Abhilash Munnangi
Vivek Naveen
I2C Lab

Part 3a Code, Explanations, and Screenshots:

In part 3a of the lab, we wrote a function to initialize a slave at a new address. In the function, we simply pass in the address of the new slave to the ADR0 register, then the CONSET register to 0x44 to enable the slave. The screenshot below shows that 0xFE has been recognized as a slave (address passed in). It is important to note that the least significant bit of the ADR0 is for general call enable, and therefore the address must be even.

```
I2C:Starting transfer with device address: 0xFC
I2C:  HW State: 0x08
I2C:  HW State: 0x20
I2C:  Finished with state: 0x20
I2C:Starting transfer with device address: 0xFE
I2C:  HW State: 0x08
I2C:  HW State: 0x18
I2C:  Finished with state: 0x00
I2C slave detected at address: 0xFE
I2C:Starting transfer with device address: 0x38
I2C:  HW State: 0x08
I2C:  HW State: 0x18
I2C:  HW State: 0x28
I2C:  HW State: 0x28
I2C:  Finished with state: 0x00
I2C:Starting transfer with device address: 0x39
I2C:  HW State: 0x08
I2C:  HW State: 0x18
I2C:  HW State: 0x28
I2C:  HW State: 0x10
I2C:  HW State: 0x40
I2C:  HW State: 0x58
I2C:  Finished with state: 0x00

entry_point(): Entering main()
Starting RTOS
```

i2c_slave_init.c

```
#include "i2c_slave_init.h"
#include "FreeRTOS.h"
#include "semphr.h"
#include "task.h"

#include "common_macros.h"
```

```
#include "lpc40xx.h"
#include "lpc_peripherals.h"
void i2c2__slave_init(uint8_t slave_address_to_respond_to) {
  LPC_I2C2->ADR0 |= slave_address_to_respond_to;
  LPC_I2C2->CONSET |= 0x44;
  fprintf(stderr, "ADR0: %d\n", LPC_I2C2->ADR0);
}
```

Part 3b, 3c, Code, Screenshots, and Explanation:

In part 3b, we implemented the state machine that was described in detail in part 2. Then, the write and read callback were written in the main to store the data. In this part, after hours of debugging, we were able to successfully write data to the slave. This is shown in the screenshot below. However, when we try to read the data back. We are getting a FreeRTOS assert. This is an interesting issue as no Interrupts were altered. We are still in the process of debugging this issue. Below is the code for the working write and the in progress read functionalities. One board is used as a master and the other is used as a slave, and therefore one will be flashed after the other. Then, the master and the slave interact with each other to send and receive data. After some debugging, we believe that the issue is in either state 0x58 (since the print is not reached), or the corresponding slave state. We are still trying to solve the issues.

```
i2c write 0xFE 0x00 10
I2C:Starting transfer with device address: 0xFE
I2C:  HW State: 0x08
I2C:  HW State: 0x18
I2C:  HW State: 0x28
I2C:  HW State: 0x28
I2C:  Finished with state: 0x00
Wrote 1 bytes to slave 0xFE
  [  0] = 0x0A (10)
--------------------------------------------------------------------------------
i2c read 0xFE 0x00
I2C:Starting transfer with device address: 0xFF
I2C:  HW State: 0x08
I2C:  HW State: 0x18
I2C:  HW State: 0x28
I2C:  HW State: 0x10
I2C:  HW State: 0x40
I2C:  HW State: 0x58
FreeRTOS ASSERT() occurred indicating an error condition that should NEVER happen
 - Did you call a blocking API or non FromISR() API inside an ISR?
 - Did you forget to use fprintf(stderr) in an ISR?
