I2C-Dimmer

Generated by Doxygen 1.7.2

Sat Dec 10 2011 12:16:43

Contents

Chapter 1

I2C-dimmer

1.1 Introduction

I haven't done many microcontroller-projects till now, but more than one of the few projects I did involved controlling LEDs by pulse width modulation (PWM). Doing this for one or more LEDs is a stressful task for a little microcontroller, but if you want to do some other more or less complicated things while keeping LEDs at certain brightnesses is likely to ruin the timings that are used in the PWM. Not to talk about the program code, which gets more and more unreadable if you try to do several different things 'at the same time'.

For my next project I need to fade some LEDs again, so I was looking for an easier way to do it. The plans include reading from memory cards, talking to real time clocks and displaying text on an LCD, so I'm almost sure that I won't be able to reliably control the five channels I'm going to use.

The first plan was to use a ready-made chip. I looked around and the best thing I could find was one made by Philips (PCA something, I forgot the number) that can be controlled via I2C-bus. That part is able to control eight LEDs, but apart from 'on' and 'off' you can set the LEDs only to two different brightnesses. Those are variable, nevertheless, but it would be impossible to light one LED at 20%, one at 50% and one at 80%. Another drawback is that it is SMD-only, and my soldering-skills don't including working with stuff that small.

So the Idea was to set up a separate controller for LED-fading, that can be externally controlled, ideally via I2C-bus since I intend to use several other devices in my next project that can make use of the bus. So I set up an ATtiny2313 on my breadboard, clocked it with an external 20MHz-crystal and we tried to control as many LEDs as possible...

1.2 Pulse width modulation

1.2.1 The old way

Controlling the brightness of LEDs by PWM is a common technique, I used it myself in several projects. Till now I used to switch on all LEDs that should light up at a level greater than zero, waited till the first of the LEDs has to be switched off, switched it off, waited for the next one and so on. After a certain time all LEDs are switched off, and I start again.

I try to visualize that with a little picture:

In this example, a full cycle of the PWM would need 50 units of time. The first LED is switched on the full time (100%), the second for 40 of the 50 units (80%), the third one for ten (20%) and the fifth one for 30 units (60%). The fourth LED is off (0%). We see that after 50 units of time the modulation starts again.

The drawback of this technique is, that it's slow. And for each additional channel you try to control, it gets even slower. We tried, but we weren't able to control more than five LEDs in this way without them to start flickering to a visible amount.

We tried to create an array with all states of the process, so the PWM only would have to loop through the array and set the outputs accordingly. But that didn't work either, because the used microcontroller doesn't have enough RAM to store the array.

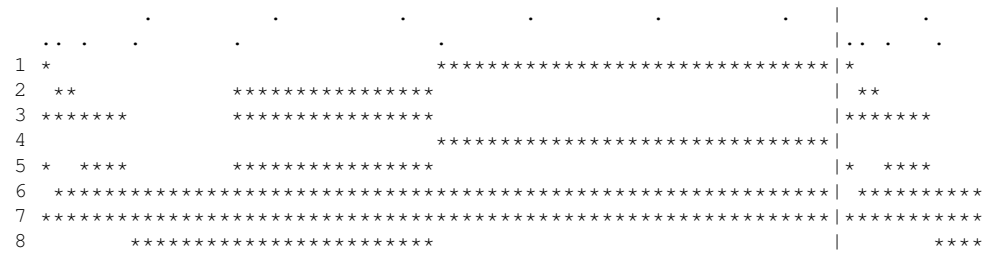
1.2.2 Thomas' idea

After some tests that didn't work out too well, Thomas had a great idea how to implement the PWM. It also works with an array for all states, but the states of the modulation are not displayed for the same time. The first state is displayed for one time-unit, the second one for two time-units, the third one for four and so on. In this way the LEDs are turned on and off more than once per cycle of the PWM, but that doesn't hurt.

Let's try to paint a picture again:

So here we see a PWM with eight channels that are able to display up to 64 different brightnesses. Channel one is switched on for one unit of time, channel two for two units and so on. The most interesting thing is on channel five: the LED is switched on for one unit of time, switched off, and switched on again for four units of time.

