```
/*
 *     SocialLedge.com - Copyright (C) 2013
 *
 *     This file is part of free software framework for embedded processors.
 *     You can use it and/or distribute it as long as this copyright header
 *     remains unmodified.  The code is free for personal use and requires
 *     permission to use in a commercial product.
 *
 *      THIS SOFTWARE IS PROVIDED "AS IS".  NO WARRANTIES, WHETHER EXPRESS,
  IMPLIED
 *      OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF
 *      MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS
  SOFTWARE.
 *      I SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL,
  OR
 *      CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
 *
 *     You can reach the author of this software at :
 *          p r e e t . w i k i @ g m a i l . c o m
 */

#include <string.h>         // memcpy

#include "i2c_base.hpp"
#include "lpc_sys.h"
#include "printf_lib.h"

/**
 * Instead of using a dedicated variable for read vs. write, we just use the
  LSB of
 * the user address to indicate read or write mode.
 */
#define I2C_SET_READ_MODE(addr)     (addr |= 1)     ///< Set the LSB to
 indicate read-mode
#define I2C_SET_WRITE_MODE(addr)    (addr &= 0xFE)  ///< Reset the LSB to
 indicate write-mode
#define I2C_CHECK_READ_MODE(addr)         (addr & 1)      ///< Read address is
 ODD
#define I2C_WRITE_ADDR(addr)        (addr & 0xFE)   ///< Write address is EVEN
#define I2C_READ_ADDR(addr)         (addr | 1)      ///< Read address is ODD

void I2C_Base::handleInterrupt()
{
  u0_dbg_printf("I2C Interrupt\n");
    /* If transfer finished (not busy), then give the signal */
    if (busy != i2cStateMachine())
    {
        long higherPriorityTaskWaiting = 0;
        xSemaphoreGiveFromISR(mTransferCompleteSignal,
         &higherPriorityTaskWaiting);
        portEND_SWITCHING_ISR(higherPriorityTaskWaiting);
    }
}

uint8_t I2C_Base::readReg(uint8_t deviceAddress, uint8_t registerAddress)
{
    uint8_t register_return = 0;
    writeRegisterThenRead(deviceAddress, &registerAddress, 1, &register_return,
     1);
    return register_return;
}
bool I2C_Base::writeReg(uint8_t deviceAddress, uint8_t registerAddress, uint8_t
 value)
{
    uint8_t writeBytes[2];
    writeBytes[0] = registerAddress;
    writeBytes[1] = value;
    //u0_dbg_printf("0x%X-0x%X\n", writeBytes[0], writeBytes[1]);
    return writeRegisters(deviceAddress, writeBytes, 2);
}

bool I2C_Base::writeRegisterThenRead(uint8_t address, uint8_t * wdata, uint32_t
 wlength, uint8_t * rdata, uint32_t rlength)
{
    I2C_SET_READ_MODE(address);
    return transfer(address, wdata, wlength, rdata, rlength);
}

bool I2C_Base::writeRegisters(uint8_t deviceAddress, uint8_t firstReg, uint8_t*
 pData, uint32_t transferSize)
{

    transferSize = (transferSize > 254) ? 254 : transferSize;
    memcpy(writeBuffer, &pData[1], transferSize);
    return writeRegisters(deviceAddress, writeBuffer, transferSize+1);
}
bool I2C_Base::writeRegisters(uint8_t address, uint8_t * wdata, uint32_t
 wlength)
{
    I2C_SET_WRITE_MODE(address);
    return transfer(address, wdata, wlength, NULL, 0);
}
bool I2C_Base::readRegisters(uint8_t deviceAddress, uint8_t firstReg, uint8_t*
 pData, uint32_t transferSize)
{
    return writeRegisterThenRead(deviceAddress, &firstReg, 1, pData,
     transferSize);
}
bool I2C_Base::readRegisters(uint8_t address, uint8_t * rdata, uint32_t
 rlength)
{
    I2C_SET_READ_MODE(address);
    return transfer(address, NULL, 0, rdata, rlength);
}

bool I2C_Base::transfer(uint8_t address, uint8_t * wdata, uint32_t wlength,
 uint8_t * rdata, uint32_t rlength)
{
    bool status = false;
    if (mDisableOperation || (wdata == NULL && rdata == NULL))
    {
        return status;
    }

    // If scheduler not running, perform polling transaction
    if (taskSCHEDULER_RUNNING != xTaskGetSchedulerState())
    {
        i2cKickOffTransfer(address, wdata, wlength, rdata, rlength);

        // Wait for transfer to finish
        const uint64_t timeout = sys_get_uptime_ms() + I2C_TIMEOUT_MS;
        while (!xSemaphoreTake(mTransferCompleteSignal, 0))
        {
            if (sys_get_uptime_ms() > timeout)
            {
                break;
            }
        }

        status = (0 == mTransaction.error);
    }
    else if (xSemaphoreTake(mI2CMutex, OS_MS(I2C_TIMEOUT_MS)))
    {
        // Clear potential stale signal and start the transfer
        xSemaphoreTake(mTransferCompleteSignal, 0);
        i2cKickOffTransfer(address, wdata, wlength, rdata, rlength);

        // Wait for transfer to finish and copy the data if it was read mode
        if (xSemaphoreTake(mTransferCompleteSignal, OS_MS(I2C_TIMEOUT_MS)))
        {
            status = (0 == mTransaction.error);
        }

        xSemaphoreGive(mI2CMutex);
    }

    return status;
}

bool I2C_Base::checkDeviceResponse(uint8_t address)
{
    uint8_t notUsed = 0;
    // The I2C State machine will not continue after 1st state when length is
     set to 0
    uint32_t lenZeroToTestDeviceReady = 0;

    return readRegisters(address, &notUsed, lenZeroToTestDeviceReady);
}

