for SPI master:

```
void spi_init_master(SPI_Type *base);
void spi_conf_master(SPI_Type *base, spi_speed_t speed, spi_cpolcpha_t cpolcpha, spi_bitorder_t 
bitorder);
void spi_master_start_xfer(SPI_Type *base, spi_master_xfer_handler_t *xfer_handler);
void spi_master_isr_hook(SPI_Type *base, spi_master_xfer_handler_t *xfer_handler);
```

In the application code, the `spi_init_master()` function should be called one time before any operation to the SPI bus. "`spi_conf_master()` is optional, as a default working mode is already set up in the `spi_init_master()`. There would be an I^{2}C command to modify the SPI's configuration, so a special API is provided here to support that feature.

Then user needs to allocate a structure variable of the `xfer_handler` type for the SPI transfer.

```
spi_master_xfer_handler_t  cmd_spi_master_xfer_handler_struct;
```

This structure variable would be filled with the data, in the buffer, to be transferred along with `spi_master_start_xfer()` in I^{2}C slave ISR:

```
cmd_spi_master_xfer_handler_struct.cs_mask = (rx_cmd & cmd_gpio_pinmask_for_spi_cs);
gpio_cs_pin_write_0(cmd_spi_master_xfer_handler_struct.cs_mask);
/* start the spi master transfer. */
cmd_spi_master_xfer_handler_struct.buff_len = xfer_len;
cmd_spi_master_xfer_handler_struct.rx_buff = cmd_i2c_slave_xfer_buff;
cmd_spi_master_xfer_handler_struct.tx_buff = cmd_i2c_slave_xfer_buff;
cmd_spi_master_xfer_handler_struct.xfer_done_callback = cmd_spi_master_xfer_done_callback;
spi_master_start_xfer(CMD_SPI_MASTER_INSTANCE, &cmd_spi_master_xfer_handler_struct);
```

The SPI ISR would handle all the process for data transfer defined in `spi_master_isr_hook()`:

```
void SPI0_IRQHandler(void)
{
    spi_master_isr_hook(CMD_SPI_MASTER_INSTANCE, &cmd_spi_master_xfer_handler_struct);
}
```

When a buffer transfer is done, the callback function would be called. This callback function is registered into `cmd_spi_master_xfer_handler_struct`, and usually used to de-assert the CS pins when used.

3.2 Enable the interrupt-based I^{2}C slave transfer

The driver of I^{2}C slave has to be working in the interrupt mode, as it should monitor the I^{2}C bus at any time. The driver source code helps to process the data transfer from the bus, while leaving the frame recognition and execution work to the code in the application level.

The *i2c_comm.h/.c* file contains the implementation of the driver for I^{2}C slave:

```
typedef void (*i2c_func1_t)(void *param1);
/* slave. interrupt method.*/
void i2c_init_slave(I2C_Type * base, uint8_t slave_addr, uint8_t *xfer_buff, uint8_t xfer_len);
void i2c_slave_isr_hook(I2C_Type *base);
void i2c_slave_install_xfer_done_callback(I2C_Type *base, i2c_func1_t func);
```

In the application code, the `i2c_init_slave()` function should be called first, to enable the I^{2}C slave function after all other peripherals, such as, SPI and GPIO, are ready, as the I^{2}C command would be executed to handle these peripherals. The `slave_addr` parameter is the local I^{2}C slave address, matched with the I^{2}C frame. `xfer_buff` and `xfer_len` are used to map to an internal buffer memory. This memory block should be pre-allocated in the application but its handler is kept inside the I^{2}C driver. It is filled with the received data in the interrupt ISR.

The `i2c_slave_isr_hook()` function should be called at the entry of I^{2}C slave ISR. As the I^{2}C driver is interrupt-based, all the operations about the I^{2}C slave would be done inside the ISR. Once there is an event, received a data or any condition signal, such as, START, STOP and ACK/NACK, asserted during the transfer, This function deals with the frame, which is assembled with a stream of bytes, reading or writing the memory for the internal data buffer. It finally calls an internal callback function once it detects a STOP function when told that a frame transfer is done.