Lets try a more complicated example -- with brighter LEDs, too:



The channels 1 to 8 have the brightnesses 33, 18, 23, 32, 21, 63, 64 and 24.

The advantage of this technique is that on the one hand you have to save a limited number of states (six states in the example), and the looping through the states is very simple: state n is sent to the output pins, then we wait for 2^(n-1) time units, then the next state is sent.

Each state represents the bit-pattern that has to be sent during one step. In other words: one column out of the above picture at the start of a new time period. So in this example, we have six states: 01010101, 01100110, 01110100, 11100000, 11110110 and 01101001. The first one is displayed for one unit of time, the second one for two units, the third one for four units and so on...

Using this technique has the advantage that adding more channels does almost nothing in terms of system load. The only time that the algorithm has to do actual calculations is when a new value has been delivered and has to be converted into the states. **So using this algorithm, it is possible to show different brightnesses on all free pins of the controller. With an ATtiny2313 that means that you can fade 13 different LEDs while still talking I2C to communicate with other devices!**

1.3 I2C communication

Speaking I2C is no rocket science, but since one has to do a lot of bit-shifting when implementing it, I took a ready-made library.

The one I used is **written by Donald R. Blake**, he was so kind to put it under GPL and post it to avrfreaks.net. You can find the original post in a thread called '8 bit communication between AVR using TWI' and some additions in the thread 'I2C Slave on an ATtiny45'.

Thanks for the great work, Donald! And for putting it under a free license.

Since his package seems to be only available as a forum-attachment, and I'm not sure for how long that will be, I included it into the tarball of this project.

1.4 Building and installing

The firmware is built and installed on the controller with the included makefile. You might need to need to customize it to match your individual environment.

Don't forget to set the fuses on the controller to make use of the external crystal. This project is using a fine algorithm, but it still needs the full power of 20MHz. The settings I used are included in the makefile, too.

Oh, and if you want the slave to use an I2C-address different from 0x10: no problem. Just change it in the code.

1.5 Usage

You should be able to use this device in the same way you would use any other I2C-slave:

1.5.1 Connecting it

The controller needs to have the following pins connected in the circuit:

- Pin 1 - Reset - should be connected to VCC with a 10k-resistor
- Pin 4 and 5 - XTAL1 and XTAL2 - connected to a 20MHz-crystal, using 22p-capacitors against GND
- Pin 10 - GND - Ground
- Pin 17 - SDA - I2C-data
- Pin 19 - SCL - I2C-clock
- Pin 20 - VCC - 5V

Your I2C-data and -clock lines should be terminated by 4,7k-resistors to pull up the lines. All the other pins can be used to connect LEDs. They are arranged in this way:

- Pin 2 - PD0 - Channel 0
- Pin 3 - PD1 - Channel 1
- Pin 6 - PD2 - Channel 2
- Pin 7 - PD3 - Channel 3
- Pin 8 - PD4 - Channel 4
- Pin 9 - PD5 - Channel 5
- Pin 11 - PD6 - Channel 6

- Pin 12 - PB0 - Channel 7
- Pin 13 - PB1 - Channel 8
- Pin 14 - PB2 - Channel 9
- Pin 15 - PB3 - Channel 10
- Pin 16 - PB4 - Channel 11
- Pin 18 - PB6 - Channel 12

1.5.2 Talking to it

For my tests I used an ATmega8 as I2C-master with the library **written by Peter Fleury**. You can find it on http://jump.to/fleury. Thanks to him for putting it online!