Here is the line of code that halted the CPU:
( portNVIC_INT_CTRL_REG & portVECTACTIVE_MASK ) == 0
```

Screenshot shows valid write (on top), and read not working

main.c

```
static volatile uint8_t slave_memory[256];

bool i2c_slave_callback__read_memory(uint8_t memory_index, uint8_t *memory) {
  if (memory_index < 256) {
    *memory = slave_memory[memory_index];
    return true;
  } else {
    return false;
  }
}

bool i2c_slave_callback__write_memory(uint8_t memory_index, uint8_t memory_value) {
  printf("callback_write\n");
  if (memory_index < 256) {
    slave_memory[memory_index] = memory_value;
    return true;
  } else {
    return false;
  }
}

void i2c_part_3_a() {
  i2c2__slave_init(0xFE);
  /**
   * Note: When another Master interacts with us, it will invoke the I2C interrupt
   *.      which will then invoke our i2c_slave_callbacks_*() from above
   *       And thus, we only need to react to the changes of memory
   */
  // while (1) {
  //   if (slave_memory[0] == 0) {
  //     // turn_on_an_led(); // TODO
  //   } else {
  //     // turn_off_an_led(); // TODO
  //   }

  //   // of course, your slave should be more creative than a simple LED on/off
  // }

  return -1;
}

int main(void) {
  create_blinky_tasks();
  create_uart_task();

  // If you have the ESP32 wifi module soldered on the board, you can try uncommenting this
code
  // See esp32/README.md for more details
  // uart3_init();                                                // Also include:  uart3_init.h
  // xTaskCreate(esp32_tcp_hello_world_task, "uart3", 1000, NULL, PRIORITY_LOW, NULL);
```

```
// Include esp32_task.h
  i2c_part_3_a();
  puts("Starting RTOS");
  i2c_slave_callback__write_memory(0, 10);
  i2c_slave_callback__read_memory(0, slave_memory);
  while (1) {
  }
  vTaskStartScheduler(); // This function never returns unless RTOS scheduler runs out of
memory and fails

  return 0;
}
```

i2c.c

```
#include "i2c.h"
#include "i2c_slave_init.h"
#include <stdio.h>

#include "FreeRTOS.h"
#include "semphr.h"
#include "task.h"

#include "common_macros.h"
#include "lpc40xx.h"
#include "lpc_peripherals.h"

#include "i2c_slave_functions.h"

/// Set to non-zero to enable debugging, and then you can use
/// I2C__DEBUG_PRINTF()
#define I2C__ENABLE_DEBUGGING 1

#if I2C__ENABLE_DEBUGGING
#include <stdio.h>
#define I2C__DEBUG_PRINTF(f_, ...)       \
  do                                     \
  {                                      \
    fprintf(stderr, "I2C:");             \
    fprintf(stderr, (f_), ##__VA_ARGS__); \
    fprintf(stderr, "\n");               \
  } while (0)
#else
#define I2C__DEBUG_PRINTF(f_, ...) /* NOOP */
#endif

/**
 * Data structure for each I2C peripheral
```

```
 */
typedef struct
{
  LPC_I2C_TypeDef * const
     registers;  ///< LPC memory mapped registers for the I2C bus
  const char * rtos_isr_trace_name;

  // Transfer complete signal informs us when I2C state machine driven by ISR
  // has finished
  SemaphoreHandle_t transfer_complete_signal;

  // Mutex to ensure only one transaction is performed at a time
  SemaphoreHandle_t mutex;

  // These are the parameters we save before a transaction is started
  uint8_t error_code;
  uint8_t slave_address;
  uint8_t starting_slave_memory_address;

  uint8_t * input_byte_pointer;  ///< Used for reading I2C slave device
  const uint8_t *
     output_byte_pointer;  ///< Used for writing data to the I2C slave device
  size_t number_of_bytes_to_transfer;
} i2c_s;

/// Instances of structs for each I2C peripheral
static i2c_s i2c_structs[] = {
  { LPC_I2C0, "i2c0" },
  { LPC_I2C1, "i2c1" },
  { LPC_I2C2, "i2c2" },
};

static bool i2c__transfer(i2c_s * i2c,
                          uint8_t slave_address,
                          uint8_t starting_slave_memory_address,
                          uint8_t * input_byte_pointer,
                          const uint8_t * output_byte_pointer,
                          uint32_t number_of_bytes_to_transfer);
static bool i2c__transfer_unprotected(i2c_s * i2c,
                                      uint8_t slave_address,
                                      uint8_t starting_slave_memory_address,
                                      uint8_t * input_byte_pointer,
                                      const uint8_t * output_byte_pointer,
                                      uint32_t number_of_bytes_to_transfer);
static void i2c__kick_off_transfer(i2c_s * i2c,
                                   uint8_t slave_address,
                                   uint8_t starting_slave_memory_address,
                                   uint8_t * input_byte_pointer,
                                   const uint8_t * output_byte_pointer,
                                   uint32_t number_of_bytes_to_transfer);
```

```
static bool i2c__handle_state_machine(i2c_s * i2c);
static bool i2c__set_stop(LPC_I2C_TypeDef * i2c);
static void i2c__handle_interrupt(i2c_s * i2c);

/**
 * 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.
