

Subsystem Design

IO Expander With SPI, I2C, and UART



1 Description

Note

TI is transitioning to use more inclusive terminology. Some language contained in this document is possibly different than previously-used terms for certain technology areas.

This subsystem demonstrates how to use MSPM0 to achieve IO expander function through the communication command from serial peripheral interface (SPI), I2C, or universal asynchronous receiver-transmitter (UART) by the host. This expansion is helpful when the number of GPIOs on the host is inadequate. In addition to supporting control of the GPIO output, the subsystem can also read back the GPIO status through SPI, I2C, or UART. The figure below shows the basic architecture of the subsystem and the modules which are used in this case.

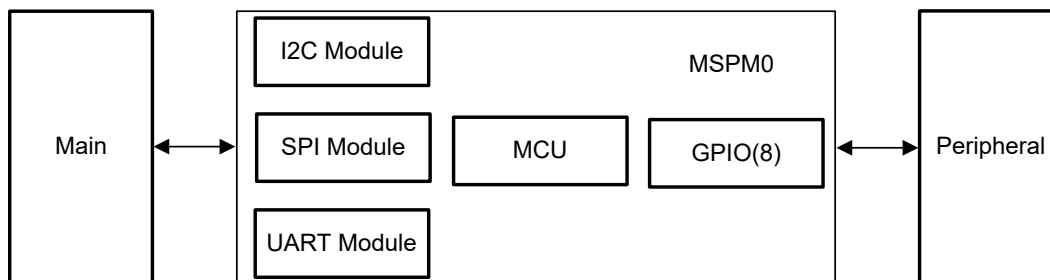


Figure 1-1. Subsystem Functional Block Diagram

Table 1-2 has links to the example code. In this demonstration, the IO control number is limited to eight. However, users can do further IO expansion by referring to this demonstration.

2 Required Peripherals

Table 2-1 shows the required peripherals and function modules in this application.

Table 2-1. Required Peripherals

Subblock Functionality	Peripheral Use	Notes
Serial Peripheral Interface	(1 ×) SPI	Called <i>SPI_0_INST</i> in code
I2C interface	(1 ×) I2C	Called <i>I2C0_INST</i> in code
UART interface	(1 ×) UART	Called <i>UART_0_INST</i> in code
GPIO	(8 ×) GPIO	Called <i>GPIO_GRP_0</i> in code

3 Hardware Setup

The following hardware elements are required to evaluate the IO expander based on MSPM0:

- MSPM0 LaunchPad™ development kit shown in Required Devices
- A computer with Microsoft® Windows® 7 or later, and .NET Framework 4.5. (or real device as primary)
- USB2ANY and compatible GUI

In this subsystem, the customer can flexibly choose different communication interfaces, including I2C, SPI, or UART. This can greatly increase the flexibility of the customer system design. [Figure 3-1](#) shows the hardware connection in this design, using LP-MSPM0C1104 as an example.

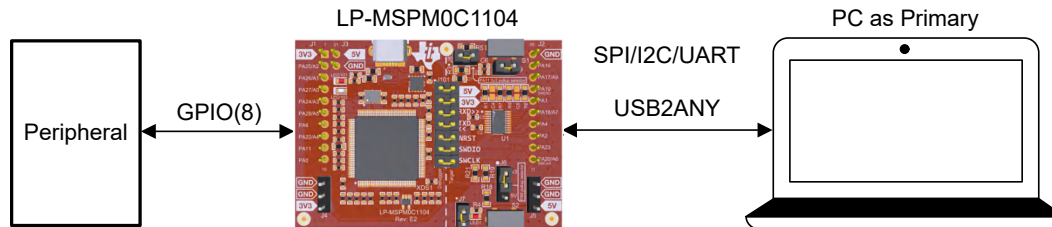


Figure 3-1. Hardware Connection

[Table 3-1](#) shows the pin configuration, you can also change the configuration following your requirements. The SPI communication is configured as three-wire mode to save GPIO resources. The pin configuration is the same for MSPM0L1306 and MSPM0C1104.

Table 3-1. Pin Configuration

Module	Function	Pin Configuration	Comment
I2C interface	SDA	PA0	Address: 0x48, I2C clock freq: 400kHz
	SDL	PA1	
Serial Peripheral Interface	POSI	PA25	SPI clock freq: 500kHz
	PISO	PA26	
	CLOCK	PA17	
UART interface	RX	PA18	Baud rate: 9600bps
	TX	PA23	
GPIO	GPIO	BIT0:PA2, BIT1:PA27, BIT2:PA17, BIT3:PA24, BIT4:PA4, BIT5:PA6, BIT6:PA16, BIT7:PA22	

4 Software Introduction

Figure 4-1 shows the software project, developed in CCS. The project mainly consists of three parts. Other files are the default files for the MSPM0 project.