The typical send function looks like this:

```
#define I2C_DIMMER 0x10

void sendi2cBytes(uint8_t address, uint8_t brightness) {
    // address: number of the LED to set (0..12)
    // brightness: value between 0 and 127
    // start the communication...
    i2c_start_wait((I2C_DIMMER << 1) + I2C_WRITE);
    // write a byte with the address. we want the highest bit of the
    // address to be 1, so the slave can be sure that this is an address.
    i2c_write(address | 0x80);
    // calculate the actual duration the LED should light up. we could do
    // this on the slave's side, but we assume that the device is more
    // flexible when it is done on the master side.
    uint16_t duration = (brightness + 1) * (brightness + 1) - 1;
    // calculate the low- and the high-byte and send it. note that we split
    // the duration into 7-bit-values, not 8 bit! in this way the highest
    // bit of the transferred bytes is always low, allowing the slave to
    // recognize the transmitted bytes as values, not as addresses.
    i2c_write(duration & 0x7f);        // low byte
    i2c_write((duration >> 7) & 0x7f); // high byte
    // stop the communication...
    i2c_stop();
}
```

1.6 Drawbacks

Till now, the device worked in all situations I tested it in. So far everything is fine.

I guess that, compared to the ready-made off-the-hook-parts that controls LEDs via I2C, this module is a bit slow. I can't see any flickering in the LEDs since they are still switched very fast (about every 6ms, which would result in a 166Hz flickering -- too fast for me).

1.7 Files in the distribution

- *Readme.txt:* Documentation, created from the htmldoc-directory.
- *htmldoc/*: Documentation, created from main.c.
- *refman.pdf:* Documentation, created from main.c.
- *main.c:* Source code of the firmware.
- *main_*.hex: Compiled version of the firmware.
- *usiTwiSlave.c:* I2C-library.
- *usiTwiSlave.h:* I2C-library.
- *USI_TWI_Slave.zip:* I2C-library (package).
- *i2c-dimmer.doxygen*: Support for creating the documentation.
- *License.txt:* Public license for all contents of this project, except for the USB driver. Look in firmware/usbdrv/License.txt for further info.
- *Changelog.txt:* Logfile documenting changes in soft-, firm- and hardware.

1.8 Thanks!

Once again, special credits go to **Thomas Stegemann**. He had the great idea for the PWM-algorithm, and I am always astonished by the patience he has to show me how to do anything complicated in a sick language like C...

1.9 About the license

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 Command Struct Reference

Holds one command that is received via i2c.

Data Fields

- uint8_t address

  number of output channel (between 0 and CHANNEL_COUNT-1

- uint16_t value

  *value to be assigned to the channel (between 0 and 128*128-1 = 16383*

- ReadCommandState state

  what are we waiting for?

4.1.1 Detailed Description

Holds one command that is received via i2c. The command consists of an address (number of output channel) and a 16-bit value. The state is used to indicate what part of the next command we are waiting for.

Definition at line 367 of file main.c.

4.1.2 Field Documentation

4.1.2.1 uint8_t Command::address

number of output channel (between 0 and CHANNEL_COUNT-1

Definition at line 368 of file main.c.

Referenced by evaluate_i2c_input().

4.1.2.2 ReadCommandState Command::state

what are we waiting for?

Definition at line 372 of file main.c.

Referenced by evaluate_i2c_input().

4.1.2.3 uint16_t Command::value

value to be assigned to the channel (between 0 and 128∗128-1 = 16383

Definition at line 370 of file main.c.

Referenced by evaluate_i2c_input().

The documentation for this struct was generated from the following file:

- main.c

Chapter 5

File Documentation

5.1 main.c File Reference

Firmware for the i2c-dimmer.