 */
static void i2c__flag_read_mode(uint8_t * slave_address)
{
  *slave_address |= 0x01;
}
static void i2c__flag_write_mode(uint8_t * slave_address)
{
  *slave_address &= ~0x01;
}
static uint8_t i2c__read_address(uint8_t slave_address)
{
  return (slave_address | 0x01);
}
static uint8_t i2c__write_address(uint8_t slave_address)
{
  return (slave_address & 0xFE);
}
static bool i2c__is_read_address(uint8_t slave_address)
{
  return 0 != (slave_address & 0x01);
}

/**
 * CONSET and CONCLR register bits
 * 0x04 AA
 * 0x08 SI
 * 0x10 STOP
 * 0x20 START
 * 0x40 ENABLE
 */
static void i2c__clear_si_flag_for_hw_to_take_next_action(LPC_I2C_TypeDef * i2c)
{
  i2c->CONCLR = 0x08;
}
static void i2c__set_start_flag(LPC_I2C_TypeDef * i2c)
{
  i2c->CONSET = 0x20;
}
static void i2c__clear_start_flag(LPC_I2C_TypeDef * i2c)
{
  i2c->CONCLR = 0x20;
}
static void i2c__set_ack_flag(LPC_I2C_TypeDef * i2c)
```

```
{
  i2c->CONSET = 0x04;
}
static void i2c__set_nack_flag(LPC_I2C_TypeDef * i2c)
{
  i2c->CONCLR = 0x04;
}

static void i2c0_isr(void)
{
  i2c__handle_interrupt(&i2c_structs[I2C__0]);
}
static void i2c1_isr(void)
{
  i2c__handle_interrupt(&i2c_structs[I2C__1]);
}
static void i2c2_isr(void)
{
  i2c__handle_interrupt(&i2c_structs[I2C__2]);
}

/*****************************************************************************
 *
 *                  P U B L I C   F U N C T I O N S
 *
 *****************************************************************************/

void i2c__initialize(i2c_e i2c_number,
                     uint32_t desired_i2c_bus_speed_in_hz,
                     uint32_t peripheral_clock_hz)
{
  const function__void_f isrs[]          = { i2c0_isr, i2c1_isr, i2c2_isr };
  const lpc_peripheral_e peripheral_ids[] = { LPC_PERIPHERAL__I2C0,
                                              LPC_PERIPHERAL__I2C1,
                                              LPC_PERIPHERAL__I2C2 };

  // Make sure our i2c structs size matches our map of ISRs and I2C peripheral
  // IDs
  COMPILE_TIME_ASSERT(ARRAY_SIZE(i2c_structs) == ARRAY_SIZE(isrs));
  COMPILE_TIME_ASSERT(ARRAY_SIZE(i2c_structs) == ARRAY_SIZE(peripheral_ids));

  i2c_s * i2c                       = &i2c_structs[i2c_number];
  LPC_I2C_TypeDef * lpc_i2c         = i2c->registers;
  const lpc_peripheral_e peripheral_id = peripheral_ids[i2c_number];

  // Create binary semaphore and mutex. We deliberately use non static memory
  // allocation because we do not want to statically define memory for all I2C
  // buses
  i2c->transfer_complete_signal = xSemaphoreCreateBinary();
  i2c->mutex                    = xSemaphoreCreateMutex();
```

```
  vTraceSetMutexName(i2c->mutex, "i2c_mutex");

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

  lpc_peripheral__turn_on_power_to(peripheral_id);
  lpc_i2c->CONCLR = 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 max_speed_hz   = UINT32_C(1) * 1000 * 1000;
  const uint32_t ideal_speed_hz = UINT32_C(100) * 1000;
  const uint32_t freq_hz        = (desired_i2c_bus_speed_in_hz > max_speed_hz)
                            ? ideal_speed_hz
                            : desired_i2c_bus_speed_in_hz;
  const uint32_t half_clock_divider = (peripheral_clock_hz / freq_hz) / 2;

