```
/*
 *     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
 */

/**
 * @file  i2c_base.hpp
 * @brief Provides I2C Base class functionality for I2C peripherals
 *
 * 20140212 : Improved the driver by not having internal memory to copy the
 *            transaction's data.  The buffer supplied from the user is used
  directly.
 * 20131211 : Used timeout for read/write semaphore (instead of portMAX_DELAY)
 *              Refactored code, and made the write transfer wait for
  completion
 *              and return true upon success.
 */
#ifndef I2C_BASE_HPP_
#define I2C_BASE_HPP_

#include <stdint.h>

#include "FreeRTOS.h"
#include "task.h"       // xTaskGetSchedulerState()
#include "semphr.h"     // Semaphores used in I2C
#include "LPC17xx.h"



/**
 * Define the maximum timeout for r/w operation (in case error occurs)
 * This is the timeout for read transaction to finish and if FreeRTOS is
  running,
 * then this is the timeout for the mutex to be obtained.
 */
```

```
#define I2C_TIMEOUT_MS          1000


/**
 * I2C Base class that can be used to write drivers for all I2C peripherals.
 *  Steps needed to write a I2C driver:
 *  - Inherit this class
 *  - Call init() and configure PINSEL to select your I2C pins
 *  - When your I2C(#) hardware interrupt occurs, call handleInterrupt()
 *
 *  To connect I2C Interrupt with your I2C, reference this example:
 *  @code
 *  extern "C"
 *   {
 *      void I2C0_IRQHandler()
 *      {
 *          I2C0::getInstance().handleInterrupt();
 *      }
 *   }
 *  @endcode
 * @ingroup Drivers
 */
class I2C_Base
{
public:
  uint8_t slave_reg;
  volatile uint8_t* slave_buffer;
  uint32_t slave_buffer_size;
  uint8_t offset;

  /**
   * When the I2C interrupt occurs, this function should be called to handle
   * future action to take due to the interrupt cause.
   */
  void handleInterrupt();

  /**
   * Reads a single byte from an I2C Slave
   * @param deviceAddress     The I2C Device Address
   * @param registerAddress   The register address to read
   * @return The byte read from slave device (might be 0 if error)
   */
  uint8_t readReg(uint8_t deviceAddress, uint8_t registerAddress);

  /**
   * Writes a single byte to an I2C Slave
   * @param deviceAddress     The I2C Device Address
   * @param registerAddress   The register address to write
   * @param value             The value to write to registerAddress
   * @return true if successful
   */
```

```
  bool writeReg(uint8_t deviceAddress, uint8_t registerAddress, uint8_t value);

  bool writeRegisterThenRead(uint8_t address, uint8_t * wdata, uint32_t
   wlength, uint8_t * rdata, uint32_t rlength);

  /// @copydoc transfer()
  bool readRegisters(uint8_t deviceAddress, uint8_t firstReg, uint8_t* pData,
   uint32_t transferSize);
  bool readRegisters(uint8_t address, uint8_t * rdata, uint32_t rlength);

  /// @copydoc transfer()
  bool writeRegisters(uint8_t deviceAddress, uint8_t firstReg, uint8_t* pData,
   uint32_t transferSize);

  bool writeRegisters(uint8_t address, uint8_t * wdata, uint32_t wlength);


  /**
   * This function can be used to check if an I2C device responds to its
    address,
   * which can therefore be used to discover all I2C hardware devices.
   * Sometimes this method is used by devices to check if they are ready for
    further
   * operations such as an EEPROM or FLASH memory.
   *
   * @param deviceAddress     The device address to check for I2C response
   * @returns true if I2C device with given address is ready
   */
  bool checkDeviceResponse(uint8_t deviceAddress);



protected:
  uint8_t writeBuffer[256];
  /**
   * Protected constructor that requires parent class to provide I2C
   * base register address for which to operate this I2C driver
   */
  I2C_Base(LPC_I2C_TypeDef* pI2CBaseAddr);

  /**
   * Initializes I2C Communication BUS
   * @param pclk  The peripheral clock to the I2C Bus
   * @param busRateInKhz  The speed to set for this I2C Bus
   */
  bool init(uint32_t pclk, uint32_t busRateInKhz);

  /**
   * Disables I2C operation
   * This can be used to disable all I2C operations in case of severe I2C Bus
    Failure
```

```
   * @warning Once disabled, I2C cannot be enabled again
   */
  void disableOperation() { mDisableOperation = true; }


private:
  LPC_I2C_TypeDef* mpI2CRegs;    ///< Pointer to I2C memory map
  IRQn_Type        mIRQ;         ///< IRQ of this I2C
  bool mDisableOperation;        ///< Tracks if I2C is disabled by
   disableOperation()
  SemaphoreHandle_t mI2CMutex;   ///< I2C Mutex used when FreeRTOS is running
  SemaphoreHandle_t mTransferCompleteSignal; ///< Signal that indicates read is
   complete
  bool first_byte = false;

  /**
   * The status of I2C is returned from the I2C function that handles state
    machine
   */
  typedef enum {
    busy,
    readComplete,
    writeComplete
  } __attribute__((packed)) mStateMachineStatus_t;

  /**
   * This structure contains I2C transaction parameters
   */
  typedef struct
  {
    uint8_t         slaveAddr;      ///< Slave Device Address
    uint8_t         error;          ///< Error if any occurred within I2C
    uint8_t         *dataWrite;     ///< Buffer of the I2C Write
    uint32_t        writeLength;    ///< # of bytes to write
    uint8_t         *dataRead;      ///< Buffer of the I2C Read
    uint32_t        readLength;     ///< # of bytes to read
  } mI2CTransaction_t;

  /// The I2C Input Output frame that contains I2C transaction information
  mI2CTransaction_t mTransaction;

  /**
   * When an interrupt occurs, this handles the I2C State Machine action
   * @returns The status of I2C State Machine, which are:
   *              - Busy
   *              - Write is complete
   *              - Read  is complete
   */
  mStateMachineStatus_t i2cStateMachine();
  mStateMachineStatus_t state;
```

```
  void clearSIFlag();
  void setSTARTFlag();
  void clearSTARTFlag();
  void setAckFlag();
  void setNackFlag();
  void setStop();

  /**
   * Read/writes multiple bytes to an I2C device starting from the first
    register
   * It is assumed that like almost all I2C devices, the register address
    increments by 1
   * upon writing each byte.  This is usually how all I2C devices work.
   * @param deviceAddress     The device address to read/write data from/to
    (odd=read, even=write)
   * @param firstReg          The first register to read/write from/to
   * @param pData             The pointer to copy/write data from/to
   * @param transferSize      The number of bytes to read/write
   * @returns true if the transfer was successful
   */
  // bool transfer(uint8_t deviceAddress, uint8_t firstReg, uint8_t* pData,
   uint32_t transferSize);
  bool transfer(uint8_t address, uint8_t * wdata, uint32_t wlength, uint8_t *
   rdata, uint32_t rlength);

  /**
   * This is the entry point for an I2C transaction
   * @param devAddr   The address of the I2C Device
   * @param regStart  The register address of I2C device to read or write
   * @param pBytes    The pointer to one or more data bytes to read or write
   * @param len       The length of the I2C transaction
   */
  // void i2cKickOffTransfer(uint8_t devAddr, uint8_t regStart, uint8_t*
   pBytes, uint32_t len);
  void i2cKickOffTransfer(uint8_t addr, uint8_t * wbytes, uint32_t wlength,
   uint8_t * rbytes, uint32_t rlength);
};



#endif /* I2C_BASE_HPP_ */
```