I2C_Base::I2C_Base(LPC_I2C_TypeDef* pI2CBaseAddr):
    mpI2CRegs(pI2CBaseAddr),
    mDisableOperation(false)
{
    mI2CMutex = xSemaphoreCreateMutex();
    mTransferCompleteSignal = xSemaphoreCreateBinary();

    // Optional: Provide names of the FreeRTOS objects for the Trace Facility
    vTraceSetMutexName(mI2CMutex, "I2C Mutex");
    vTraceSetSemaphoreName(mTransferCompleteSignal, "I2C Finish Sem");

    switch((unsigned int)mpI2CRegs)
    {
        case LPC_I2C0_BASE:
            mIRQ = I2C0_IRQn;
            break;
        case LPC_I2C1_BASE:
            mIRQ = I2C1_IRQn;
            break;
        case LPC_I2C2_BASE:
            mIRQ = I2C2_IRQn;
            break;
        default:
            mIRQ = (IRQn_Type)(99); // Using invalid IRQ on purpose
            break;
    }
}

bool I2C_Base::init(uint32_t pclk, uint32_t busRateInKhz)
{
    // Power on I2C
    switch (mIRQ)
    {
        case I2C0_IRQn: lpc_pconp(pconp_i2c0, true);  break;
        case I2C1_IRQn: lpc_pconp(pconp_i2c1, true);  break;
        case I2C2_IRQn: lpc_pconp(pconp_i2c2, true);  break;
        default: return false;
    }

    mpI2CRegs->I2CONCLR = 0x6C;           // Clear ALL I2C Flags

    /**
     * Per I2C high speed mode:
```

```
     * HS mode master devices generate a serial clock signal with a HIGH to LOW
      ratio of 1 to 2.
     * So to be able to optimize speed, we use different duty cycle for high/
      low
     *
     * Compute the I2C clock dividers.
     * The LOW period can be longer than the HIGH period because the rise time
     * of SDA/SCL is an RC curve, whereas the fall time is a sharper curve.
     */
    const uint32_t percent_high = 40;
    const uint32_t percent_low = (100 - percent_high);
    const uint32_t freq_hz = (busRateInKhz > 1000) ? (100 * 1000) :
     (busRateInKhz * 1000);
    const uint32_t half_clock_divider = (pclk / freq_hz) / 2;
    mpI2CRegs->I2SCLH = (half_clock_divider * percent_high) / 100;
    mpI2CRegs->I2SCLL = (half_clock_divider * percent_low ) / 100;

    // Set I2C slave address and enable I2C
    mpI2CRegs->I2ADR0 = 0;
    mpI2CRegs->I2ADR1 = 0;
    mpI2CRegs->I2ADR2 = 0;
    mpI2CRegs->I2ADR3 = 0;

    // Enable I2C and the interrupt for it
    mpI2CRegs->I2CONSET = 0x40;
    vTraceSetISRProperties(mIRQ, "I2C", IP_i2c);
    NVIC_EnableIRQ(mIRQ);

    return true;
}

/// Private ///

void I2C_Base::i2cKickOffTransfer(uint8_t addr, uint8_t * wbytes, uint32_t
 wlength, uint8_t * rbytes, uint32_t rlength)
{
    mTransaction.slaveAddr      = addr;
    mTransaction.error          = 0;
    mTransaction.dataWrite      = wbytes;
    mTransaction.writeLength    = wlength;
    mTransaction.dataRead       = rbytes;
    mTransaction.readLength     = rlength;
    // Send START, I2C State Machine will finish the rest.
    mpI2CRegs->I2CONSET = 0x20;
}

inline void I2C_Base::clearSIFlag()
{
    mpI2CRegs->I2CONCLR = (1 << 3);
}
inline void I2C_Base::setSTARTFlag()
```

```
{
    mpI2CRegs->I2CONSET = (1 << 5);
}
inline void I2C_Base::clearSTARTFlag()
{
    mpI2CRegs->I2CONCLR = (1 << 5);
}
inline void I2C_Base::setAckFlag()
{
    mpI2CRegs->I2CONSET = (1 << 2);
}
inline void I2C_Base::setNackFlag()
{
    mpI2CRegs->I2CONCLR = (1 << 2);
}
inline void I2C_Base::setStop()
{
    clearSTARTFlag();
    mpI2CRegs->I2CONSET = (1 << 4);
    clearSIFlag();
    while ((mpI2CRegs->I2CONSET & (1 << 4)));
    if (I2C_CHECK_READ_MODE(mTransaction.slaveAddr))
    {
        state = readComplete;
    }
    else
    {
        state = writeComplete;
    }
}

/*
 * I2CONSET bits
 * 0x04 AA
 * 0x08 SI
 * 0x10 STOP
 * 0x20 START
 * 0x40 ENABLE
 *
 * I2CONCLR bits
 * 0x04 AA
 * 0x08 SI
 * 0x20 START
 * 0x40 ENABLE
 */
__attribute__ ((weak)) I2C_Base::mStateMachineStatus_t
 I2C_Base::i2cStateMachine()
{
    enum
    {
        // General states :
```

```
    busError        = 0x00,
    start           = 0x08,
    repeatStart     = 0x10,
    arbitrationLost = 0x38,

    // Master Transmitter States:
    slaveAddressAcked  = 0x18,
    slaveAddressNacked = 0x20,
    dataAckedBySlave   = 0x28,
    dataNackedBySlave  = 0x30,

    // Master Receiver States:
    readAckedBySlave      = 0x40,
    readModeNackedBySlave = 0x48,
    dataAvailableAckSent  = 0x50,
    dataAvailableNackSent = 0x58,

// Slave Receiver States:
slaveWriteAddressed = 0x60,
slaveWriteGetData  = 0x80,
slaveWriteStopOrRepeat = 0xA0,

// Slave Transmitter States:
slaveReadAddressed = 0xA8,
slaveReadContinue = 0xB8,
slaveReadFinished = 0xC0,
};

static bool register_read = false;

state = busy;

/*

  *************************************************************************
  **********************************
 * Write-mode state transition :
 * start --> slaveAddressAcked --> dataAckedBySlave --> ...
  (dataAckedBySlave) --> (stop)
 *
 * Read-mode state transition :
 * start --> slaveAddressAcked --> dataAcked --> repeatStart -->
  readAckedBySlave
 *  For 2+ bytes:  dataAvailableAckSent --> ... (dataAvailableAckSent) -->
  dataAvailableNackSent --> (stop)
 *  For 1  byte :  dataAvailableNackSent --> (stop)

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