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/wdt.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#include "usiTwiSlave.h"
```

Data Structures

- struct Command

  Holds one command that is received via i2c.

Defines

- #define TWI_SLA 0x10

  i2c slave address

- #define CHANNEL_COUNT 13

  number of 'fadeable' channels

- #define PORT_COUNT 2

  the channels are distributed over two ports

- #define OUTPORT0 PORTB

  output port 0

- #define OUTDDR0 DDRB

  set port 0 to be output

- #define OUTMASK0 0x5F

  see channel_pin, channel_port

- #define OUTPORT1 PORTD

  output port 0

- #define OUTDDR1 DDRD

  set port 0 to be output

- #define OUTMASK1 0x7F

  see channel_pin, channel_port

- #define STATE_COUNT 14

  number of states for pwm

- #define STATE_START_COUNT 2

  number of state groups to be treated individually

Enumerations

- enum ReadCommandState { WAIT_FOR_ADDRESS, WAIT_FOR_VALUE_LOW, WAIT_FOR_VALUE_HIGH }

  Three bytes have to be received for a full command.

Functions

- void timer_start ()

  initialize timer

- void set_brightness (uint8_t channel, uint16_t brightness)

  Set brightness on one channel.

- void init_ports (void)

  initialize hardware

- void set_port (int port, uint8_t state)

  set output

- void evaluate_i2c_input (void)

  Check if anything has been received via i2c and evaluate the received data.

- int main (void)

  Main-function.

Variables

- const uint8_t channel_port[CHANNEL_COUNT] PROGMEM

  We want to drive as many channels as possible.

- const uint8_t switch_timer_index [STATE_START_COUNT] = { 13, 0 }

  start interval of the state groups

- uint8_t switch_state [STATE_COUNT][PORT_COUNT]

  contains the port assignments for each interval

- uint8_t switch_state_new [STATE_COUNT][PORT_COUNT]
- Command command = {0, 0, WAIT_FOR_ADDRESS}

  the next command is built in this variable

5.1.1 Detailed Description

Firmware for the i2c-dimmer.

Author

Ronald Schaten <ronald@schatenseite.de> & Thomas Stegemann

Version

Id:

main.c,v 1.1 2007/07/29 17:19:50 rschaten Exp

Definition in file main.c.

5.1.2 Define Documentation

5.1.2.1 #define CHANNEL_COUNT 13

number of 'fadeable' channels

Definition at line 309 of file main.c.

Referenced by main().

5.1.2.2 #define OUTDDR0 DDRB

set port 0 to be output

Definition at line 313 of file main.c.

Referenced by init_ports().

5.1.2.3 #define OUTDDR1 DDRD

set port 0 to be output

Definition at line 317 of file main.c.

Referenced by init_ports().

5.1.2.4 #define OUTMASK0 0x5F

see channel_pin, channel_port

Definition at line 314 of file main.c.

Referenced by init_ports(), and set_port().

5.1.2.5 #define OUTMASK1 0x7F

see channel_pin, channel_port

Definition at line 318 of file main.c.

Referenced by init_ports(), and set_port().

5.1.2.6 #define OUTPORT0 PORTB

output port 0

Definition at line 312 of file main.c.

Referenced by init_ports(), and set_port().

5.1.2.7 #define OUTPORT1 PORTD

output port 0

Definition at line 316 of file main.c.

Referenced by init_ports(), and set_port().

5.1.2.8 #define PORT_COUNT 2

the channels are distributed over two ports

Definition at line 310 of file main.c.

Referenced by main().

5.1.2.9 #define STATE_COUNT 14

number of states for pwm

Definition at line 340 of file main.c.

Referenced by main(), and set_brightness().

5.1.2.10 #define STATE_START_COUNT 2

number of state groups to be treated individually

Definition at line 341 of file main.c.

Referenced by main().

5.1.2.11 #define TWI_SLA 0x10

i2c slave address

Definition at line 307 of file main.c.

Referenced by main().

5.1.3 Enumeration Type Documentation

5.1.3.1 enum ReadCommandState

Three bytes have to be received for a full command.

This enum is used to indicate what part of the command we are waiting for.

Enumerator:

WAIT_FOR_ADDRESS first byte is the address

WAIT_FOR_VALUE_LOW second byte is the lower part of the value

WAIT_FOR_VALUE_HIGH third byte is the higher part of the value

Definition at line 356 of file main.c.

5.1.4 Function Documentation

5.1.4.1 void evaluate_i2c_input (void)

Check if anything has been received via i2c and evaluate the received data.

The received data is set into the command variable according to the state of the command we are waiting for.

Definition at line 449 of file main.c.

References Command::address, set_brightness(), Command::state, usiTwiDataInReceiveBuffer(), usiTwiReceiveByte(), Command::value, WAIT_FOR_ADDRESS, WAIT_FOR_VALUE_HIGH, and WAIT_FOR_VALUE_LOW.

Referenced by main().

5.1.4.2 void init_ports (void)

initialize hardware

Definition at line 418 of file main.c.

References OUTDDR0, OUTDDR1, OUTMASK0, OUTMASK1, OUTPORT0, and OUTPORT1.

Referenced by main().

5.1.4.3 int main (void)

Main-function.

Initializes everything and contains the main loop which controls the actual PWM output.

Returns

An integer. Whatever... :-)

Definition at line 496 of file main.c.

References CHANNEL_COUNT, evaluate_i2c_input(), init_ports(), PORT_COUNT, set_brightness(), set_port(), STATE_COUNT, STATE_START_COUNT, switch_state, switch_state_new, switch_timer_index, timer_start(), TWI_SLA, and usiTwiSlaveInit().

5.1.4.4 void set_brightness (uint8_t *channel,* uint16_t *brightness*)

Set brightness on one channel.

Parameters

channel	the channel to address (0 .. CHANNEL_COUNT)
brightness	the value to set (0 .. 16383)

Definition at line 395 of file main.c.

References STATE_COUNT, and switch_state_new.

Referenced by evaluate_i2c_input(), and main().

5.1.4.5 void set_port (int *port,* uint8_t *state*)

set output

Parameters

port	port to set
state	value to be sent to the port

Definition at line 431 of file main.c.

References OUTMASK0, OUTMASK1, OUTPORT0, and OUTPORT1.

Referenced by main().

5.1.4.6 void timer_start ( )

initialize timer

Definition at line 381 of file main.c.

Referenced by main().

5.1.5 Variable Documentation

5.1.5.1 Command command = {0, 0, WAIT_FOR_ADDRESS}

the next command is built in this variable

Definition at line 376 of file main.c.

5.1.5.2 const uint16_t switch_timer [STATE_COUNT] PROGMEM

Initial value:

