Lab 7
Driver For The I/O Expander
`MCP23S17` (SPI Interface)

M. Briday

January 25, 2024

1 Principle

This lab focuses on a standard communication interface (SPI) to interact with another chip. The slave component is a `MCP23S017` I/O Extender from Microchip that adds 2 8-bits GPIOs. The component exists in 2 versions, one with an `i2c` interface (`MCP23017`), and the other with a `spi` interface (`MCP23S17`). We will use the spi version. The functional diagram is in figure 1.

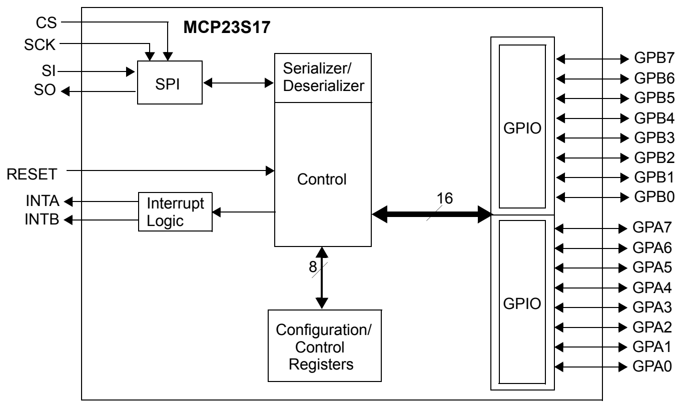

Figure 1: Functional Block Diagram of the MCP23S17.

1.1 Hardware Part

On the board, the component is the one with the board number, at the left of the TFT. The 2 ports are used as:

PORTA 8 leds (seen as `EXP A` on the board).

PORTB 4 DIP Switches (`B0` to `B3`), and 4 push buttons (`B4` to `B7`)

1.2 Software Part

This lab consists of writing a driver to control the component with high-level functions. The API (Application Programming Interface) is a set of high level functions that are available for an easy use of the component in the application. The lab implements these functions, one after the other.

The API interface may be written in `C++`, or in `C` if you are not comfortable with the object approach.