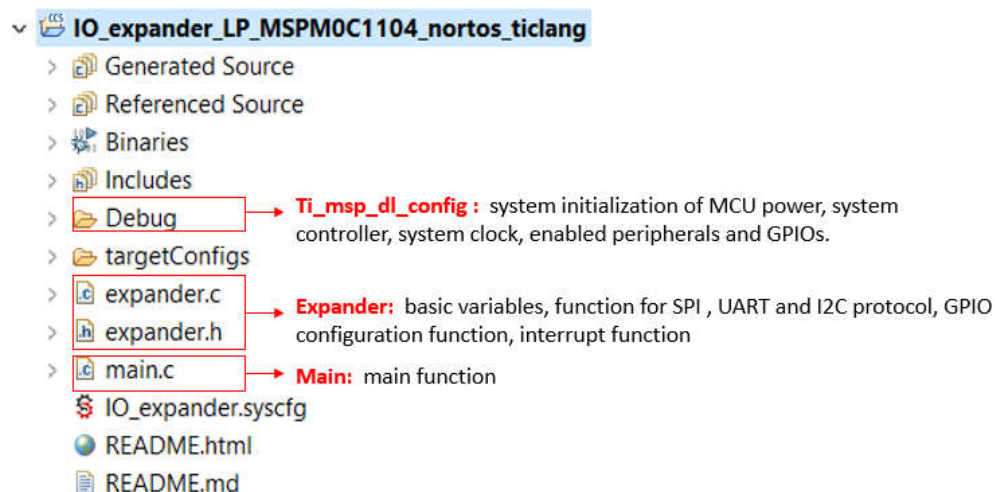


Figure 4-1. Software Project View

The `ti_msp_dl_config` part is generated by [SysConfig](#) (graphic code generation tool), the MSPM0 initialization is for system initialization of MCU power, system controller, system clock, enabled peripherals and GPIOs.

The `expander` part declares the basic variables, GPIO configuration function, and interrupt function, and `expander` also contains some basic functions for SPI, I2C and UART protocol.

The `main` part includes the highest system function code, after system initialization, the MCU waits for commands from the primary and executes the corresponding GPIO operation.

4.1 Code Introduction

Figure 4-2 shows the main function code in this design. The main function initializes the system configuration and then enters a loop to handle IO control. The code supports three functions for IO control: gpioDirectionSet, gpioOutputCtl, and gpioStateRead. See also [Protocol Introduction](#).

```
#include <expander.h>
#include "ti_msp_dl_config.h"

tDataType ioExpander;

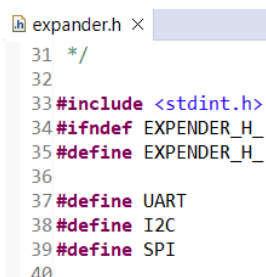
int main(void)
{
    SYSCFG_DL_init();
    systemInit();

    while (1)
    {
        if(ioExpander.pktTyp != eNoPkt)
        {
            switch(ioExpander.pktTyp)
            {
                case eGpioDir:
                    gpioDirectionSet();
                    break;
                case eGpioOtp:
                    gpioOutputCtl();
                    break;
                case eGpioInp:
                    gpioStateRead();
                    break;
                default:
                    break;
            }

            ioExpander.pktTyp = eNoPkt;
        }
    }
}
```

Figure 4-2. Main Function

UART, I2C, and SPI are enabled by default. Uncomment the definition in the expander.h file for real implementation, as [Figure 4-3](#) shows.



```
expander.h ×
31 */
32
33 #include <stdint.h>
34 #ifndef EXPENDER_H_
35 #define EXPENDER_H_
36
37 #define UART
38 #define I2C
39 #define SPI
40
```

Figure 4-3. Communication Enable and Disable

All the communication command reception is done in the relative interrupts. Using UART, command transmission is done in the gpioStateRead() function. Command transmission is done in the relative interrupts for SPI and I2C.

4.2 Protocol Introduction

This section presents the support command for this demonstration. The `gpioDirectionSet` function is used to enable or disable GPIO output. Because MSPM0 can enable GPIO input and output at the same time, GPIO input is always enabled. Then every bit in the direction byte is used to enable or disable GPIO output. A *1* means enabling GPIO output and input, and a *0* means only enabling GPIO input. See [Table 3-1](#) for the relationship between the controlled bit and GPIO.