In the internal callback function, the driver leaves the operation to a complete transfer event to apply, so that users can do something according the most recent transfer. The callback function can be installed with the API function, `i2c_slave_install_xfer_done_callback()`, following the type of `i2c_func1_t` with one parameter. Actually, the parameter of the callback function is used to pass the pointer of frame information, so that users can access them. The parameter is actually pointing to the structure defined in the following type.

```
typedef struct
{
    uint32_t flags;
    uint32_t xfer_len;
    uint8_t *xfer_data;
    uint8_t rx_cmd;
  } i2c_slave_xfer_done_callback_param_t;
```

3.3 Enable the command dispatcher in callback for I^{2}C slave transfer

A callback function would be called once a frame transfer is done, marked with the STOP condition on the I^{2}C bus. The pointer to the memory buffer, keeping the data of most recent xfer frame, is also passed into this function as a parameter.

In this application, I recognized the commands and dispatched the execution inside this callback function. The code used was in the `switch ... case ...` pattern.

```
void cmd_i2c_slave_xfer_done_callback(void *param1)
{
    i2c_slave_xfer_done_callback_param_t *xfer_done_param_ptr = (i2c_slave_xfer_done_callback_param_t
*)param1;
    uint32_t flags = xfer_done_param_ptr->flags;
    uint8_t rx_cmd = xfer_done_param_ptr->rx_cmd;
    uint8_t *xfer_data = xfer_done_param_ptr->xfer_data;
    uint32_t xfer_len = xfer_done_param_ptr->xfer_len;

    /* receive. */
    if (flags & I2C_SLAVE_XFER_DONE_FLAG_RECEIVE_DATA)
    {
       /* the 1st item from rx is function cmd. */
       switch ( rx_cmd )
       {
          case CMD_I2C_SLAVE_CMD_CONF_SPI:
#if DEBUG_ENABLE_LOG
                printf("CMD_I2C_SLAVE_CMD_CONF_SPI: 0x%-2X.\r\n",xfer_data[0]);
#endif /* DEBUG_ENABLE_LOG */

                uint8_t bitorder = (uint8_t)((xfer_data[0] >> 5u) & 0x1);
                uint8_t cpolcpha = (uint8_t)((xfer_data[0] >> 2u) & 0x3);
                uint8_t speed = (uint8_t)(xfer_data[0] & 0x3);
                spi_conf_master(CMD_SPI_MASTER_INSTANCE,
                               (spi_speed_t)speed,
                               (spi_cpolcpha_t)cpolcpha,
                               (spi_bitorder_t)bitorder );
                break;
```

```
            case CMD_I2C_SLAVE_CMD_CLEAR_INT:
#if DEBUG_ENABLE_LOG
                printf("CMD_I2C_SLAVE_CMD_CLEAR_INT\r\n");
#endif /* DEBUG_ENABLE_LOG */
                gpio_int_pin_clear();
                break;
            /* ...... other commands.*/

            default:
                if (rx_cmd <= CMD_I2C_SLAVE_CMD_WRITE_BUFF)
                {
#if DEBUG_ENABLE_LOG
                    printf("CMD_I2C_SLAVE_CMD_WRITE_BUFF: 0x%-2X, %d\r\n", rx_cmd, xfer_len);
#endif /* DEBUG_ENABLE_LOG */
                    cmd_spi_master_xfer_handler_struct.cs_mask = (rx_cmd &
cmd_gpio_pinmask_for_spi_cs);
                    gpio_cs_pin_write_0(cmd_spi_master_xfer_handler_struct.cs_mask);

                    cmd_spi_master_xfer_handler_struct.buff_len = xfer_len;
                    cmd_spi_master_xfer_handler_struct.rx_buff = cmd_i2c_slave_xfer_buff;
                    cmd_spi_master_xfer_handler_struct.tx_buff = cmd_i2c_slave_xfer_buff;
                    cmd_spi_master_xfer_handler_struct.xfer_done_callback =
cmd_spi_master_xfer_done_callback;
                    spi_master_start_xfer(CMD_SPI_MASTER_INSTANCE,
&cmd_spi_master_xfer_handler_struct);
                 }
                 break;
         }
      }
      /* ...... */
}
```