```
switch (mpI2CRegs->I2STAT)
{
  // 0x60
case slaveWriteAddressed:
  u0_dbg_printf("0x60: write\n");
  setAckFlag();
  clearSIFlag();
  break;

  // 0x80
case slaveWriteGetData:
  u0_dbg_printf("0x80: reading value\n");
  if (!register_read)
{
  slave_reg = mpI2CRegs->I2DAT;
  u0_dbg_printf("0x80: Writing to offset %d\n", slave_reg);
  register_read = true;
}
  else
{
  *(slave_buffer + slave_reg + offset) = mpI2CRegs->I2DAT;
  u0_dbg_printf("0x80: Wrote %d to offset %d\n", *(slave_buffer + slave_reg
   + offset),  slave_reg);
  offset++;
}
  setAckFlag();
  clearSIFlag();
  break;

  // 0xA0
case slaveWriteStopOrRepeat:
  u0_dbg_printf("0xA0: done or repeating\n");
  setAckFlag();
  clearSIFlag();
  register_read = false;
  offset = 0;
  state = writeComplete;
  break;

  // 0xA8
case slaveReadAddressed:
  u0_dbg_printf("0xA8: Going to send data\n");
  mpI2CRegs->I2DAT = *(slave_buffer + slave_reg + offset);
  offset++;
  offset %= 255;
  setAckFlag();
  clearSIFlag();
  break;

  // 0xB8
case slaveReadContinue:
  u0_dbg_printf("0xB8: Continue to send data\n");
  mpI2CRegs->I2DAT = *(slave_buffer + slave_reg + offset);
  offset++;
  offset %= 255;
  setAckFlag();
  clearSIFlag();
  break;