Table 4-1. GPIO Output Enable Command

Type	Header	Command	Direction	Checksum
<code>gpioDirectionSet</code>	0x5A	0x01	1 Byte (1: OUT; 0: IN)	1 Byte

The `gpioOutputCtl` function is used to control GPIO output. Then every bit in the output control byte is used to set GPIO output high or output low. A *1* means outputting high, and a *0* means outputting low. Remember this function only works after the primary send `gpioDirectionSet` function. See [Table 3-1](#) for the relationship between the controlled bit and GPIO.

Table 4-2. GPIO Output Control Command

Type	Header	Command	Output Control	Checksum
<code>gpioOutputCtl</code>	0x5A	0x02	1 Byte(1: High; 0: Low)	1 Byte

The `gpioStateRead` function is used to read the GPIO status. The primary needs to send this command to the secondary, and then the secondary sends the GPIO state back to the primary. Then every bit in the pin state byte sent by the secondary is used to show the GPIO state. A *1* means the GPIO state is high, and a *0* means the GPIO state is low. This command can also be used to check whether the GPIO control is as expected, after the primary sends the `gpioOutputCtl` command. See [Table 3-1](#) for the relationship between the controlled bit and GPIO.

Table 4-3. GPIO State Read Command

Type	Header	Command	Pin State	Checksum
<code>gpioStateRead</code>	0x5A	0x03	1 Byte(1: High; 0: Low)	1 Byte

[Figure 4-4](#) shows the command send conditions on the primary and secondary, see also [Section 6](#).

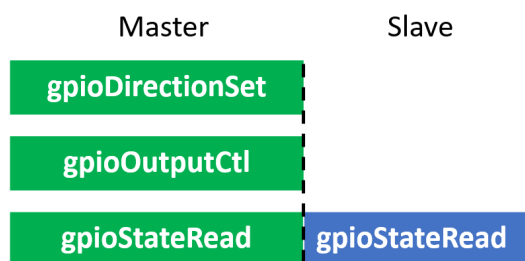


Figure 4-4. Command Send Conditions

5 Design Steps

Complete the following steps to evaluate the design:

1. Prepare the necessary resource:

Prepare the relevant hardware resources and software code as previously discussed in this document. Use the real host system to replace the PC with USB2ANY GUI. Pay attention to the I2C setup for MSPM0C1104, because PA1 is reused as the RESET pin. For software disable the reset function in SysConfig, for hardware remove the J9 connector to avoid the pulldown capacitor influence and on I2C communication. Use additional pullup resistors.

2. Choose communication interface:

Choose the appropriate communication interface (UART, SPI, or I2C) by uncommenting the unwanted communication functions macro in expander.h, shown in [Figure 4-3](#). Now initialize the communication interface and set the communication parameters such as baud rate, data bits, and stop bits, in the IO_expander.syscfg (if using UART).

Make sure that the transmitPacket and receivePacket functions are correctly called to send and receive data packets during communication, and verify the checksum to make sure the communication is correct.

3. Start to use:

Write functions to execute the gpioDirectionSet, gpioOutputCtl, and gpioStateRead commands by referring to [Section 4.2](#). Send the relevant commands to the expansion module from the host and then you can verify if the GPIO direction setting, output value, and input value match the expectations.

6 Results

Protocol usage based on I2C: [Figure 6-1](#) shows the usage of four different commands, including GPIO Output Enable Command, GPIO Output Control Command, GPIO State Read Request Command and GPIO State Read Response Command. Notice that the read back GPIO state is the same as the settled GPIO state in the GPIO Output Control Command.

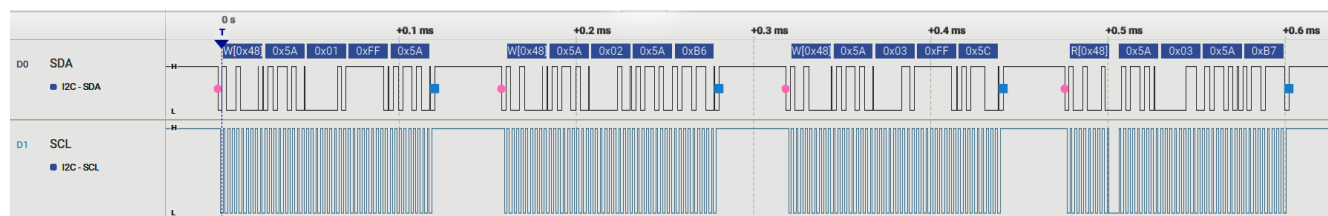


Figure 6-1. I2C Communication 1

[Figure 6-2](#) shows the usage of three different commands, including GPIO Output Enable Command, GPIO State Read Request Command, and GPIO State Read Response Command. In this test, all the GPIOs are settled to be the input state.