This callback function was installed in the I^{2}C slave driver in the application-level code:

```
void cmd_i2c_slave_init(void)
{
    i2c_init_slave(CMD_I2C_SLAVE_INSTANCE, CMD_I2C_SLAVE_ADDRESS, cmd_i2c_slave_xfer_buff,
CMD_I2C_SLAVE_XFER_BUFF_LEN);
    i2c_slave_install_xfer_done_callback(CMD_I2C_SLAVE_INSTANCE, cmd_i2c_slave_xfer_done_callback);
}
```

3.4 Enable the I^{2}C Master driver on the test machine

To verify the working condition on the slave machine, a test machine running the I^{2}C master is also necessary.

In this project, I used the I^{2}C master driver in the polling mode for a simpler use case. The *i2c_comm.h/.c* file contained the implementation of the driver for I^{2}C master in the polling mode:

```
/* master. polling method.*/
void i2c_init_master(I2C_Type * base, i2c_speed_t speed);
void i2c_master_read(I2C_Type * base, uint8_t dev_addr, uint8_t *buff, uint8_t len);
void i2c_master_write(I2C_Type * base, uint8_t dev_addr, uint8_t *buff, uint8_t len);
```

NOTE

Here, the APIs are dealing with the operations to the bus, not to the device. It means, when calling the `i2c_master_read()` function, it directly performs the read operation from the bus, not the case that writing the register address and reading from the previous pointer. To avoid the operation, you need to call the `i2c_master_write()` first, with the register address as the first item in the transferring buffer, and then call the `i2c_master_read()` to get the contents for the register.

3.5 Enable a command shell on the test machine

A *cmd_shell* component is used to enable a shell based on the UART.

In this project, I interaced the test machine through the console. The source codes of *cmdshell* were all in the *cmdshell.h/.c* file. The porting work was included in the *cmd_shell_adapter.c* file, to remap the IO channel to UART:

```
#include "cmd_shell.h"
#include <stdio.h>

void CmdPutChar(char c)
{
    /* Put data to bus. */
    putchar(c);
}

char CmdGetChar(void)
{
     char ch;

     /* Fetch data from bus. */
     ch = getchar();

     return ch;
}

const CMD_HandlerCallback_T CmdHandlerCallbackStruct =
{
      .SER_PutCharFunc = CmdPutChar,
      .SER_GetCharFunc = CmdGetChar,
      ">"
};
```

Then I created the commands according to requirements in the *main.c* file:

```
extern const CMD_HandlerCallback_T CmdHandlerCallbackStruct;

/* Cmd Table. */
CMD_TableItem_T CmdI2CMasterTestCmdsTable[] =
{
    {"help",   1, cmd_show_help},
    {"spi_write",  8, cmd_i2c_master_spi_write},
    {NULL}
};
int32_t cmd_show_help(int32_t argc, char *argv[])
{
     uint32_t index = 0u;
     printf("# cmd_show_help()\r\n\r\n");
     while ( CmdI2CMasterTestCmdsTable[index].CmdName != NULL )
     {
         printf(" - %s\r\n", CmdI2CMasterTestCmdsTable[index].CmdName);
         index++;
```

```
      }
      printf("\r\n");
      return 0;
}

int32_t cmd_i2c_master_spi_write(int32_t argc, char *argv[])
{
     for (uint32_t i = 0u; i < argc-1; i++)
    {
         cmd_i2c_master_xfer_buff[i] = atoi(argv[i+1]);
         printf("0x%-2X, ", cmd_i2c_master_xfer_buff[i]);
     }
     printf("\r\n");

         i2c_master_write(CMD_I2C_MASTER_INSTANCE, CMD_I2C_SLAVE_ADDRESS, cmd_i2c_master_xfer_buff, 
argc-1u);
         printf("i2c_master_write() done.\r\n");

         return 0;
}
```