  // Each clock phase contributes to 50%
  lpc_i2c->SCLH = ((half_clock_divider * percent_high) / 100) / 2;
  lpc_i2c->SCLL = ((half_clock_divider * percent_low) / 100) / 2;

  // Set I2C slave address to zeroes and enable I2C
  lpc_i2c->ADR0 = lpc_i2c->ADR1 = lpc_i2c->ADR2 = lpc_i2c->ADR3 = 0;

  // Enable I2C and the interrupt for it
  lpc_i2c->CONSET = 0x40;
  lpc_peripheral__enable_interrupt(peripheral_id,
                                   isrs[i2c_number],
                                   i2c_structs[i2c_number].rtos_isr_trace_name);
  // i2c2__slave_init(0xFE);
}

bool i2c__detect(i2c_e i2c_number, uint8_t slave_address)
{
  // The I2C State machine will not continue after 1st state when length is set
  // to 0
  const size_t zero_bytes      = 0;
  const uint8_t dummy_register = 0;
  uint8_t unused               = 0;
```

```
  return i2c__write_slave_data(
      i2c_number, slave_address, dummy_register, &unused, zero_bytes);
}

uint8_t i2c__read_single(i2c_e i2c_number,
                         uint8_t slave_address,
                         uint8_t slave_memory_address)
{
  uint8_t byte = 0;
  i2c__read_slave_data(
      i2c_number, slave_address, slave_memory_address, &byte, 1);
  return byte;
}

bool i2c__read_slave_data(i2c_e i2c_number,
                          uint8_t slave_address,
                          uint8_t starting_slave_memory_address,
                          uint8_t * bytes_to_read,
                          uint32_t number_of_bytes_to_transfer)
{
  const uint8_t * no_output_byte_pointer = NULL;
  i2c__flag_read_mode(&slave_address);
  return i2c__transfer(&i2c_structs[i2c_number],
                       slave_address,
                       starting_slave_memory_address,
                       bytes_to_read,
                       no_output_byte_pointer,
                       number_of_bytes_to_transfer);
}

bool i2c__write_single(i2c_e i2c_number,
                       uint8_t slave_address,
                       uint8_t slave_memory_address,
                       uint8_t value)
{
  return i2c__write_slave_data(
      i2c_number, slave_address, slave_memory_address, &value, 1);
}

bool i2c__write_slave_data(i2c_e i2c_number,
                           uint8_t slave_address,
                           uint8_t starting_slave_memory_address,
                           const uint8_t * bytes_to_write,
                           uint32_t number_of_bytes_to_transfer)
{
  uint8_t * no_input_byte_pointer = NULL;
  i2c__flag_write_mode(&slave_address);
  return i2c__transfer(&i2c_structs[i2c_number],
                       slave_address,
```

```
                starting_slave_memory_address,
                no_input_byte_pointer,
                bytes_to_write,
                number_of_bytes_to_transfer);
}

/*****************************************************************************
 *
 *              P R I V A T E   F U N C T I O N S
 *
 *****************************************************************************/

static bool i2c__transfer(i2c_s * i2c,
                  uint8_t slave_address,
                  uint8_t starting_slave_memory_address,
                  uint8_t * input_byte_pointer,
