Centrale Nantes

MAC : 7^{th} Lab's Report
Driver For The I/O Expander MCP23S17 (SPI Interface)

1^{st} year Embedded Systems Engineering

By

EL KHAYDER Zakaria

SAOUTI Rayan

Professor

BRIDAY Mikael

February 12, 2023

Session

2023-2024

Made with LaTeX

Contents

List of Figures

List of Listings

1 SPI Communication

We have access to `spi.h` that defines these functions:

- **setupSPI**: To initiate and configure the STM32 dedicated SPI chip.
- **beginTransaction**: Start an SPI transaction by lowering the CS pin (CS = 0).
- **endTransaction**: To end the SPI transaction by waiting for all buffer data to be transmitted and then raising back the CS pin (CS = 1).
- **transfer8**: Transfer 1 byte of data using SPI and return the response.

Thanks to these already-implemented functionalities, interfacing with the MCP23S17 will be more about sending the correct data and less about configuring and handling the STM32 signals ourselves.

1.1 Read/Write

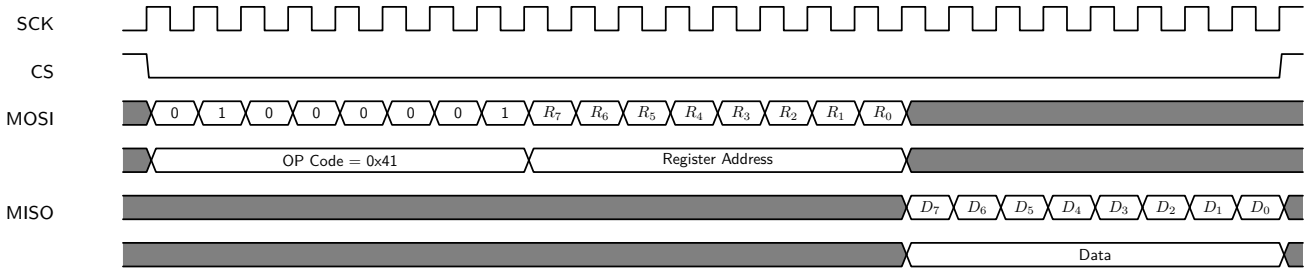

Figure 1: SPI Transaction: Read

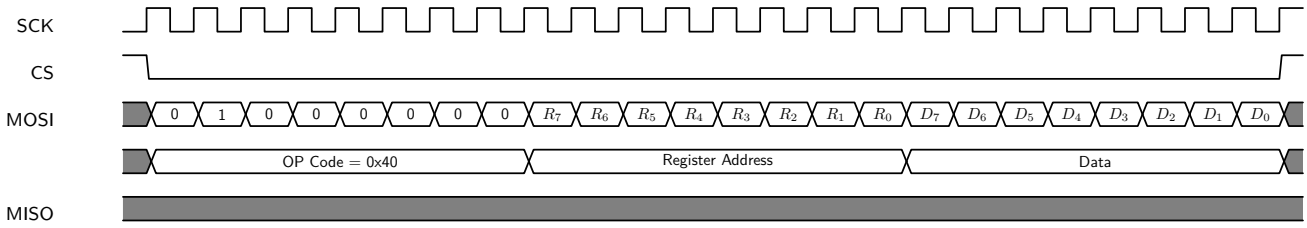

Figure 2: SPI Transaction: Write

1.2 Register Manipulation

1.2.1 Registers Enum

To avoid having to type register addresses by hand, which is both unreadable and error-prone, we chose to declare a `reg` enum that holds the addresses of the 21 different internal registers of the MCP23S17 chip.

```cpp
enum class reg : uint8_t {
    IODIRA   = 0x00, IODIRB   = 0x01,
    IPOLA    = 0x02, IPOLB    = 0x03,
    GPINTENA = 0x04, GPINTENB = 0x05,
    DEFVALA  = 0x06, DEFVALB  = 0x07,
    INTCONA  = 0x08, INTCONB  = 0x09,
    IOCON1   = 0x0A, IOCON2   = 0x0B,
    GPPUA    = 0x0C, GPPUB    = 0x0D,
    INTFA    = 0x0E, INTFB    = 0x0F,
    INTCAPA  = 0x10, INTCAPB  = 0x11,
    IOA      = 0x12, IOB      = 0x13,
    OLATA    = 0x14, OLATB    = 0x15,
};
```

Listing 1: Registers enum

1.2.2 Write to register

We can build our custom register write function using the previously mentioned functions from `spi.h`.

