```
//I2c LAB
class lab5: public scheduler_task {
public:
  lab5() :
    scheduler_task("lab5", 2000, PRIORITY_LOW) {
    printf("lab5 task started");
  }

  bool init(void) {
    i2c.slave(slaveadder, buffer, sizeof(buffer));

    return true;
  }

  bool run(void *p) {
    //      I2C2& i2c = I2C2::getInstance(); // Get I2C driver instance
    //      const uint8_t slaveadder = 0x60;  // Pick any address other than the used used at
i2c2.hpp
    //      uint8_t buffer[256] = { 0 };     // Our slave read/write buffer
    uint8_t prev = buffer[0];
    while (1) {
      if (prev != buffer[0]) {
        prev = buffer[0];
        u0_dbg_printf("buffer[0] changed to %#x\n", buffer[0]);
      }
    }
    return true;
  }
private:
  I2C2& i2c = I2C2::getInstance(); // Get I2C driver instance
  const uint8_t slaveadder = 0x66; // Pick any address other than the used used at i2c2.hpp
  uint8_t buffer[256] = { 0 };
};

int main(void) {
    I2C2& i2c = I2C2::getInstance(); // Get I2C driver instance
    const uint8_t slaveAddr = 0xC0;  // Pick any address other than an existing one at i2c2.hpp
    volatile uint8_t buffer[256] = { 0 }; // Our slave read/write buffer (This is the memory your
other master board will read/write)

     I2C is already initialized before main(), so you will have to add initSlave() to i2c base class
for your slave driver
    i2c.init_slave(slaveAddr, &buffer[0], sizeof(buffer));
```

```
      I2C interrupt will (should) modify our buffer.
      So just monitor our buffer, and print and/or light up LEDs
      ie: If buffer[0] == 0, then LED ON, else LED OFF
        scheduler_add_task(new lab5());
        uint8_t prev = buffer[0];
        while(1)
        {
           if (prev != buffer[0]) {
              prev = buffer[0];
              printf("buffer[0] changed to %#x by the other Master Board\n", buffer[0]);
           }
     scheduler_add_task(new terminalTask(PRIORITY_HIGH));
     scheduler_start();
 }

/*
 *    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:
      /**
       * 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);

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

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

    bool slave(const uint8_t slaveadder, uint8_t *buffer, uint16_t size);

    /**
     * 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:
```

```
  /**
   * 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

  /**
   * 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
  {
    uint32_t trxSize;   ///< # of bytes to transfer.
    uint8_t slaveAddr;  ///< Slave Device Address
```

```
    uint8_t firstReg;   ///< 1st Register to Read or Write
    uint8_t error;      ///< Error if any occurred within I2C
    uint8_t *pMasterData;  ///< Buffer of the I2C Read or Write
    uint8_t *pslavedata;
    uint8_t *start;
    uint8_t data;
    uint8_t indexer;
    bool ireg;

} 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();

/**
 * 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);

/**
 * 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);
};



#endif /* I2C_BASE_HPP_ */

/*
 *     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"

/**
 * 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_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() {
   /* 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 byte = 0;
    readRegisters(deviceAddress, registerAddress, &byte, 1);
    return byte;
}

bool I2C_Base::readRegisters(uint8_t deviceAddress, uint8_t firstReg,
        uint8_t* pData, uint32_t bytesToRead) {
    I2C_SET_READ_MODE(deviceAddress);
    return transfer(deviceAddress, firstReg, pData, bytesToRead);
}

bool I2C_Base::writeReg(uint8_t deviceAddress, uint8_t registerAddress,
        uint8_t value) {
    return writeRegisters(deviceAddress, registerAddress, &value, 1);
}

bool I2C_Base::writeRegisters(uint8_t deviceAddress, uint8_t firstReg,
        uint8_t* pData, uint32_t bytesToWrite) {
    I2C_SET_WRITE_MODE(deviceAddress);
    return transfer(deviceAddress, firstReg, pData, bytesToWrite);
}

bool I2C_Base::transfer(uint8_t deviceAddress, uint8_t firstReg, uint8_t* pData,
        uint32_t transferSize) {
    bool status = false;
    if (mDisableOperation || !pData) {
        return status;
    }

    // If scheduler not running, perform polling transaction
    if (taskSCHEDULER_RUNNING != xTaskGetSchedulerState()) {
        i2cKickOffTransfer(deviceAddress, firstReg, pData, transferSize);

        // 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(deviceAddress, firstReg, pData, transferSize);

        // 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 deviceAddress) {
    uint8_t dummyReg = 0;
    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(deviceAddress, dummyReg, &notUsed,
            lenZeroToTestDeviceReady);
}

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

    /// Binary semaphore needs to be taken after creating it
    xSemaphoreTake(mTransferCompleteSignal, 0);

    if ((unsigned int) mpI2CRegs == LPC_I2C0_BASE) {
        mIRQ = I2C0_IRQn;
```

```
    } else if ((unsigned int) mpI2CRegs == LPC_I2C1_BASE) {
        mIRQ = I2C1_IRQn;
    } else if ((unsigned int) mpI2CRegs == LPC_I2C2_BASE) {
        mIRQ = I2C2_IRQn;
    } else {
        mIRQ = (IRQn_Type) 99; // Using invalid IRQ on purpose
    }
}

bool I2C_Base::init(uint32_t pclk, uint32_t busRateInKhz) {
    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;
    NVIC_EnableIRQ(mIRQ);

    return true;
}

/// Private ///

void I2C_Base::i2cKickOffTransfer(uint8_t devAddr, uint8_t regStart,
        uint8_t* pBytes, uint32_t len) {
    mTransaction.error = 0;
    mTransaction.slaveAddr = devAddr;
    mTransaction.firstReg = regStart;
    mTransaction.trxSize = len;
    mTransaction.pMasterData = pBytes;

    // Send START, I2C State Machine will finish the rest.
    mpI2CRegs->I2CONSET = 0x20;
}

/*
 * I2CONSET bits
 * 0x04 AA
 * 0x08 SI
 * 0x10 STOP
 * 0x20 START
 * 0x40 ENABLE
 *
 * I2CONCLR bits
 * 0x04 AA
 * 0x08 SI
 * 0x20 START
 * 0x40 ENABLE
 */
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:
    dataPrepareSlaveR = 0x60,
    lastByteAckedR = 0x80,
    lastByteNackedR = 0x88,
    startStopSlaveR = 0xA0,

    //Slave Transmitter State:
    dataPrepare2SlaveT = 0xA8,
    additionalData2SlaveT = 0xB8,
    lastByteNackedT = 0xC0,
    lastByteAckedT = 0xC8
  };

  mStateMachineStatus_t 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)
```