```
{
      0,    0,    0,    0,    0,    0,
      1,    1,    1,    1,    1,    1,    1 }
```

We want to drive as many channels as possible.

interval length of the states

this is used to determine the pin that is used for output

Unfortunately the usable pins aren't 'in a row', so we have to determine which channel ends up on which port and pin. this is used to determine the port that is used for output

Definition at line 326 of file main.c.

5.1.5.3 uint8_t switch_state[STATE_COUNT][PORT_COUNT]

contains the port assignments for each interval

Definition at line 349 of file main.c.

Referenced by main().

5.1.5.4 uint8_t switch_state_new[STATE_COUNT][PORT_COUNT]

Definition at line 351 of file main.c.

Referenced by main(), and set_brightness().

5.1.5.5 const uint8_t switch_timer_index[STATE_START_COUNT] = { 13, 0 }

start interval of the state groups

Definition at line 346 of file main.c.

Referenced by main().

5.2 usiTwiSlave.c File Reference

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include "usiTwiSlave.h"
```

Defines

- #define SET_USI_TO_SEND_ACK()
- #define SET_USI_TO_READ_ACK()
- #define SET_USI_TO_TWI_START_CONDITION_MODE()
- #define SET_USI_TO_SEND_DATA()
- #define SET_USI_TO_READ_DATA()

Enumerations

- enum overflowState_t {

  USI_SLAVE_CHECK_ADDRESS = 0x00, USI_SLAVE_SEND_DATA = 0x01, USI_-SLAVE_REQUEST_REPLY_FROM_SEND_DATA = 0x02, USI_SLAVE_CHECK_-REPLY_FROM_SEND_DATA = 0x03,

  USI_SLAVE_REQUEST_DATA = 0x04, USI_SLAVE_GET_DATA_AND_SEND_-ACK = 0x05 }

Functions

- void usiTwiSlaveInit (uint8_t ownAddress)
- void usiTwiTransmitByte (uint8_t data)
- uint8_t usiTwiReceiveByte (void)
- bool usiTwiDataInReceiveBuffer (void)
- ISR (USI_START_VECTOR)
- ISR (USI_OVERFLOW_VECTOR)

5.2.1 Define Documentation

5.2.1.1 #define SET_USI_TO_READ_ACK( )

Value:

```
{ \
  /* set SDA as input */ \
  DDR_USI &= ~( 1 << PORT_USI_SDA ); \
  /* prepare ACK */ \
  USIDR = 0; \
  /* clear all interrupt flags, except Start Cond */ \
  USISR = \
       ( 0 << USI_START_COND_INT ) | \
       ( 1 << USIOIF ) | \
       ( 1 << USIPF ) | \
       ( 1 << USIDC ) | \
       /* set USI counter to shift 1 bit */ \
       ( 0x0E << USICNT0 ); \
}
```

Definition at line 165 of file usiTwiSlave.c.