We should cast `reg` enum to `uint8_t` since it is the only type accepted by `transfer8`.

```cpp
void MCP23S17::_writeRegister(reg r, uint8_t val) {
    beginTransaction();

    transfer8(0x40); // 0b01000000
    transfer8((uint8_t)r);
    transfer8(val);

    endTransaction();
}
```

Listing 2: Write to Register

1.2.3 Read from register

For reading, besides sending both the OP code and the register address, we should send an extra byte so that we can read the data sent by MCP23S17.

```
uint8_t MCP23S17::_readRegister(reg r) {
    beginTransaction();

    transfer8(0x41); // 0b01000001
    transfer8((uint8_t)r);
    uint8_t val = transfer8(0x00); // Get output

    endTransaction();

    return val;
}
```

Listing 3: Read from register

1.3 Bit Manipulation

Flipping a single bit (set/reset) can be done in three steps:

- Read the existing value
- Modify the corresponding bits
- Write the value back into the register

```
void MCP23S17::_setBit(reg r, uint8_t idx) {
    uint8_t state = _readRegister(r);
    _writeRegister(r, state | (0x01 << idx));
}
```

Listing 4: Bit Set

```
void MCP23S17::_clearBit(reg r, uint8_t idx) {
    uint8_t state = _readRegister(r);
    _writeRegister(r, state & ~(0x01 << idx));
}
```

Listing 5: Bit Clear

2 MCP23S17 API

The functions defined in the previous section were not meant to be used by the user directly as they are private `MCP23S17` class methods, rather, they will be used internally by the rest of the methods that we will be declaring, which are meant to be used by the user.

We aim to provide functions similar to the `Arduino` API:

- **pinMode**: Output, Input, Input Pullup
- **digitalRead**
- **digitalWrite**
- **attachInterrupt**: Rising, Falling, Both

2.1 Easier register access

To simplify our API consumers' lives further, we can declare a `Port` enum (A or B), and hide our previous `reg` enum for internal use only.

```
enum class Port : uint8_t {
    A = 0x00,
    B = 0x01,
};
```

Listing 6: Port enum

We will still use the `reg` enum, although only half of it, to calculate the corresponding Register address for the specific port.

This works wonders with the current register mapping we use (BANK 0) since the registers are only offset with a single address for each port.

```
MCP23S17::reg MCP23S17::_getRelativeRegister(reg base, Port p) {
    return static_cast<reg>(static_cast<uint8_t>(base) + static_cast<uint8_t>(p));
}
```

Listing 7: Relative Register

2.2 Digital write

We can write an output on our MCP23S17 by setting the value to the corresponding `GPIO` register

```
void MCP23S17::digitalWrite(Port p, uint8_t idx, bool value) {
    reg portRegister = _getRelativeRegister(reg::IOA, p);

    if (value) _setBit(portRegister, idx); // 1
    else _clearBit(portRegister, idx); // 0
}
```

Listing 8: digitalWrite

2.3 Read port

```
uint8_t MCP23S17::readBits(Port p) {
    return _readRegister(_getRelativeRegister(reg::IOA, p));
}
```

Listing 9: readBits

2.4 Digital read

```
bool MCP23S17::digitalRead(Port p, uint8_t idx) {
    return readBits(p) & (0x01 << idx);
}
```

Listing 10: digitalRead

2.5 Pin mode

According to the MCP23S17 documentation, we should use the corresponding port `IODIR` register. By setting the value of the corresponding bit to `0` the pin would be an `OUTPUT`, otherwise, if it is set to `1`, the pin would be an `INPUT`.

To activate the pullup resistor for a specific pin, we have to set the corresponding bit on the corresponding port `GPPU` register to `1`.

```
enum class PinMode { Output, Input, Input_Pullup };
```

Listing 11: PinMode Enum

```cpp
void MCP23S17::pinMode(Port p, uint8_t idx, PinMode mode) {
    reg portIoRegister = _getRelativeRegister(reg::IODIRA, p);
    reg portPullupRegister = _getRelativeRegister(reg::GPPUA, p);

    switch (mode)
    {
    case PinMode::Output:
        _clearBit(portIoRegister, idx);
        break;

    case PinMode::Input_Pullup:
        _setBit(portPullupRegister, idx);
        [[fallthrough]]; // Allowing fallthrough to set the pin as input

    case PinMode::Input:
        _setBit(portIoRegister, idx);
        break;
    }
}
```

Listing 12: pinMode

2.6 Interrupts

Interrupts are probably the most complicated part of our MCP23S17 API.