                  const uint8_t * output_byte_pointer,
                  uint32_t number_of_bytes_to_transfer)
{
  bool status = false;

  // Either the input or the output data needs to be non NULL (XOR)
  if ((NULL != input_byte_pointer) ^ (NULL != output_byte_pointer))
  {
    if (xSemaphoreTake(i2c->mutex, portMAX_DELAY))
    {
      status = i2c__transfer_unprotected(i2c,
                                slave_address,
                                starting_slave_memory_address,
                                input_byte_pointer,
                                output_byte_pointer,
                                number_of_bytes_to_transfer);
      xSemaphoreGive(i2c->mutex);
    }
  }

  I2C__DEBUG_PRINTF("  Finished with state: 0x%02X", (int)i2c->error_code);
  return status;
}

static bool i2c__transfer_unprotected(i2c_s * i2c,
                              uint8_t slave_address,
                              uint8_t starting_slave_memory_address,
                              uint8_t * input_byte_pointer,
                              const uint8_t * output_byte_pointer,
                              uint32_t number_of_bytes_to_transfer)
{
  bool status             = false;
  const uint32_t timeout_ms = 3000;
  const bool rtos_is_running =
```

```
    (taskSCHEDULER_RUNNING == xTaskGetSchedulerState());

  xSemaphoreTake(i2c->transfer_complete_signal,
                 0);  // Clear potential stale transfer complete signal
  i2c__kick_off_transfer(i2c,
                         slave_address,
                         starting_slave_memory_address,
                         input_byte_pointer,
                         output_byte_pointer,
                         number_of_bytes_to_transfer);

  // Wait for transfer to finish; the signal will be sent by the ISR once the
  // transaction finishes
  if (rtos_is_running)
  {
    // If the RTOS is running, we can block (sleep) on this signal
    if (xSemaphoreTake(i2c->transfer_complete_signal, timeout_ms))
    {
      status = (0 == i2c->error_code);
    }
  }
  else
  {
    // We cannot block on the semaphore if the RTOS is not running, so eat the
    // CPU until transaction is done
    while (!xSemaphoreTake(i2c->transfer_complete_signal, 0))
    {
    }
    status = (0 == i2c->error_code);
  }

  return status;
}

static void i2c__kick_off_transfer(i2c_s * i2c,
                                   uint8_t slave_address,
                                   uint8_t starting_slave_memory_address,
                                   uint8_t * input_byte_pointer,
                                   const uint8_t * output_byte_pointer,
                                   uint32_t number_of_bytes_to_transfer)
{
  i2c->error_code                 = 0;
  i2c->slave_address              = slave_address;
  i2c->starting_slave_memory_address = starting_slave_memory_address;

  i2c->input_byte_pointer         = input_byte_pointer;
  i2c->output_byte_pointer        = output_byte_pointer;
  i2c->number_of_bytes_to_transfer = number_of_bytes_to_transfer;

  // Send START, I2C State Machine will finish the rest through interrupts; @see
```

```
  // i2c__handle_state_machine()