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

#define clearSIFlag()      mpI2CRegs->I2CONCLR = (1<<3)
#define setSTARTFlag()     mpI2CRegs->I2CONSET = (1<<5)
#define clearSTARTFlag()   mpI2CRegs->I2CONCLR = (1<<5)
#define setAckFlag()       mpI2CRegs->I2CONSET = (1<<2)
#define setNackFlag()      mpI2CRegs->I2CONCLR = (1<<2)

#define setStop()          clearSTARTFlag();                    \
      mpI2CRegs->I2CONSET = (1<<4);            \
      clearSIFlag();                        \
      while((mpI2CRegs->I2CONSET&(1<<4)));       \
      if(I2C_READ_MODE(mTransaction.slaveAddr))   \
      state = readComplete;                \
      else                                 \
      state = writeComplete;

   switch (mpI2CRegs->I2STAT) {
      case start:
         mpI2CRegs->I2DAT = I2C_WRITE_ADDR(mTransaction.slaveAddr);
         clearSIFlag();
         break;
      case repeatStart:
         mpI2CRegs->I2DAT = I2C_READ_ADDR(mTransaction.slaveAddr);
         clearSIFlag();
         break;

      case slaveAddressAcked:
         clearSTARTFlag();
         // No data to transfer, this is used just to test if the slave responds
         if (0 == mTransaction.trxSize) {
            setStop()
                  ;
         } else {
            mpI2CRegs->I2DAT = mTransaction.firstReg;
            clearSIFlag();
         }
         break;