We need to declare a custom `Interrupt struct` where we can save the config (type and handler) for each interrupt.

```cpp
typedef void (*InterruptCallback)(void);

struct Interrupt {
    bool enabled = false;
    InterruptType type;
    InterruptCallback callback = nullptr;
}
```

Listing 13: Interrupt struct

2.6.1 Interrupt config

```cpp
enum class InterruptType { Rising, Falling, Both };
```

Listing 14: Interrupt type

We reserve an array of size 16 and type `Interrupt` where we can save our config for the different pins (8 pins per port = 16)

```
struct Interrupt _interrupts[16];
```

We will also need a helper function that returns an index for each port and pin (hashing if you like)

```
uint8_t MCP23S17::_getInterruptHandlerIndex(Port p, uint8_t idx) {
    return (uint8_t)p * 8 + idx;
}
```

Listing 15: _getInterruptHandlerIndex

Now with that implemented, we can declare our `attachInterrupt` function that will handle configuring the right register on our MCP23S17 and save the required info in the correct `_interrupts` case.

```
void MCP23S17::attachInterrupt(Port p, uint8_t idx, InterruptType type, InterruptCallback callback) {
    _clearBit(_getRelativeRegister(reg::INTCONA, p), idx); // Edge interrupt (any)
    _setBit(_getRelativeRegister(reg::GPINTENA, p), idx);  // Enable Interrupt

    _interrupts[_getInterruptHandlerIndex(p, idx)].type = type;
    _interrupts[_getInterruptHandlerIndex(p, idx)].enabled = true;
    _interrupts[_getInterruptHandlerIndex(p, idx)].callback = callback;
}
```

Listing 16: _getInterruptHandlerIndex

As you might notice, we cannot configure the interrupt on a specific edge directly on the MCP23S17, so we will have to handle the edge detection ourselves in our handler. Which makes a nice segue to our next chapter :)

2.6.2 Interrupt handler

When there is an interrupt in our MCP23S17, it will trigger an external interrupt in our STM32 uc, the external interrupt pin connected to the MCP23S17 is PA9.

We need to first create an init function where we will set the SPI driver, zero out the `_interrupts` array from whatever garbage existed before, and configure the interrupt on the PA9.

```cpp
void MCP23S17::Init() {
    setupSPI(); // Setup SPI

    for (uint16_t i = 0; i < sizeof(_interrupts) / sizeof(*_interrupts); i++) {
        _interrupts[i].enabled = false;
        _interrupts[i].callback = nullptr;
    }

    // Configure EXTI (PA9)
    ::pinMode(GPIOA, 9, INPUT_PULLUP);
    ::attachInterrupt(GPIOA, 9, FALLING);
}
```

Listing 17: MCP23S17::Init

In our interrupt handler, we will have to get the `INTF` status register state and the `INTCAP` state for both ports. With this information, we can determine if an interrupt occurred on a pin, and what type.

```cpp
void MCP23S17::onInterrupt(void) {
    uint16_t interruptFlags = _readRegister(MCP23S17::reg::INTFB) << 8
                            | _readRegister(MCP23S17::reg::INTFA);
    uint16_t state = _readRegister(reg::INTCAPB) << 8 | _readRegister(reg::INTCAPA);

    for (int i = 0; i < 16; i++) {
        uint16_t bit_Msk = 1 << i;
        auto intr = &_interrupts[i];

        // If there is no interrupt flag, or the interrupt is not enabled
        if (!(interruptFlags & bit_Msk) || !intr->enabled)
            continue;

        // Any edge
        if (intr->type == InterruptType::Both)
            intr->callback();
        // Falling Edge
        else if (intr->type == InterruptType::Falling && !(state & bit_Msk))
            intr->callback();
        // Rising Edge
        else if (intr->type == InterruptType::Rising && (state & bit_Msk))
            intr->callback();
    }
}
```

Listing 18: MCP23S17::onInterrupt

This function should be called on the actual PA9 interrupt handler `EXTI9_5_IRQHandler`:

```cpp
extern "C" void EXTI9_5_IRQHandler(void) {
    ioExt.onInterrupt();
    EXTI->PR |= EXTI_PR_PR9; // acknowledge
}
```

Listing 19: EXTI9_5_IRQHandler

With this implemented, we have officially implemented all the required and essential functions to facilitate the usage of the MCP23S17 external io expander.