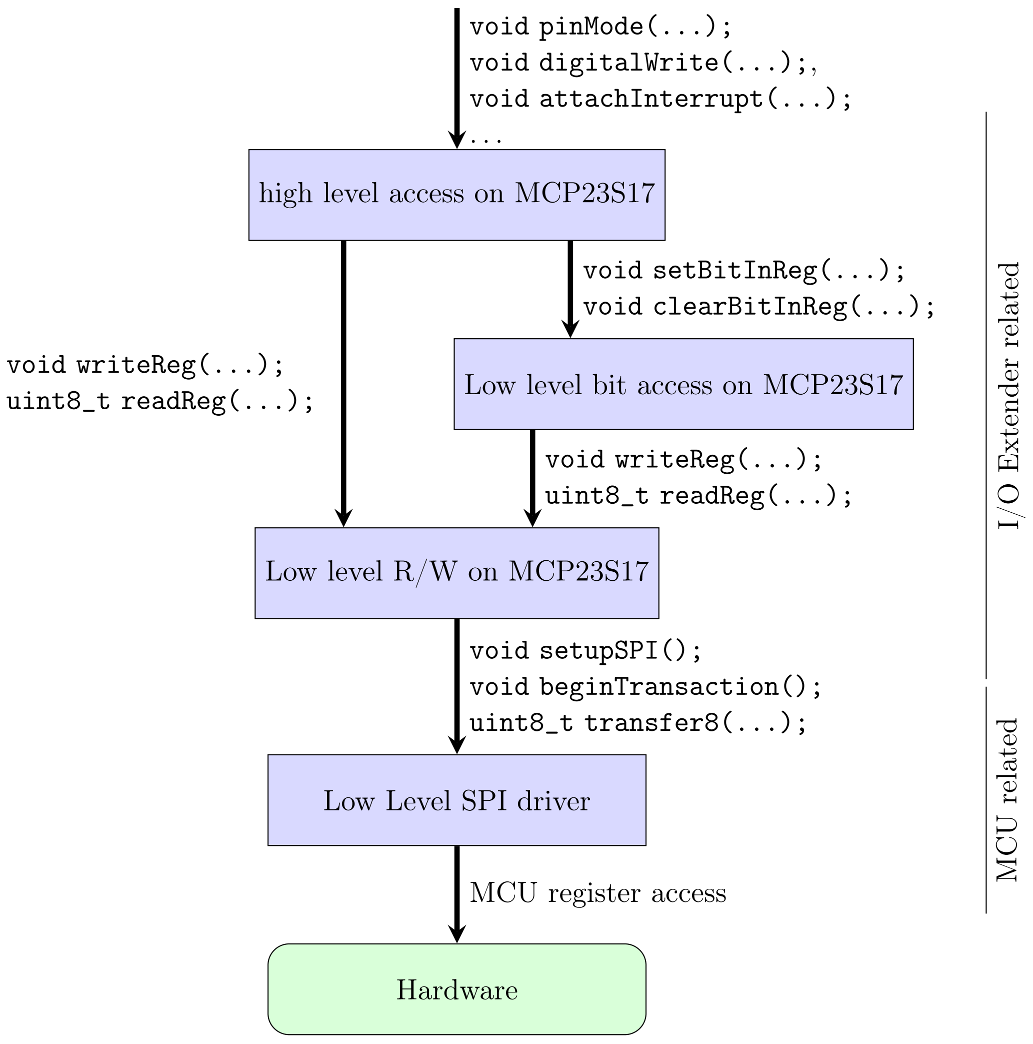

Figure 2: MCP23S17 Driver Architecture.

The architecture of the whole driver is defined in figure 2. The spi low-level driver is given (see files `spi.c/h`). The driver is organized as a stack of different stages, and arrows shows the relationship between each stage (API functions).

2 Low Level Driver

2.1 Remote Register access

The component is seen as a set of registers that can be read/written using the SPI interface. Microchip defines two modes for the register access (in `IOCON.BANK`) register field, which only have an impact on register addresses. In this lab, *we use only the default mode 0*. Registers are defined in the datasheet, table 1-2, p. 5.

In this section, basic functions to read/write to a remote register are defined (stage "*Low level R/W on MCP23S17*" on figure 2). The SPI communication frame is defined in figure 3, adapted from figure 1-5 of the datasheet, p. 8.

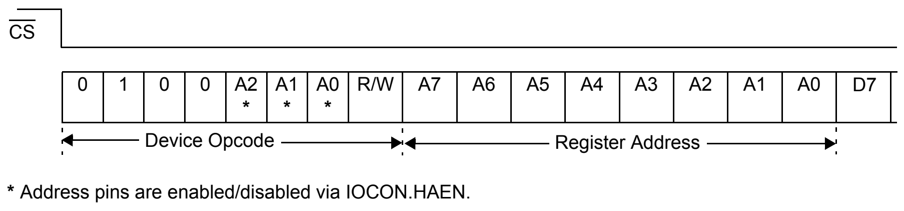

Figure 3: SPI Register Access. The A2-0 bits in the opcode field should be set to 0.

There must be an additionnal byte at the end of the frame: In write mode, this is the data that should be written to the register, and in read mode, this is the answer from the component.

The API of the low level driver consists in only 2 functions:

```
//write to a MCP register, using spi.
void writeReg(uint8_t reg, uint8_t val);
//read a MCP register, using spi.
uint8_t readReg(uint8_t reg);
```

Register addresses may be defined using a `#define` or an `enum` approach:

```
enum reg {  //use mode 0 (default)
  IODIRA    = 0x0, //direction input(1), output(0)
  IODIRB    = 0x1,
  IOPOLA    = 0x2, //polarity (toggle) -> not used
  //...
};

// or
//direction input(1), output(0)
#define IODIRA 0x0
#define IODIRB 0x1
#define IPOLA 0x2
//...
```

The R/W functions have to set the Chip Select, send a frame of 3 bytes (2W and 1R, or 3W), and unset the chip select.

Question 1 *Write the two low level functions to access to a remote registers. Check your configuration with the logic analyzer, and use the SPI protocol decoder.*
You can also check by writing to a register (IODIRA for example), then reading the value previously written.