  I2C__DEBUG_PRINTF("Starting transfer with device address: 0x%02X",
                    (unsigned)slave_address);
  i2c__set_start_flag(i2c->registers);
}

static bool i2c__handle_state_machine(i2c_s * i2c)
{
  enum
  {
    // General states :
    I2C__STATE_BUS_ERROR        = 0x00,
    I2C__STATE_START            = 0x08,
    I2C__STATE_REPEAT_START     = 0x10,
    I2C__STATE_ARBRITRATION_LOST = 0x38,

    // Master Transmitter States (MT):
    I2C__STATE_MT_SLAVE_ADDR_ACK  = 0x18,
    I2C__STATE_MT_SLAVE_ADDR_NACK = 0x20,
    I2C__STATE_MT_SLAVE_DATA_ACK  = 0x28,
    I2C__STATE_MT_SLAVE_DATA_NACK = 0x30,

    // Master Receiver States (MR):
    I2C__STATE_MR_SLAVE_READ_ACK  = 0x40,
    I2C__STATE_MR_SLAVE_READ_NACK = 0x48,
    I2C__STATE_MR_SLAVE_ACK_SENT  = 0x50,
    I2C__STATE_MR_SLAVE_NACK_SENT = 0x58,

    // Slave Transmitter States (MT):
    I2C__STATE_SLAVE_TRANSMIT_READ_ACK  = 0xA8,
    I2C__STATE_SLAVE_TRANSMIT_READ_NACK = 0xB0,
    I2C__STATE_SLAVE_TRANSMIT_ACK_SENT  = 0xB8,
    I2C__STATE_SLAVE_TRANSMIT_NACK_SENT = 0xC0,
    I2C__STATE_SLAVE_TRANSMIT_FINAL     = 0xC8,

    // Slave Receiver States (MR):
    I2C__STATE_SLAVE_RECEIVE_READ_ACK  = 0x60,
    I2C__STATE_SLAVE_RECEIVE_READ_NACK = 0x68,
    I2C__STATE_SLAVE_RECEIVE_ACK_SENT  = 0x80,
    I2C__STATE_SLAVE_RECEIVE_NACK_SENT = 0x88,
    I2C__STATE_SLAVE_RECEIVE_FINAL     = 0xA0,
  };

  bool stop_sent = false;

  /*

*****************************************************************************************************
*
   * Write-mode state transition :
```

```
   * I2C__STATE_START --> I2C__STATE_MT_SLAVE_ADDR_ACK -->
   *I2C__STATE_MT_SLAVE_DATA_ACK -->
   *                                  ...
   *(I2C__STATE_MT_SLAVE_DATA_ACK) --> (stop)
   *
   * Read-mode state transition :
   * I2C__STATE_START --> I2C__STATE_MT_SLAVE_ADDR_ACK --> dataAcked -->
   *I2C__STATE_REPEAT_START --> I2C__STATE_MR_SLAVE_READ_ACK For 2+ bytes:
   *I2C__STATE_MR_SLAVE_ACK_SENT --> ... (I2C__STATE_MR_SLAVE_ACK_SENT) -->
   * I2C__STATE_MR_SLAVE_NACK_SENT --> (stop) For 1  byte :
   *I2C__STATE_MR_SLAVE_NACK_SENT --> (stop)

*****************************************************************************************************************
*
   */
  int NUM_BYTES            = 1;
  bool check               = false;
  LPC_I2C_TypeDef * lpc_i2c = i2c->registers;
  const unsigned i2c_state  = lpc_i2c->STAT;
  I2C__DEBUG_PRINTF("  HW State: 0x%02X", i2c_state);

  switch (i2c_state)
  {
    // Start condition sent, so send the device address
    case I2C__STATE_START:
      lpc_i2c->DAT = i2c__write_address(i2c->slave_address);
      i2c__clear_si_flag_for_hw_to_take_next_action(lpc_i2c);
      break;

    case I2C__STATE_REPEAT_START:
      lpc_i2c->DAT = i2c__read_address(i2c->slave_address);
      i2c__clear_si_flag_for_hw_to_take_next_action(lpc_i2c);
      break;

    // Slave acknowledged its address, so send the first register
    case I2C__STATE_MT_SLAVE_ADDR_ACK:
      i2c__clear_start_flag(lpc_i2c);

      // No data to transfer, this is used just to test if the slave responds
      if (0 == i2c->number_of_bytes_to_transfer)
      {
        stop_sent = i2c__set_stop(lpc_i2c);