Our result class definition looks like this:

```
class MCP23S17 {
public:
    typedef void (*InterruptCallback)(void);
    enum class Port : uint8_t { ... };
    enum class PinMode : uint8_t { ... };
    enum class InterruptType { ... };

private:
    enum class reg : uint8_t { ... };

public:
    void Init();

    void pinMode(Port p, uint8_t idx, PinMode mode);
    void digitalWrite(Port p, uint8_t idx, bool value);
    bool digitalRead(Port p, uint8_t idx);
    uint8_t readBits(Port p);

    void onInterrupt(void);

    void attachInterrupt(Port p, uint8_t idx, InterruptType type, InterruptCallback callback);

private:
    void _writeRegister(reg r, uint8_t val);
    uint8_t _readRegister(reg r);
    void _setBit(reg r, uint8_t idx);
    void _clearBit(reg r, uint8_t idx);

    reg _getRelativeRegister(reg base, Port p);
    uint8_t _getInterruptHandlerIndex(Port p, uint8_t idx);

private:
    struct Interrupt {
        bool enabled = false;
        InterruptType type;
        InterruptCallback callback = nullptr;
    } _interrupts[16]; // 16 PINS (2 ports, 8 pins each)
};

extern MCP23S17 ioExt;
```

Listing 20: EXTI9_5_IRQHandler

3 Sample Application

Given the simplicity of our application thanks to our beautiful MCP23S17 API, we will not delve into the details of every nook and cranny of it.

```cpp
enum class Mode { Chaser, BlinkingAll, BlinkingOdd, BlinkingEven };
volatile Mode mode = Mode::Chaser;

void SetModeChaser() { mode = Mode::Chaser; }
void SetModeBlinkingAll() { mode = Mode::BlinkingAll; }
void SetModeBlinkingOdd() { mode = Mode::BlinkingOdd; }
void SetModeBlinkingEven() { mode = Mode::BlinkingEven; }

void ModeChaser() {
    static int i = 0;
    i++;
    if (i > 7)  i = 0;
    else if (i < 0) i = 7;

    for (int j = 0; j < 8; j++) ioExt.digitalWrite(MCP23S17::Port::A, j, i == j);

    for (volatile int x = 0; x < 500000; x++);
}

void ModeBlinking() {
    static bool ledsAreOn = false;
    ledsAreOn = !ledsAreOn;
    uint8_t val;

    switch (mode) {
        case Mode::BlinkingAll:  val = 0xFF; break;
        case Mode::BlinkingEven: val = 0xAA; break;
        case Mode::BlinkingOdd:  val = 0x55; break;
        default: break;
    }

    for (int i = 0; i < 8; i++) {
        if (!ledsAreOn) ioExt.digitalWrite(MCP23S17::Port::A, i, 0);
        else ioExt.digitalWrite(MCP23S17::Port::A, i, val & 1 << i);
    }

    for (volatile int x = 0; x < 1000000; x++);
}

void setup() {
    ioExt.Init();

    for (int i = 0; i < 8; i++) {
        ioExt.pinMode(MCP23S17::Port::A, i, MCP23S17::PinMode::Output);
        ioExt.pinMode(MCP23S17::Port::B, i, MCP23S17::PinMode::Input_Pullup);
    }

    ioExt.attachInterrupt(MCP23S17::Port::B, 4, MCP23S17::InterruptType::Rising, SetModeChaser);
    ioExt.attachInterrupt(MCP23S17::Port::B, 5, MCP23S17::InterruptType::Rising, SetModeBlinkingAll);
    ioExt.attachInterrupt(MCP23S17::Port::B, 6, MCP23S17::InterruptType::Rising, SetModeBlinkingOdd);
    ioExt.attachInterrupt(MCP23S17::Port::B, 7, MCP23S17::InterruptType::Rising, SetModeBlinkingEven);
}

int main() {
    setup();
    while (1) {
        switch (mode) {
            case Mode::Chaser:   ModeChaser();   break;
            default:             ModeBlinking(); break;
        }
    }
}
```

Listing 21: main.c

Resources

The code files, and this report's source code, are available on this GitHub repository: elkhayder/sec1-tp-mac