Referenced by ISR().

5.2.1.2 #define SET_USI_TO_READ_DATA( )

Value:

```
{ \
  /* set SDA as input */ \
  DDR_USI &= ~( 1 << PORT_USI_SDA ); \
```

```
  /* clear all interrupt flags, except Start Cond */ \
  USISR    = \
       ( 0 << USI_START_COND_INT ) | ( 1 << USIOIF ) | \
       ( 1 << USIPF ) | ( 1 << USIDC ) | \
       /* set USI to shift out 8 bits */ \
       ( 0x0 << USICNT0 ); \
}
```

Definition at line 211 of file usiTwiSlave.c.

Referenced by ISR().

5.2.1.3 #define SET_USI_TO_SEND_ACK( )

Value:

```
{ \
  /* prepare ACK */ \
  USIDR = 0; \
  /* set SDA as output */ \
  DDR_USI |= ( 1 << PORT_USI_SDA ); \
  /* clear all interrupt flags, except Start Cond */ \
  USISR = \
       ( 0 << USI_START_COND_INT ) | \
       ( 1 << USIOIF ) | ( 1 << USIPF ) | \
       ( 1 << USIDC )| \
       /* set USI counter to shift 1 bit */ \
       ( 0x0E << USICNT0 ); \
}
```

Definition at line 150 of file usiTwiSlave.c.

Referenced by ISR().

5.2.1.4 #define SET_USI_TO_SEND_DATA( )

Value:

```
{ \
  /* set SDA as output */ \
  DDR_USI |=  ( 1 << PORT_USI_SDA ); \
  /* clear all interrupt flags, except Start Cond */ \
  USISR    =  \
       ( 0 << USI_START_COND_INT ) | ( 1 << USIOIF ) | ( 1 << USIPF ) | \
       ( 1 << USIDC) | \
       /* set USI to shift out 8 bits */ \
       ( 0x0 << USICNT0 ); \
}
```

Definition at line 199 of file usiTwiSlave.c.

Referenced by ISR().

5.2.1.5 #define SET_USI_TO_TWI_START_CONDITION_MODE( )

Value:

```
{ \
  USICR = \
       /* enable Start Condition Interrupt, disable Overflow Interrupt */ \
       ( 1 << USISIE ) | ( 0 << USIOIE ) | \
       /* set USI in Two-wire mode, no USI Counter overflow hold */ \
       ( 1 << USIWM1 ) | ( 0 << USIWM0 ) | \
       /* Shift Register Clock Source = External, positive edge */ \
       /* 4-Bit Counter Source = external, both edges */ \
       ( 1 << USICS1 ) | ( 0 << USICS0 ) | ( 0 << USICLK ) | \
       /* no toggle clock-port pin */ \
       ( 0 << USITC ); \
  USISR = \
        /* clear all interrupt flags, except Start Cond */ \
        ( 0 << USI_START_COND_INT ) | ( 1 << USIOIF ) | ( 1 << USIPF ) | \
        ( 1 << USIDC ) | ( 0x0 << USICNT0 ); \
}
```

Definition at line 181 of file usiTwiSlave.c.

Referenced by ISR().

5.2.2 Enumeration Type Documentation

5.2.2.1 enum overflowState_t

Enumerator:

USI_SLAVE_CHECK_ADDRESS

USI_SLAVE_SEND_DATA

USI_SLAVE_REQUEST_REPLY_FROM_SEND_DATA

USI_SLAVE_CHECK_REPLY_FROM_SEND_DATA

USI_SLAVE_REQUEST_DATA

USI_SLAVE_GET_DATA_AND_SEND_ACK

Definition at line 231 of file usiTwiSlave.c.

