```
/*****************************************************************************
*
* Atmel Corporation
*
* File              : TWI_Slave.c
* Compiler          : IAR EWAAVR 2.28a/3.10c
* Revision          : $Revision: 2475 $
* Date              : $Date: 2007-09-20 12:00:43 +0200 (to, 20 sep 2007) $
* Updated by        : $Author: mlarsson $
*
* Support mail      : avr@atmel.com
*
* Supported devices : All devices with a TWI module can be used.
*                     The example is written for the ATmega16
*
* AppNote           : AVR311 - TWI Slave Implementation
*
* Description       : This is sample driver to AVRs TWI module.
*                     It is interupt driveren. All functionality is controlled
through
*                     passing information to and from functions. Se main.c for
samples
*                     of how to use the driver.
*
****************************************************************************/
/*! \page MISRA
 *
 * General disabling of MISRA rules:
 * * (MISRA C rule 1) compiler is configured to allow extensions
 * * (MISRA C rule 111) bit fields shall only be defined to be of type unsigned int
or signed int
 * * (MISRA C rule 37) bitwise operations shall not be performed on signed integer
types
 * As it does not work well with 8bit architecture and/or IAR

 * Other disabled MISRA rules
 * * (MISRA C rule 109) use of union - overlapping storage shall not be used
 * * (MISRA C rule 61) every non-empty case clause in a switch statement shall be
terminated with a break statement
*/

#if defined(__ICCAVR__)
#include "ioavr.h"
#include "inavr.h"
#else
#include <avr/io.h>
#include <avr/interrupt.h>
#endif
```

```
#include "TWI_slave.h"

// Emulate GCC ISR() statement in IAR
#if defined(__ICCAVR__)
#define PRAGMA(x) _Pragma( #x )
#define ISR(vec) PRAGMA( vector=vec ) __interrupt void handler_##vec(void)
#endif

static unsigned char TWI_buf[TWI_BUFFER_SIZE];     // Transceiver buffer. Set the
size in the header file
static unsigned char TWI_msgSize  = 0;             // Number of bytes to be
transmitted.
static unsigned char TWI_state    = TWI_NO_STATE;  // State byte. Default set to
TWI_NO_STATE.

// This is true when the TWI is in the middle of a transfer
// and set to false when all bytes have been transmitted/received
// Also used to determine how deep we can sleep.
static volatile unsigned char TWI_busy = 0;

union TWI_statusReg_t TWI_statusReg = {0};           // TWI_statusReg is defined in
TWI_Slave.h

/****************************************************************************
Call this function to set up the TWI slave to its initial standby state.
Remember to enable interrupts from the main application after initializing the TWI.
Pass both the slave address and the requrements for triggering on a general call in
the
same byte. Use e.g. this notation when calling this function:
TWI_Slave_Initialise( (TWI_slaveAddress<<TWI_ADR_BITS) | (TRUE<<TWI_GEN_BIT) );
The TWI module is configured to NACK on any requests. Use a TWI_Start_Transceiver
function to
start the TWI.
****************************************************************************/
void TWI_Slave_Initialise( unsigned char TWI_ownAddress )
{
  TWAR = TWI_ownAddress;                            // Set own TWI slave address.
Accept TWI General Calls.
  TWCR = (1<<TWEN)|                                 // Enable TWI-interface and
release TWI pins.
         (0<<TWIE)|(0<<TWINT)|                      // Disable TWI Interupt.
         (0<<TWEA)|(0<<TWSTA)|(0<<TWSTO)|           // Do not ACK on any requests,
yet.
         (0<<TWWC);                                 //
  TWI_busy = 0;
}

/****************************************************************************
```

```
Call this function to test if the TWI_ISR is busy transmitting.
****************************************************************************/
unsigned char TWI_Transceiver_Busy( void )
{
  return TWI_busy;
}

/****************************************************************************
Call this function to fetch the state information of the previous operation. The
function will hold execution (loop)
until the TWI_ISR has completed with the previous operation. If there was an error,
then the function
will return the TWI State code.
****************************************************************************/
unsigned char TWI_Get_State_Info( void )
{
  while ( TWI_Transceiver_Busy() ) {}             // Wait until TWI has completed
the transmission.
  return ( TWI_state );                         // Return error state.
}

/****************************************************************************
Call this function to send a prepared message, or start the Transceiver for
reception. Include
a pointer to the data to be sent if a SLA+W is received. The data will be copied to
the TWI buffer.
Also include how many bytes that should be sent. Note that unlike the similar Master
function, the
Address byte is not included in the message buffers.
The function will hold execution (loop) until the TWI_ISR has completed with the
previous operation,
then initialize the next operation and return.
****************************************************************************/
void TWI_Start_Transceiver_With_Data( unsigned char *msg, unsigned char msgSize )
{
  unsigned char temp;

  while ( TWI_Transceiver_Busy() ) {}             // Wait until TWI is ready for
next transmission.