Finally, I called in the `main()` function:

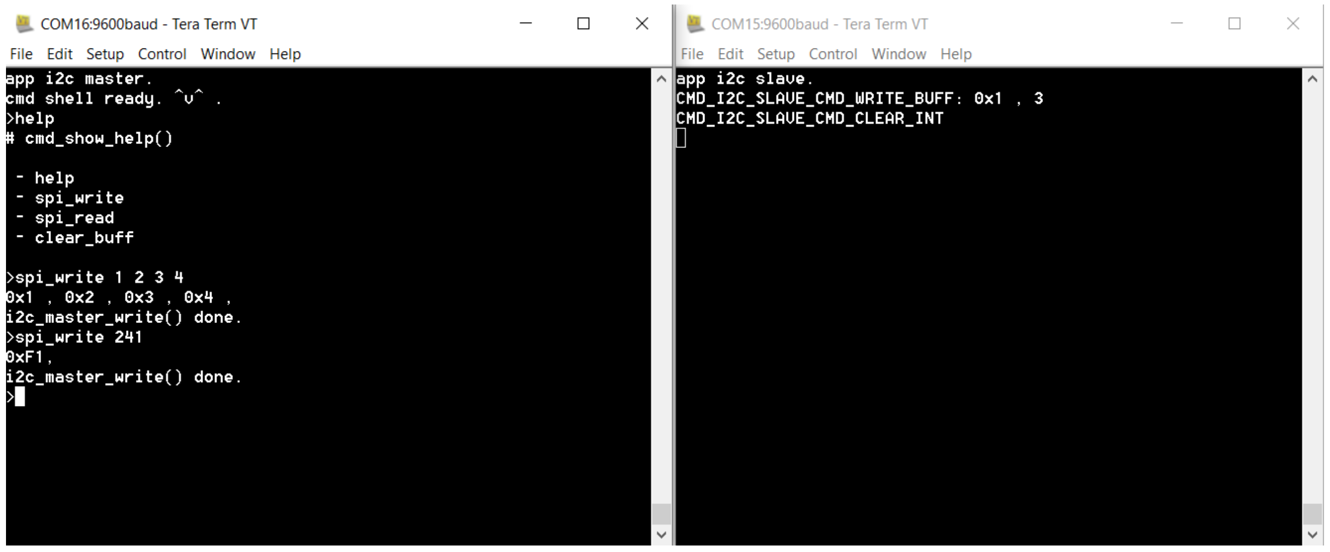
```
    CMD_InstallHandler(&CmdHandlerCallbackStruct);

    while (1)
    {
        CMD_LoopShell(CmdI2CMasterTestCmdsTable);
     }
```

4 Run the demo

When the demo is running, a shell is working based on the UART terminal of master' board. The following three commands are in the list:

- `help`: To show all the available commands.
- `spi_write`: To write the data through the I^{2}C bus and keep them into slave's buffer. The application running on the slave board recognizes the commands and activates, sending data or just updating the settings.
- `spi_read`: To read the SPI data buffer from the slave board.

Figure 3 shows an example.


COM16:9600baud - Tera Term VT
File Edit Setup Control Window Help

```
app i2c master.
cmd shell ready. ^v^ .
>help
# cmd_show_help()

 - help
 - spi_write
 - spi_read
 - clear_buff

>spi_write 1 2 3 4
0x1 , 0x2 , 0x3 , 0x4 ,
i2c_master_write() done.
>spi_write 241
0xF1,
i2c_master_write() done.
>
```

COM15:9600baud - Tera Term VT
File Edit Setup Control Window Help

```
app i2c slave.
CMD_I2C_SLAVE_CMD_WRITE_BUFF: 0x1 , 3
CMD_I2C_SLAVE_CMD_CLEAR_INT
```

Figure 3. Run the commands through the console

Date of release: 11/2020

Document identifier: AN13059