MCP23S17 Library

Generated by Doxygen 1.8.4

Wed Jun 25 2014 19:37:57

Contents

Chapter 1

MCP23S17

Arduino library for MCP23S17 IO Expanders

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

Class Documentation

3.1 MCP23S17 Class Reference

Public Member Functions

- MCP23S17 (SPIClass *spi, uint8_t cs, uint8_t addr)
- void begin ()
- void pinMode (uint8_t pin, uint8_t mode)
- void digitalWrite (uint8_t pin, uint8_t value)
- uint8_t digitalRead (uint8_t pin)
- uint8_t readPort (uint8_t port)
- uint16_t readPort ()
- void writePort (uint8_t port, uint8_t val)
- void writePort (uint16_t val)
- void enableInterrupt (uint8_t pin, uint8_t type)
- void disableInterrupt (uint8_t pin)
- void setMirror (boolean m)
- uint16_t getInterruptPins ()
- uint16_t getInterruptValue ()
- void setInterruptLevel (uint8_t level)
- void setInterruptOD (boolean openDrain)

3.1.1 Constructor & Destructor Documentation

3.1.1.1 MCP23S17::MCP23S17 (SPIClass * *spi*, uint8_t *cs*, uint8_t *addr*)

The constructor takes three parameters. The first is an SPI class pointer. This is the address of an SPI object (either the default SPI object on the Arduino, or an object made using the DSPIx classes on the chipKIT). The second parameter is the chip select pin number to use when communicating with the chip. The third is the internal address number of the chip. This is controlled by the three Ax pins on the chip.

Example:

```
MCP23S17 myExpander(&SPI, 10, 0);
```

3.1.2 Member Function Documentation

3.1.2.1 void MCP23S17::begin ( )

The begin function performs the initial configuration of the IO expander chip. Not only does it set up the SPI communications, but it also configures the chip for address-based communication and sets the default parameters and registers to sensible values.

Example:

```
myExpander.begin();
```

3.1.2.2 uint8_t MCP23S17::digitalRead (uint8_t *pin*)

This will return the current state of a pin set to INPUT, or the last value written to a pin set to OUTPUT.

Example:

```
byte value = myExpander.digitalRead(4);
```

3.1.2.3 void MCP23S17::digitalWrite (uint8_t *pin,* uint8_t *value*)

Like the Arduino API's namesake, this function will set an output pin to a specific value, either HIGH (1) or LOW (0). If the pin is currently set to an INPUT instead of an OUTPUT, then this function acts like the old way of enabling / disabling the pullup resistor, which pre-1.0.0 versions of the Arduino API used - i.e., set HIGH to enable the pullup, or LOW to disable it.

Example:

```
myExpander.digitalWrite(3, HIGH);
```

3.1.2.4 void MCP23S17::disableInterrupt (uint8_t *pin*)

This disables the interrupt functionality of a pin.

Example:

```
myExpander.disableInterrupt(4);
```

3.1.2.5 void MCP23S17::enableInterrupt (uint8_t *pin,* uint8_t *type*)

This enables the interrupt functionality of a pin. The interrupt type can be one of:

- CHANGE
- RISING
- FALLING

When an interrupt occurs the corresponding port's INT pin will be driven to it's configured level, and will remain there until either the port is read with a readPort or digitalRead, or the captured port status at the time of the interrupt is read using getInterruptValue.

Example:

```
myExpander.enableInterrupt(4, RISING);
```

3.1.2.6 uint16_t MCP23S17::getInterruptPins ( )

This function returns a 16-bit bitmap of the the pin or pins that have cause an interrupt to fire.

Example:

```
unsigned int pins = myExpander.getInterruptPins();
```

3.1.2.7 uint16_t MCP23S17::getInterruptValue ( )

This returns a snapshot of the IO pin states at the moment the last interrupt occured. Reading this value clears the interrupt status (and hence the INT pins) for the whole chip. Until this value is read (or the current live port value is read) no further interrupts can be indicated.

Example:

```
unsigned int pinValues = myExpander.getInterruptPins();
```

3.1.2.8 void MCP23S17::pinMode (uint8_t *pin*, uint8_t *mode*)

Just like the pinMode() function of the Arduino API, this function sets the direction of the pin. The first parameter is the pin nimber (0-15) to use, and the second parameter is the direction of the pin. There are standard Arduino macros for different modes which should be used. The supported macros are:

- OUTPUT
- INPUT
- INPUT_PULLUP

The INPUT_PULLUP mode enables the weak pullup that is available on any pin.

Example:

```
myExpander.pinMode(5, INPUT_PULLUP);
```

3.1.2.9 uint8_t MCP23S17::readPort (uint8_t *port*)

This function returns the entire 8-bit value of a GPIO port. Note that only the bits which correspond to a GPIO pin set to INPUT are valid. Other pins should be ignored. The only parameter defines which port (A/B) to retrieve: 0 is port A and 1 (or anything other than 0) is port B.

Example:

```
byte portA = myExpander.readPort(0);
```

3.1.2.10 uint16_t MCP23S17::readPort ( )

This is a full 16-bit version of the parameterised readPort function. This version reads the value of both ports and combines them into a single 16-bit value.

Example:

```
unsigned int value = myExpander.readPort();
```

3.1.2.11 void MCP23S17::setInterruptLevel (uint8_t *level*)

This sets the "active" level for an interrupt. HIGH means the interrupt pin will go HIGH when an interrupt occurs, LOW means it will go LOW.

Example:

```
myExpander.setInterruptLevel(HIGH);
```

3.1.2.12 void MCP23S17::setInterruptOD (boolean *openDrain*)

Using this function it is possible to configure the interrupt output pins to be open drain. This means that interrupt pins from multiple chips can share the same interrupt pin on the host MCU. This causes the level set by setInterruptLevel to be ignored. A pullup resistor will be required on the host MCU's interrupt pin.

Example:

```
myExpander.setInterruptOD(true);
```

3.1.2.13 void MCP23S17::setMirror (boolean *m*)

The two IO banks can have their INT pins connected together. This enables you to monitor both banks with just one interrupt pin on the host microcontroller. Calling setMirror with a parameter of *true* will enable this feature. Calling it with *false* will disable it.

Example:

```
myExpander.setMirror(true);
```

3.1.2.14 void MCP23S17::writePort (uint8_t *port,* uint8_t *val*)

This writes an 8-bit value to one of the two IO port banks (A/B) on the chip. The value is output direct to any pins on that bank that are set as OUTPUT. Any bits that correspond to pins set to INPUT are ignored. As with the readPort function the first parameter defines which bank to use (0 = A, 1+ = B).

Example:

```
myExpander.writePort(0, 0x55);
```

3.1.2.15 void MCP23S17::writePort (uint16_t *val*)

This is the 16-bit version of the writePort function. This takes a single 16-bit value and splits it between the two IO ports, the upper half going to port B and the lower to port A.

Example:

```
myExpander.writePort(0x55AA);
```

The documentation for this class was generated from the following files:

- MCP23S17.h
- MCP23S17.cpp

Index