  // 0xC0
case slaveReadFinished:
  u0_dbg_printf("0xC0: Read Done\n");
  setAckFlag();
  clearSIFlag();
  offset = 0;
  state = readComplete;
  break;

    case start:
        //u0_dbg_printf("sta\n");
        if(mTransaction.writeLength == 0)
        {
            //u0_dbg_printf("ard-%X\n",
             I2C_READ_ADDR(mTransaction.slaveAddr));
            mpI2CRegs->I2DAT = I2C_READ_ADDR(mTransaction.slaveAddr);
        }
        else
        {
            //u0_dbg_printf("awr-%X\n",
             I2C_WRITE_ADDR(mTransaction.slaveAddr));
            mpI2CRegs->I2DAT = I2C_WRITE_ADDR(mTransaction.slaveAddr);
        }
        clearSIFlag();
        break;
    case repeatStart:
        //u0_dbg_printf("rsta-%X\n",
         I2C_READ_ADDR(mTransaction.slaveAddr));
        mpI2CRegs->I2DAT = I2C_READ_ADDR(mTransaction.slaveAddr);
        clearSIFlag();
        break;

    case slaveAddressAcked:
        //u0_dbg_printf("sack\n");
        clearSTARTFlag();
        // No data to transfer, this is used just to test if the slave
         responds
        if (0 == mTransaction.readLength && 0 == mTransaction.writeLength)
        {
            //u0_dbg_printf("sto\n");
            setStop();
        }
        else if(0 != mTransaction.writeLength)
```

```
        {
            mpI2CRegs->I2DAT = *(mTransaction.dataWrite);
            ++mTransaction.dataWrite;
            --mTransaction.writeLength;
            clearSIFlag();

            //u0_dbg_printf("1st-%X\n", mpI2CRegs->I2DAT);
        }
        break;

    case dataAckedBySlave:
        if (0 == mTransaction.writeLength)
        {
            //u0_dbg_printf("wend\n");
            if (I2C_CHECK_READ_MODE(mTransaction.slaveAddr))
            {
                //u0_dbg_printf("rread\n");
                setSTARTFlag(); // Send Repeat-start for read-mode
                clearSIFlag();
            }
            else
            {
                //u0_dbg_printf("sto\n");
                setStop();
            }
        }
        else
        {
            mpI2CRegs->I2DAT = *(mTransaction.dataWrite);
            ++mTransaction.dataWrite;
            --mTransaction.writeLength;
            clearSIFlag();
            //u0_dbg_printf("wlo-%X\n", mpI2CRegs->I2DAT);
        }
        break;

    /* In this state, we are about to initiate the transfer of data from
     slave to us
     * so we are just setting the ACK or NACK that we'll do AFTER the byte
      is received.
     */
    case readAckedBySlave:
        clearSTARTFlag();
        if (mTransaction.readLength > 1)
        {
            //u0_dbg_printf("ack\n");
            setAckFlag();  // 1+ bytes: Send ACK to receive a byte and
             transition to dataAvailableAckSent
        }
        else
        {
            //u0_dbg_printf("nack\n");
            setNackFlag();  //  1 byte : NACK next byte to go to
             dataAvailableNackSent for 1-byte read.
        }
        clearSIFlag();
        break;
    case dataAvailableAckSent:
        *mTransaction.dataRead = mpI2CRegs->I2DAT;
        ++mTransaction.dataRead;
        --mTransaction.readLength;

        //u0_dbg_printf("rdat-%X\n", mpI2CRegs->I2DAT);

        if (1 == mTransaction.readLength)  // Only 1 more byte remaining
        {
            //u0_dbg_printf("nack\n");
            setNackFlag();// NACK next byte --> Next state:
             dataAvailableNackSent
        }
        else
        {
            //u0_dbg_printf("ack\n");
            setAckFlag(); // ACK next byte --> Next state:
             dataAvailableAckSent(back to this state)
        }

        clearSIFlag();
        break;
    case dataAvailableNackSent: // Read last-byte from Slave
        //u0_dbg_printf("nack-sto-%X\n", mpI2CRegs->I2DAT);
        *mTransaction.dataRead = mpI2CRegs->I2DAT;
        setStop();
        break;

    case arbitrationLost:
        // We should not issue stop() in this condition, but we still need
         to end our  transaction.
        state = I2C_CHECK_READ_MODE(mTransaction.slaveAddr) ?
         readComplete : writeComplete;
        mTransaction.error = mpI2CRegs->I2STAT;
        break;

    case slaveAddressNacked:    // no break
    case dataNackedBySlave:     // no break
    case readModeNackedBySlave: // no break
    case busError:              // no break
    default:
        mTransaction.error = mpI2CRegs->I2STAT;
        setStop();
        break;
}
return state;
}
```