      }
      else
      {
        lpc_i2c->DAT = i2c->starting_slave_memory_address;
        i2c__clear_si_flag_for_hw_to_take_next_action(lpc_i2c);
      }
      break;
```

```
// Slave acknowledged the data byte we sent, so send more bytes or finish
// the transaction by sending STOP
case I2C__STATE_MT_SLAVE_DATA_ACK:
  // We were flagged as a READ transaction, so send repeat start
  if (i2c__is_read_address(i2c->slave_address))
  {
    i2c__set_start_flag(lpc_i2c);
    i2c__clear_si_flag_for_hw_to_take_next_action(lpc_i2c);
  }
  else
  {
    if (0 == i2c->number_of_bytes_to_transfer)
    {
      stop_sent = i2c__set_stop(lpc_i2c);
    }
    else
    {
      lpc_i2c->DAT = *(i2c->output_byte_pointer);
      ++(i2c->output_byte_pointer);
      --(i2c->number_of_bytes_to_transfer);
      i2c__clear_si_flag_for_hw_to_take_next_action(lpc_i2c);
    }
  }
  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 I2C__STATE_MR_SLAVE_READ_ACK:
  i2c__clear_start_flag(lpc_i2c);

  // 1+ bytes: Send ACK to receive a byte and transition to
  // I2C__STATE_MR_SLAVE_ACK_SENT
  if (i2c->number_of_bytes_to_transfer > 1)
  {
    i2c__set_ack_flag(lpc_i2c);
  }
  // 1 byte : NACK next byte to go to I2C__STATE_MR_SLAVE_NACK_SENT for
  // 1-byte read
  else
  {
    i2c__set_nack_flag(lpc_i2c);
  }

  i2c__clear_si_flag_for_hw_to_take_next_action(lpc_i2c);
  break;

case I2C__STATE_MR_SLAVE_ACK_SENT:
  *(i2c->input_byte_pointer) = lpc_i2c->DAT;
```

```
  ++(i2c->input_byte_pointer);
  --(i2c->number_of_bytes_to_transfer);

  if (1 == i2c->number_of_bytes_to_transfer)
  {                           // Only 1 more byte remaining
    i2c__set_nack_flag(lpc_i2c);  // NACK next byte --> Next state:
                              // I2C__STATE_MR_SLAVE_NACK_SENT
  }
  else
  {
    i2c__set_ack_flag(
        lpc_i2c);  // ACK next byte --> Next state:
                // I2C__STATE_MR_SLAVE_ACK_SENT(back to this state)
  }

  i2c__clear_si_flag_for_hw_to_take_next_action(lpc_i2c);
  break;

 case I2C__STATE_MR_SLAVE_NACK_SENT:  // Read last-byte from Slave
  printf("inside MR\n");
  *(i2c->input_byte_pointer)       = lpc_i2c->DAT;
  i2c->number_of_bytes_to_transfer = 0;
  stop_sent                        = i2c__set_stop(lpc_i2c);
  break;

 case I2C__STATE_ARBRITRATION_LOST:
  // We should not issue stop() in this condition, but we still need to end
  // our  transaction.
  stop_sent       = true;
  i2c->error_code = lpc_i2c->STAT;
  break;

 // 0x60
 case I2C__STATE_SLAVE_RECEIVE_READ_ACK:
  check          = true;
  lpc_i2c->CONSET = 0x04;  // write 0x04 to *2CONSET to set the AA bit
  lpc_i2c->CONCLR = 0x08;  // write 0x08 to I2CONCLR to clear the SI flag.
                           // triggering interrupt
  i2c->number_of_bytes_to_transfer = NUM_BYTES;
  break;
 // 0x68
 case I2C__STATE_SLAVE_RECEIVE_READ_NACK:
  lpc_i2c->CONSET                  = 0x24;
  lpc_i2c->CONCLR                  = 0x08;
  i2c->number_of_bytes_to_transfer = NUM_BYTES;
  break;
 // 0x80
 case I2C__STATE_SLAVE_RECEIVE_ACK_SENT:
  /*
 1. Read data byte from I2DAT into the Slave Receive buffer.
```

```
2. Decrement the Slave data counter, skip to step 5 if not the last data
byte.
