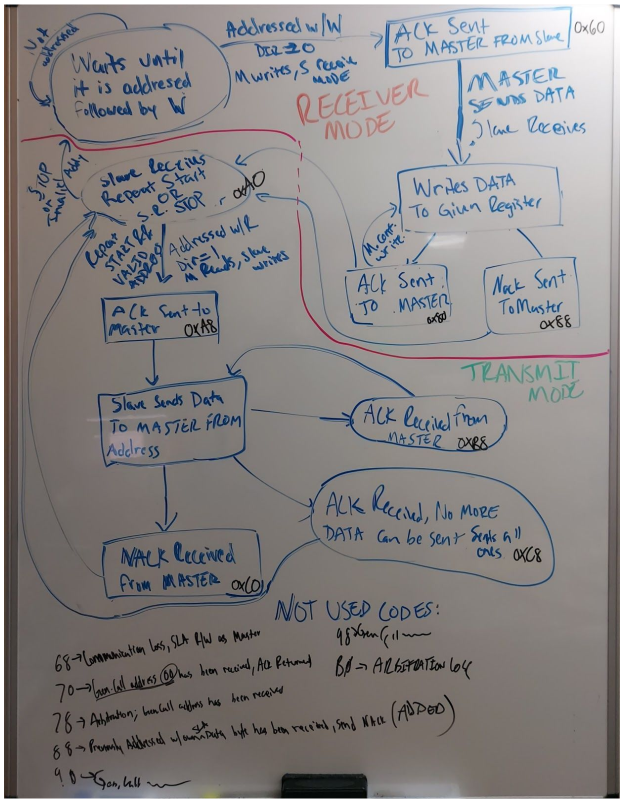
Prabjyot Obhi
Nick Scheele

I2C Lab

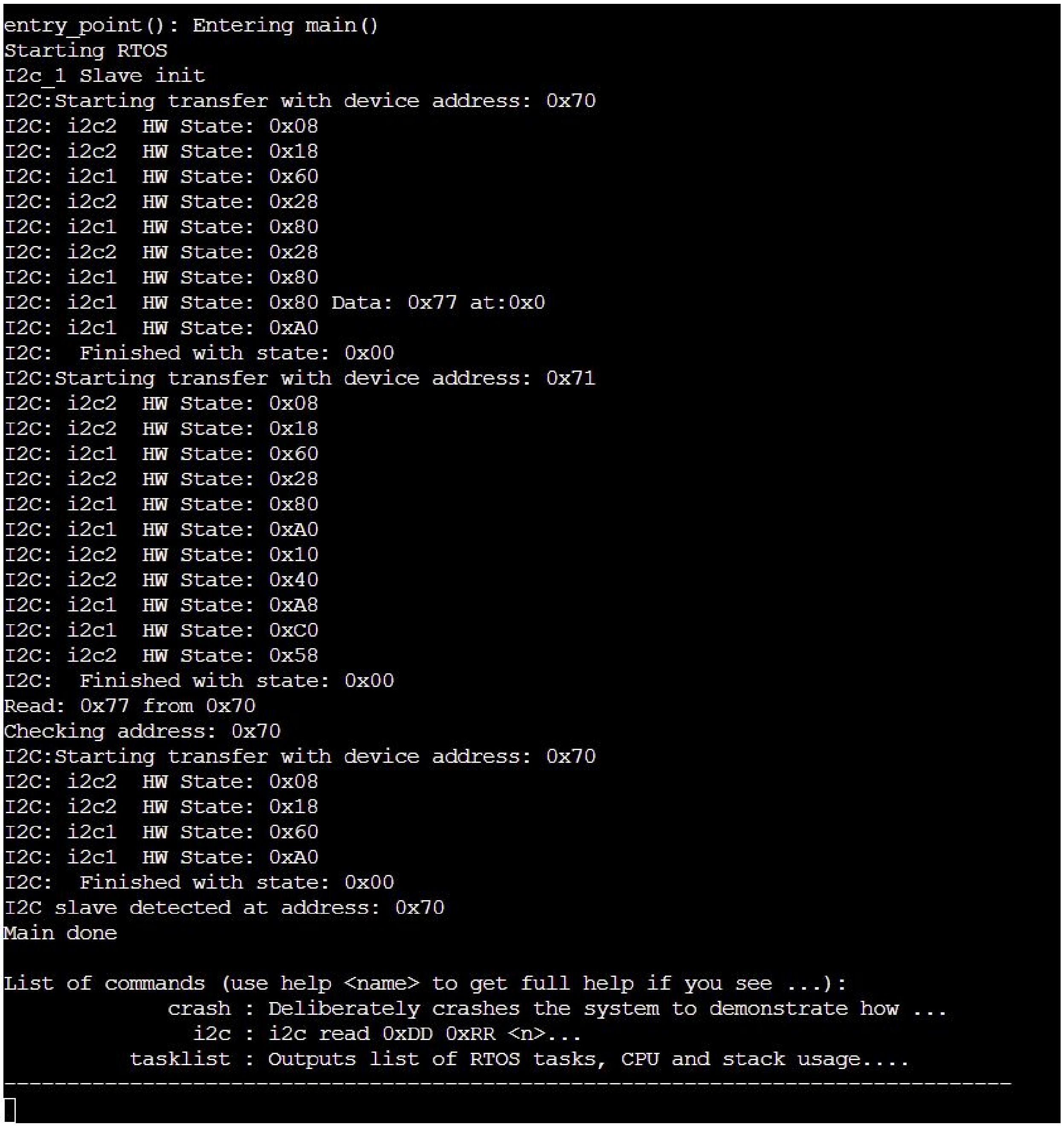
Parts 0 and 1 of this lab were primarily focused on reading the given source code and data sheet. We needed to enable the debugging printfs feature in the i2c.c file to see output in a telemetry window. In part 2 we were tasked with creating a state machine for the way that slave devices should respond to a master's actions. In this portion, the most difficult step was logically working our way through how a slave device should respond in any situation. Below is an image of our state machine and the debugging printfs.