Important note: The SPI speed is set to 8 MHz by default, and the logic analyzer must be configured for at least twice this frequency to get correct signals. Analyzers can sometimes crash at these frequencies. One solution is to decrease the frequency by uncommenting line 37 of the spi file `lib/spi.c`

```
35 // MCP23S17 SPI bus max frequency 10MHz. fPCLK = 64MHz
36 SPI1->CR1 = SPI_CR1_BR_1 |      // fPCLK/8   (BR=010) => 8MHz
37             // SPI_CR1_BR_2 |   // fPCLK/128 (BR=110) => 0.5MHz
38             SPI_CR1_MSTR;       // master mode
```

As soon as the low level functions are validated, you may comment it back... in the other case, the spi speed is divided by 16 and it will affect LCD timing access!

2.2 Remote register access: bit access

In this section, we add 2 useful functions to update only one bit of a remote register. This is the stack "*Low level bit access on MCP23S17*" of the driver in figure 2.

The functions definitions are:

```
void setBitInReg(uint8_t reg, uint8_t bitNum);
void clearBitInReg(uint8_t reg, uint8_t bitNum);
```

where `reg` is the remote register address defined in the previous section, and `bitNum` the bit number that should be updated.

These two functions *do not* access directly to the spi driver, but use the functions defined in the previous section.

Question 2 *Write these two functions to modify a single bit of a remote registers.*

3 High Level Driver

3.1 Output mode

The *High Level* driver stage can now be defined to allows an easy access to the device. It can be defined either in C or C++

3.1.1 Implementation in C

The interface may be in `C` language, with API functions:

```
enum port   {PORTA=0, PORTB=1};
enum mode   {MCP_OUTPUT=0,MCP_INPUT=1,MCP_INPUT_PULLUP=2};
enum itType {MCP_RISING, MCP_FALLING, MCP_BOTH};

//configure a pin
// - port is PORTA or PORTB
// - numBit is the pin number (0 to 7)
// - mode is in MCP_DISABLE, MCP_OUTPUT, MCP_INPUT, ...)
void mcpPinMode(enum port p, unsigned char bitNum, enum mode m);
```

3.1.2 Implementation in C++

With an object oriented approach, we can define one class for the MCP23s17 with only one instance. In that way, we encapsulate both data and functions related to the driver. This approach is the one of Arduino.

the interface would be for instance:

```
class mcp23s17 {
  public:
    enum port   {PORTA=0, PORTB=1};
    enum mode   {MCP_OUTPUT=0,MCP_INPUT=1,MCP_INPUT_PULLUP=2};
    enum itType {MCP_RISING, MCP_FALLING, MCP_BOTH};
  private:
    enum reg {  //use mode 0 (default)
      IODIRA    = 0x0, //direction input(1), output(0)
      IODIRB    = 0x1,
      //...
    };
  public:
    mcp23s17();
    void begin();
    //configure a pin
    // - port is PORTA or PORTB
    // - numBit is the pin number (0 to 7)
    // - mode is in DISABLE, OUTPUT, INPUT, ...)
    int pinMode(  port p,
            unsigned char bitNum,
            mode m);
    ...
};
```

Then, a single object is defined, at the end of the c++ implementation file (`.cpp` file):

```
mcp23s17 ioExt;
```

The object is declared as `extern` in the header file (`.h` file), so that it can be called by the application:

```
extern mcp23s17 ioExt;
```

In the application, it can be used like this: `ioExt.pinMode(...);`

Question 3 *Define the functions of the output mode of the ports. This means:*

- *`pinMode()` that configures a pin as output/input/input with pullup;*
- *`digitalWrite()` that controls a single pin:*

```
// high state if 'value' is different from 0
// low state if 'value' is 0.
void digitalWrite(port p,
                  unsigned char bitNum,
                  bool value);
```

Note: the type `bool` it defined in c++. In C, you should use an `uint8_t`.

3.2 First application

To test our driver, we want to make a single chaser with the leds of MCP GPIOA.

Question 4 *Write this single chaser, using your driver and a timer. Check the SPI communication using the logic analyzer. How long is a full transaction?*

3.3 Input Mode

The input mode is now easy to write, as the configuration function is already written (`pinMode()`).