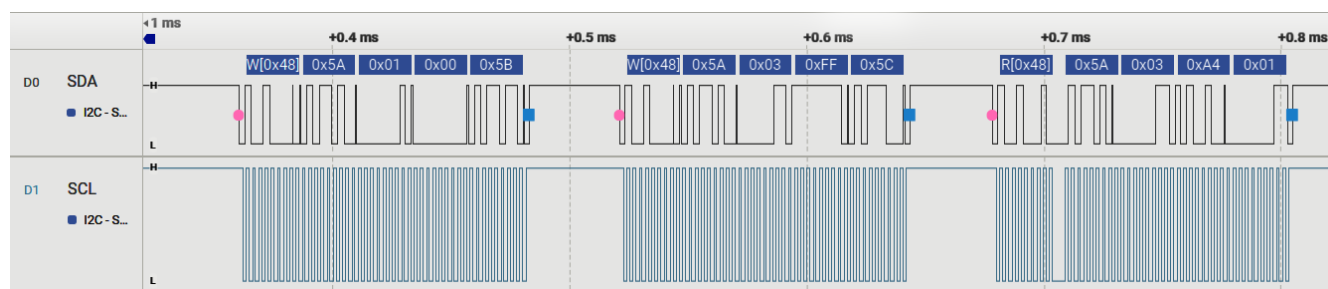


Figure 6-2. I2C Communication 2

Protocol usage based on UART: Figure 6-1 shows the usage of four different commands, including GPIO Output Enable Command, GPIO Output Control Command, GPIO State Read Request Command and GPIO State Read Response Command.

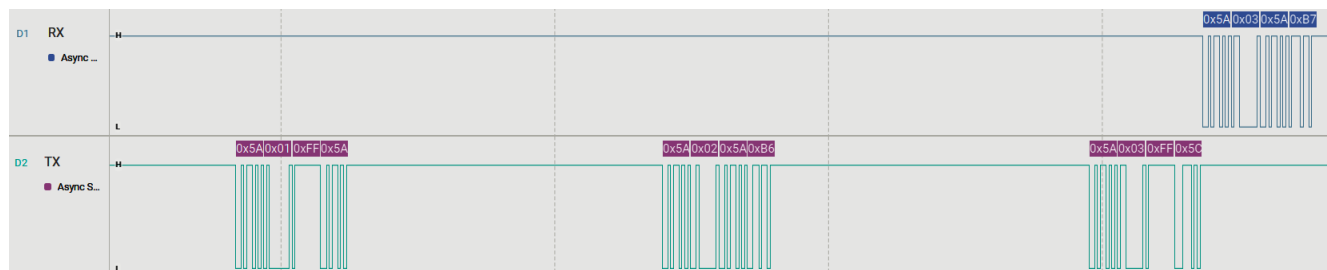


Figure 6-3. UART Communication

Protocol usage based on SPI: Figure 6-1 shows the usage of four different commands, including GPIO Output Enable Command, GPIO Output Control Command, GPIO State Read Request Command and GPIO State Read Response Command.

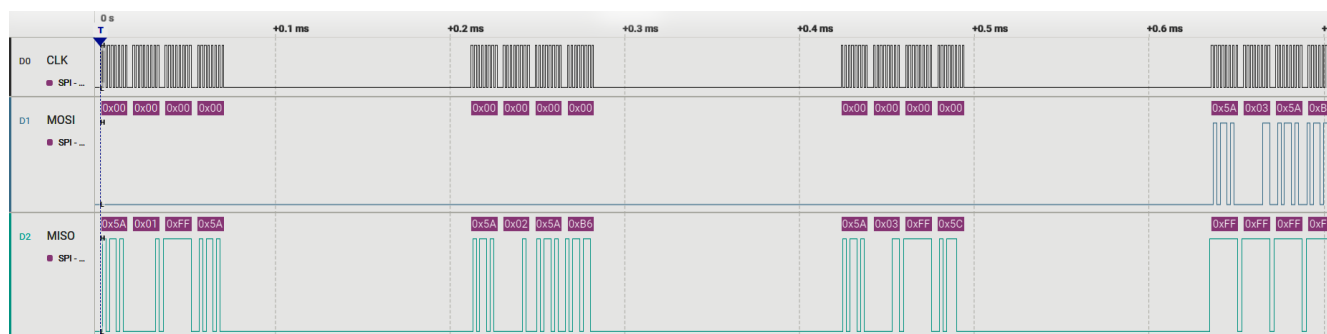


Figure 6-4. SPI Communication

7 Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Revision * (October 2024) to Revision A (August 2025)	Page
• Removed Compatible Devices section.....	1

8 Trademarks

LaunchPad™ is a trademark of Texas Instruments.

Microsoft® and Windows® are registered trademarks of Microsoft Corporation.

All trademarks are the property of their respective owners.

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](https://www.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
Copyright © 2025, Texas Instruments Incorporated