  TWI_msgSize = msgSize;                        // Number of data to transmit.
  for ( temp = 0; temp < msgSize; temp++ )      // Copy data that may be transmitted
if the TWI Master requests data.
  {
```

```
 TWI_buf[ temp ] = msg[ temp ];
  }
  TWI_statusReg.all = 0;
  TWI_state         = TWI_NO_STATE ;
  TWCR = (1<<TWEN)|                             // TWI Interface enabled.
         (1<<TWIE)|(1<<TWINT)|                  // Enable TWI Interupt and clear the
flag.
         (1<<TWEA)|(0<<TWSTA)|(0<<TWSTO)|       // Prepare to ACK next time the
Slave is addressed.
         (0<<TWWC);                             //
  TWI_busy = 1;
}

/****************************************************************************
Call this function to start the Transceiver without specifing new transmission data.
Useful for restarting
a transmission, or just starting the transceiver for reception. The driver will
reuse the data previously put
in the transceiver buffers. The function will hold execution (loop) until the
TWI_ISR has completed with the
previous operation, then initialize the next operation and return.
****************************************************************************/
void TWI_Start_Transceiver( void )
{
  while ( TWI_Transceiver_Busy() ) {}             // Wait until TWI is ready for
next transmission.
  TWI_statusReg.all = 0;
  TWI_state         = TWI_NO_STATE ;
  TWCR = (1<<TWEN)|                             // TWI Interface enabled.
         (1<<TWIE)|(1<<TWINT)|                  // Enable TWI Interupt and clear the
flag.
         (1<<TWEA)|(0<<TWSTA)|(0<<TWSTO)|       // Prepare to ACK next time the
Slave is addressed.
         (0<<TWWC);                             //
  TWI_busy = 0;
}
/****************************************************************************
Call this function to read out the received data from the TWI transceiver buffer.
I.e. first call
TWI_Start_Transceiver to get the TWI Transceiver to fetch data. Then Run this
function to collect the
```

```
data when they have arrived. Include a pointer to where to place the data and the 
number of bytes
to fetch in the function call. The function will hold execution (loop) until the 
TWI_ISR has completed
with the previous operation, before reading out the data and returning.
If there was an error in the previous transmission the function will return the TWI 
State code.
****************************************************************************/
unsigned char TWI_Get_Data_From_Transceiver( unsigned char *msg, unsigned char 
msgSize )
{
  unsigned char i;

  while ( TWI_Transceiver_Busy() ) {}             // Wait until TWI is ready for 
next transmission.

  if( TWI_statusReg.lastTransOK )               // Last transmission completed 
successfully.
  {
    for ( i=0; i<msgSize; i++ )                 // Copy data from Transceiver 
buffer.
    {
      msg[ i ] = TWI_buf[ i ];
    }
    TWI_statusReg.RxDataInBuf = FALSE;          // Slave Receive data has been read 
from buffer.
  }
  return( TWI_statusReg.lastTransOK );
}


// ********** Interrupt Handlers ********** //
/****************************************************************************
This function is the Interrupt Service Routine (ISR), and called when the TWI 
interrupt is triggered;
that is whenever a TWI event has occurred. This function should not be called 
directly from the main
application.
****************************************************************************/
ISR(TWI_vect)
{
  static unsigned char TWI_bufPtr;

  switch (TWSR)
  {
    case TWI_STX_ADR_ACK:            // Own SLA+R has been received; ACK has been 
returned
//    case TWI_STX_ADR_ACK_M_ARB_LOST: // Arbitration lost in SLA+R/W as Master; own
```

```
SLA+R has been received; ACK has been returned
      TWI_bufPtr   = 0;                                 // Set buffer pointer to 
first data location
    case TWI_STX_DATA_ACK:           // Data byte in TWDR has been transmitted; ACK 
has been received
      TWDR = TWI_buf[TWI_bufPtr++];
      TWCR = (1<<TWEN)|                                 // TWI Interface enabled
             (1<<TWIE)|(1<<TWINT)|                      // Enable TWI Interupt and 
clear the flag to send byte
             (1<<TWEA)|(0<<TWSTA)|(0<<TWSTO)|           //
             (0<<TWWC);                                 //
      TWI_busy = 1;
      break;
    case TWI_STX_DATA_NACK:          // Data byte in TWDR has been transmitted; NACK 
has been received.
                                     // I.e. this could be the end of the 
transmission.
      if (TWI_bufPtr == TWI_msgSize) // Have we transceived all expected data?
      {
        TWI_statusReg.lastTransOK = TRUE;               // Set status bits to 
completed successfully.
      }
      else                          // Master has sent a NACK before all data where 
sent.