Question 5 *write the input read function. We don't provide a `digitalRead()` but a function that reads the whole port:*

```
//read the whole port.
uint8_t readBits(port p);
```

Question 6 *update the application (chaser) so that Dip Switch 0 (`MCP PORTB.0`) defines the direction of the chaser.*
Note: *The switch **needs** an input pullup configuration.*

4 Interrupts

4.1 Basic Driver

The spi connection does not handle any interrupt management, but 2 external lines are provided (see figure 1), one for each port. The first line `INTA` is not connected (only leds on the port), but the line `INTB` is connected to the MCU (`PA9`).

To use interrupts, we have 2 make the following steps:

1 the MCP device is configured to detect a rising/falling edge on the port

2 the MCP device generates an interrupt, *i.e.* the pin `INTB` on the MCP connected to the STM32 gets into a `LOW` state.

3 the STM32 configures an external interrupt to receive the signal from the MCP device

4 the external interrupt handler is called

We first want to add a new function to the API:

```
//attach an interrupt to an input pin (port/bitNum)
void attachInterrupt(port p,          //e.g. mcp23s17::PORTB
                     uint8_t bitNum,  //e.g. 4
                     itType type);    //e.g. mcp23s17::MCP_FALLING
```

The interrupt mode should be configured in **active low** mode, which means that the interrupt pin goes low when there is an interrupt.

Question 7 ***Step 1:*** *Configure the MCP device so that the interrupt signal is generated on the MCP23S17 device (i.e. the interrupt pin goes low). An interrupt will be generated if there is an edge detected (either falling or rising). You may have to use at least* `INTCONA` *and* `GPINTENx`*. Check using an oscilloscope (or logic analyser) on the physical pin.*

Step 2 is done by hardware.

Question 8 ***Step 3:*** *Configure the STM32 to generate an interrupt on the external interrupt related to the pin of the MCP23S17. The interrupt on external line is* `EXT9`*. The handler, shared with other lines, is* `EXTI9_5_IRQHandler()`*.*

Question 9 ***Step 4:*** *The interrupt handler is called by the hardware, but we still have to filter according to the detected edge (falling,rising or both). In the interrupt routine, the register* `INTCAPB` *should be read to find the line (the* `bitNum`*) that has just generated the interrupt, and call the appropriate callback. Implement this interrupt service routine that should call an hard-coded function (that will be written by the user).*

4.2 Extended Driver

For greater flexibility, the user function associated to an interrupt may be a *callback function*. In our case, the callback function is a function with no argument[1]:

```
typedef void (*mcpCallBack)();
```

The function to associate an interrupt on one pin is now:

```
//attach an interrupt to an input pin (port/bitNum)
void attachInterrupt(port p,           //e.g. mcp23s17::PORTB
                     uint8_t bitNum,   //e.g. 4
                     itType type,      //e.g. mcp23s17::MCP_FALLING
                     mcpCallBack cb); //e.g. myIsr
```

And the user can now define its own callback:

```
void myIsr()
{
  //user defined interrupt service routine
}
```

One callback function is associated to each (port,pin), so there are 2×8 functions. A tabular of callbacks may be defined in the driver, for an easy access.

Question 10 *Provide an implementation of the `attachInterrupt` function that configures a (port,pin) and the interrupt handler that will call a dedicated callback function.*

4.3 Basic application

Question 11 *Use your new interrupt driver, so that the chaser direction is now toggled each time button 4 (`PORTB.4`) is pushed.*

The latency of such an IO extender is negligible when using an HMI (Human Machine Interface). However, for reactive systems, this may be important.

Question 12 *Determine the latency of an interrupt, from the input signal (button pushed) to the application software part (start of the user interrupt handler). You will have to use the logic analyzer.*

4.4 Full Application

The full application uses the 4 buttons. The mode depends on the last button pushed:

- BP4 basic cheaser

[1] A function pointer is no more than a pointer which points to the first instruction of a function, *i.e.* instead of storing the address of a data, it stores the address of the function. A remainder of the use of function pointers: `https://www.zentut.com/c-tutorial/c-function-pointer/`

- `BP5` leds are blinking
- `BP6` only odd leds are blinking
- `BP7` only even leds are blinking

Question 13 *Write and test the application.*