      case dataAckedBySlave:
         if (I2C_READ_MODE(mTransaction.slaveAddr)) {
```

```
            setSTARTFlag(); // Send Repeat-start for read-mode
            clearSIFlag();
          } else {
            if (0 == mTransaction.trxSize) {
              setStop()
                    ;
            } else {
              mpI2CRegs->I2DAT = *(mTransaction.pMasterData);
              ++mTransaction.pMasterData;
              --mTransaction.trxSize;
              clearSIFlag();
            }
          }
          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.trxSize > 1) {
            setAckFlag(); // 1+ bytes: Send ACK to receive a byte and transition to
dataAvailableAckSent
          } else {
            setNackFlag(); //  1 byte : NACK next byte to go to dataAvailableNackSent for 1-byte
read.
          }
          clearSIFlag();
          break;
        case dataAvailableAckSent:
          *mTransaction.pMasterData = mpI2CRegs->I2DAT;
          ++mTransaction.pMasterData;
          --mTransaction.trxSize;

          if (1 == mTransaction.trxSize) { // Only 1 more byte remaining
            setNackFlag(); // NACK next byte --> Next state: dataAvailableNackSent
          } else {
            setAckFlag(); // ACK next byte --> Next state: dataAvailableAckSent(back to this state)
          }

          clearSIFlag();
          break;
        case dataAvailableNackSent: // Read last-byte from Slave
          *mTransaction.pMasterData = mpI2CRegs->I2DAT;
```

```
    setStop()
    ;
    break;

case arbitrationLost:
    // We should not issue stop() in this condition, but we still need to end our  transaction.
    state = I2C_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

case dataPrepareSlaveR: //0x60q
    mpI2CRegs->I2CONSET = 0x04;
    mpI2CRegs->I2CONCLR = 0x08;
    mTransaction.ireg = true;
    mTransaction.pslavedata = mTransaction.start;
    break;

case lastByteAckedR: //0x80
    if (mTransaction.ireg == true) {
        mTransaction.indexer = mpI2CRegs->I2DAT;
        mTransaction.pslavedata += mTransaction.indexer;
        mpI2CRegs->I2CONSET = 0x04;
        mpI2CRegs->I2CONCLR = 0x08;
        mTransaction.ireg = false;
        break;
    }
    *(mTransaction.pslavedata) = mpI2CRegs->I2DAT;
    mpI2CRegs->I2CONCLR = 0x0C;
    break;

case lastByteNackedR: //0x88
    mpI2CRegs->I2CONSET = 0x04;
    mpI2CRegs->I2CONCLR = 0x08;
    break;

case startStopSlaveR: //0xA0
    mpI2CRegs->I2CONSET = 0x04;
    mpI2CRegs->I2CONCLR = 0x08;
```

```
            break;

        case dataPrepare2SlaveT: //0xA8
            mpI2CRegs->I2DAT = *(mTransaction.pslavedata);
            mpI2CRegs->I2CONSET = 0x04;
            mpI2CRegs->I2CONCLR = 0x08;
            mTransaction.pslavedata++;
            break;

        case additionalData2SlaveT: //0xB8
            mpI2CRegs->I2DAT = *(mTransaction.pslavedata);
            mpI2CRegs->I2CONSET = 0x04;
            mpI2CRegs->I2CONCLR = 0x08;
            mTransaction.pslavedata++;
            break;

        case lastByteNackedT: //0xC0
            mpI2CRegs->I2CONSET = 0x04;
            mpI2CRegs->I2CONCLR = 0x08;
            break;

        case lastByteAckedT: //0xC8
            mpI2CRegs->I2CONSET = 0x04;
            mpI2CRegs->I2CONCLR = 0x08;
            break;

        default:
            mTransaction.error = mpI2CRegs->I2STAT;
            setStop()
            ;
            break;
    }

    return state;
}

bool I2C_Base::slave(const uint8_t slaveadder, uint8_t *buffer, uint16_t size) {
    LPC_SC->PCONP |= ~(1<<26);
    mpI2CRegs->I2ADR0 = slaveadder;
    mpI2CRegs->I2ADR1 = 0;
    mpI2CRegs->I2ADR2 = 0;
    mpI2CRegs->I2ADR3 = 0;
    mpI2CRegs->I2MASK0 = 0;
    mpI2CRegs->I2MASK1 = 0;
```

```
    mpI2CRegs->I2MASK2 = 0;
    mpI2CRegs->I2MASK3 = 0;
    mpI2CRegs->I2CONSET = 0x44;
    //NVIC_EnableIRQ(mIRQ);
    mTransaction.pslavedata = buffer;
    mTransaction.start = buffer;
    mTransaction.data = size;
    LPC_PINCON->PINSEL0 &= ~(0xF << 20);
    LPC_PINCON->PINSEL0 |=  (0xA << 20);
    return true;
}
```