      {
        TWI_state = TWSR;                               // Store TWI State as 
errormessage.
      }

      TWCR = (1<<TWEN)|                                 // Enable TWI-interface and 
release TWI pins
             (1<<TWIE)|(1<<TWINT)|                      // Keep interrupt enabled 
and clear the flag
             (1<<TWEA)|(0<<TWSTA)|(0<<TWSTO)|           // Answer on next address 
match
             (0<<TWWC);                                 //

      TWI_busy = 0;   // Transmit is finished, we are not busy anymore
      break;
    case TWI_SRX_GEN_ACK:            // General call address has been received; ACK 
has been returned
//    case TWI_SRX_GEN_ACK_M_ARB_LOST: // Arbitration lost in SLA+R/W as Master; 
General call address has been received; ACK has been returned
      TWI_statusReg.genAddressCall = TRUE;
    case TWI_SRX_ADR_ACK:            // Own SLA+W has been received ACK has been 
returned
//    case TWI_SRX_ADR_ACK_M_ARB_LOST: // Arbitration lost in SLA+R/W as Master; own 
SLA+W has been received; ACK has been returned
```

```
                                                                // Dont need to clear 
TWI_S_statusRegister.generalAddressCall due to that it is the default state.
      TWI_statusReg.RxDataInBuf = TRUE;
      TWI_bufPtr   = 0;                                         // Set buffer pointer to 
first data location
      
                                                                // Reset the TWI Interupt to 
wait for a new event.
      TWCR = (1<<TWEN)|                                         // TWI Interface enabled
             (1<<TWIE)|(1<<TWINT)|                              // Enable TWI Interupt and 
clear the flag to send byte
             (1<<TWEA)|(0<<TWSTA)|(0<<TWSTO)|                   // Expect ACK on this 
transmission
             (0<<TWWC);
      TWI_busy = 1;
      
      break;
    case TWI_SRX_ADR_DATA_ACK:       // Previously addressed with own SLA+W; data 
has been received; ACK has been returned
    case TWI_SRX_GEN_DATA_ACK:       // Previously addressed with general call; data 
has been received; ACK has been returned
      TWI_buf[TWI_bufPtr++]     = TWDR;
      TWI_statusReg.lastTransOK = TRUE;                         // Set flag transmission 
successfull.
                                                                // Reset the TWI Interupt to 
wait for a new event.
      TWCR = (1<<TWEN)|                                         // TWI Interface enabled
             (1<<TWIE)|(1<<TWINT)|                              // Enable TWI Interupt and 
clear the flag to send byte
             (1<<TWEA)|(0<<TWSTA)|(0<<TWSTO)|                   // Send ACK after next 
reception
             (0<<TWWC);                                         // 
      TWI_busy = 1;
      break;
    case TWI_SRX_STOP_RESTART:       // A STOP condition or repeated START condition 
has been received while still addressed as Slave
                                                                // Enter not addressed mode 
and listen to address match
      TWCR = (1<<TWEN)|                                         // Enable TWI-interface and 
release TWI pins
             (1<<TWIE)|(1<<TWINT)|                              // Enable interrupt and 
clear the flag
             (1<<TWEA)|(0<<TWSTA)|(0<<TWSTO)|                   // Wait for new address 
match
             (0<<TWWC);                                         //
      
      TWI_busy = 0;  // We are waiting for a new address match, so we are not busy
      break;
    case TWI_SRX_ADR_DATA_NACK:      // Previously addressed with own SLA+W; data
has been received; NOT ACK has been returned
    case TWI_SRX_GEN_DATA_NACK:      // Previously addressed with general call; data
has been received; NOT ACK has been returned
    case TWI_STX_DATA_ACK_LAST_BYTE: // Last data byte in TWDR has been transmitted
(TWEA = “0”); ACK has been received
//    case TWI_NO_STATE              // No relevant state information available;
TWINT = “0”
    case TWI_BUS_ERROR:         // Bus error due to an illegal START or STOP
condition
      TWI_state = TWSR;                 //Store TWI State as errormessage, operation
also clears noErrors bit
      TWCR =   (1<<TWSTO)|(1<<TWINT);   //Recover from TWI_BUS_ERROR, this will
release the SDA and SCL pins thus enabling other devices to use the bus
      break;
    default:
      TWI_state = TWSR;                                 // Store TWI State as
errormessage, operation also clears the Success bit.
      TWCR = (1<<TWEN)|                                 // Enable TWI-interface and
release TWI pins
             (1<<TWIE)|(1<<TWINT)|                      // Keep interrupt enabled
and clear the flag
             (1<<TWEA)|(0<<TWSTA)|(0<<TWSTO)|           // Acknowledge on any new
requests.
             (0<<TWWC);                                 //

      TWI_busy = 0; // Unknown status, so we wait for a new address match that might
be something we can handle
  }
}
```