3. Write 0x0C to I2CONCLR to clear the SI flag and the AA bit.
4. Exit.
5. Write 0x04 to I2CONSET to set the AA bit.
6. Write 0x08 to I2CONCLR to clear the SI flag.
7. Increment Slave Receive buffer pointer.
8. Exit*/
  /*
  uint_8 value = i2c->DAT & 0xFF;
      i2c->DATA_BUFFER |= value;
      //Decrement the Slave data counter
      if (the last data byte.) {
          //steps 3 and 4
      }*/
  // uint_8 value = lpc_i2c->DAT & 0xFF;
  // lpc_i2c->DATA_BUFFER |= value;
  // lpc_i2c->DATA_BUFFER |= lpc_i2c->DAT & 0xFF;
  *(i2c->input_byte_pointer) = lpc_i2c->DAT;
  --(i2c->number_of_bytes_to_transfer);
  if (check)
  {  // check to see if this is 0
    lpc_i2c->CONCLR = 0x0C;
    break;
  }
  lpc_i2c->CONSET = 0x04;
  lpc_i2c->CONCLR = 0x08;
  // lpc_i2c->DAT = *(i2c->output_byte_pointer);
  ++(i2c->input_byte_pointer);
  // i2c_state = I2C__STATE
  break;
// 0x88
case I2C__STATE_SLAVE_RECEIVE_NACK_SENT:
  lpc_i2c->CONSET = 0x04;
  lpc_i2c->CONCLR = 0x08;
  break;
// 0xA0
case I2C__STATE_SLAVE_RECEIVE_FINAL:
  lpc_i2c->CONSET = 0x04;
  lpc_i2c->CONCLR = 0x08;
  break;

// Start of Transmit
// 0xA8
case I2C__STATE_SLAVE_TRANSMIT_READ_ACK:
  // LPC_I2C2->DAT
  lpc_i2c->DAT    = *(i2c->output_byte_pointer);
  lpc_i2c->CONSET = 0x04;
  lpc_i2c->CONCLR = 0x08;
  ++(i2c->output_byte_pointer);
```

```
    break;
   // 0xB0
   case I2C__STATE_SLAVE_TRANSMIT_READ_NACK:
    lpc_i2c->DAT    = *(i2c->output_byte_pointer);
    lpc_i2c->CONSET = 0x24;
    lpc_i2c->CONCLR = 0x08;
    ++(i2c->output_byte_pointer);
    break;
   // 0xB8
   case I2C__STATE_SLAVE_TRANSMIT_ACK_SENT:
    lpc_i2c->DAT    = *(i2c->output_byte_pointer);
    lpc_i2c->CONSET = 0x04;
    lpc_i2c->CONCLR = 0x08;
    ++(i2c->output_byte_pointer);
    break;
   // 0xC0
   case I2C__STATE_SLAVE_TRANSMIT_NACK_SENT:
    lpc_i2c->CONSET = 0x04;
    lpc_i2c->CONCLR = 0x08;
    break;
   // 0xC8
   case I2C__STATE_SLAVE_TRANSMIT_FINAL:
    lpc_i2c->CONSET = 0x04;
    lpc_i2c->CONCLR = 0x08;
    break;

   case I2C__STATE_MT_SLAVE_ADDR_NACK:  // no break
   case I2C__STATE_MT_SLAVE_DATA_NACK:  // no break
   case I2C__STATE_MR_SLAVE_READ_NACK:  // no break
   case I2C__STATE_BUS_ERROR:           // no break

   default:
    i2c->error_code = lpc_i2c->STAT;
    stop_sent       = i2c__set_stop(lpc_i2c);
    break;
  }

  return stop_sent;
}

static bool i2c__set_stop(LPC_I2C_TypeDef * i2c)
{
  const uint32_t stop_bit = (1 << 4);
  i2c__clear_start_flag(i2c);
  i2c->CONSET = stop_bit;
  i2c__clear_si_flag_for_hw_to_take_next_action(i2c);

  while (i2c->CONSET & stop_bit)
  {
  }
```

```
  return true;
}

static void i2c__handle_interrupt(i2c_s * i2c)
{
  const bool stop_sent = i2c__handle_state_machine(i2c);

  if (stop_sent)
  {
    long higher_priority_task_woke = 0;
    xSemaphoreGiveFromISR(i2c->transfer_complete_signal,
                          &higher_priority_task_woke);
    portEND_SWITCHING_ISR(higher_priority_task_woke);
  }
}
```