5.2.3 Function Documentation

5.2.3.1 ISR (USI_OVERFLOW_VECTOR)

Definition at line 495 of file usiTwiSlave.c.

References SET_USI_TO_READ_ACK, SET_USI_TO_READ_DATA, SET_USI_TO_-SEND_ACK, SET_USI_TO_SEND_DATA, SET_USI_TO_TWI_START_CONDITION_-MODE, TWI_RX_BUFFER_MASK, TWI_TX_BUFFER_MASK, USI_SLAVE_CHECK_-ADDRESS, USI_SLAVE_CHECK_REPLY_FROM_SEND_DATA, USI_SLAVE_GET_DATA_-AND_SEND_ACK, USI_SLAVE_REQUEST_DATA, USI_SLAVE_REQUEST_REPLY_-FROM_SEND_DATA, and USI_SLAVE_SEND_DATA.

5.2.3.2 ISR (USI_START_VECTOR)

Definition at line 413 of file usiTwiSlave.c.

References USI_SLAVE_CHECK_ADDRESS.

5.2.3.3 bool usiTwiDataInReceiveBuffer (void)

Definition at line 395 of file usiTwiSlave.c.

Referenced by evaluate_i2c_input().

5.2.3.4 uint8_t usiTwiReceiveByte (void)

Definition at line 374 of file usiTwiSlave.c.

References TWI_RX_BUFFER_MASK.

Referenced by evaluate_i2c_input().

5.2.3.5 void usiTwiSlaveInit (uint8_t *ownAddress*)

Definition at line 298 of file usiTwiSlave.c.

Referenced by main().

5.2.3.6 void usiTwiTransmitByte (uint8_t *data*)

Definition at line 348 of file usiTwiSlave.c.

References TWI_TX_BUFFER_MASK.

5.3 usiTwiSlave.h File Reference

```
#include <stdbool.h>
```

Defines

- #define TWI_RX_BUFFER_SIZE (16)
- #define TWI_RX_BUFFER_MASK (TWI_RX_BUFFER_SIZE - 1)
- #define TWI_TX_BUFFER_SIZE (16)
- #define TWI_TX_BUFFER_MASK (TWI_TX_BUFFER_SIZE - 1)

Functions

- void usiTwiSlaveInit (uint8_t)

- void usiTwiTransmitByte (uint8_t)
- uint8_t usiTwiReceiveByte (void)
- bool usiTwiDataInReceiveBuffer (void)

5.3.1 Define Documentation

5.3.1.1 #define TWI_RX_BUFFER_MASK (TWI_RX_BUFFER_SIZE - 1)

Definition at line 71 of file usiTwiSlave.h.

Referenced by ISR(), and usiTwiReceiveByte().

5.3.1.2 #define TWI_RX_BUFFER_SIZE (16)

Definition at line 70 of file usiTwiSlave.h.

5.3.1.3 #define TWI_TX_BUFFER_MASK (TWI_TX_BUFFER_SIZE - 1)

Definition at line 80 of file usiTwiSlave.h.

Referenced by ISR(), and usiTwiTransmitByte().

5.3.1.4 #define TWI_TX_BUFFER_SIZE (16)

Definition at line 79 of file usiTwiSlave.h.

5.3.2 Function Documentation

5.3.2.1 bool usiTwiDataInReceiveBuffer (void)

Definition at line 395 of file usiTwiSlave.c.

Referenced by evaluate_i2c_input().

5.3.2.2 uint8_t usiTwiReceiveByte (void)

Definition at line 374 of file usiTwiSlave.c.

References TWI_RX_BUFFER_MASK.

Referenced by evaluate_i2c_input().

5.3.2.3 void usiTwiSlaveInit (uint8_t)

Definition at line 298 of file usiTwiSlave.c.

Referenced by main().

5.3.2.4 void usiTwiTransmitByte (uint8_t)

Definition at line 348 of file usiTwiSlave.c.

References TWI_TX_BUFFER_MASK.

Index