Not addressed
Waits until it is addresed Followed by W
Addressed w/W DIR=0 M writes, S receive MODE
ACK Sent TO MASTER FROM Slave 0x60
RECEIVER MODE
MASTER SENDS DATA Slave Receives
Writes DATA To Given Register
M. cont. write
ACK Sent TO MASTER 0x80
Nack Sent To Master 0x88
STOP or Invalid Addr
Slave Receives Repeat Start OR S.R. STOP, 0xA0
Repeat START & VALID ADDR
Addressed w/R Dir=1 M Reads, slave writes
ACK Sent to master 0xA8
TRANSMIT MODE
Slave Sends Data To MASTER FROM Address
ACK Received from MASTER 0xB8
ACK Received, No MORE DATA Can be sent Sends all ones 0xC8
NACK Received From MASTER 0xC0
NOT USED CODES:
68 → Communication loss, SLA R/W as Master
70 → Gen. Call address (00) has been received, ACK Returned
78 → Arbitration; Gen Call address has been received
88 → Previously Addressed w/ own SLA Data byte has been received, Send NACK (ADDED)
90 → Gen, Call
98 → Gen Call
B0 → ARBITRATION LOS

The debugging printfs go on for quite a while, but once we reach addresses 0x38 and 0x39, the statements change because these are the addresses of actual sensors being interfaced to the board.

In part 3a, we needed to configure our slave board to respond to a particular I2C bus address. To do this, we followed the example given for the initialization of the master, but we altered it slightly to account for the uniqueness of a slave device. The most difficulty came from the actual detection of the slave device. Initially, in order to detect addresses a for loop was used and debug printfs were analyzed.

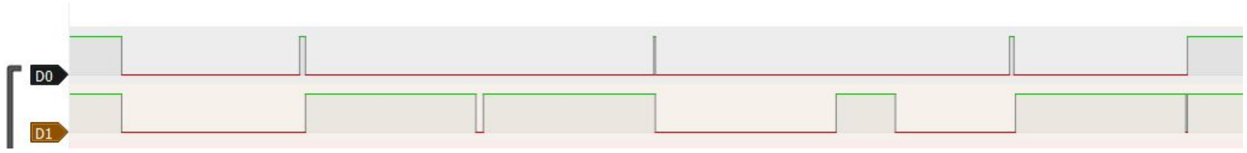
Once this was done, we had to implement two callback functions for the slave. These two functions related to whether the slave was in transmitter mode or receiver mode. We also had to take the logical state machine from part 2 and transfer it to code. There were quite a lot of examples given in the i2c.c file to follow, so this was a general mapping of the state machine to code. The following are different scenarios we had tested/played around with.



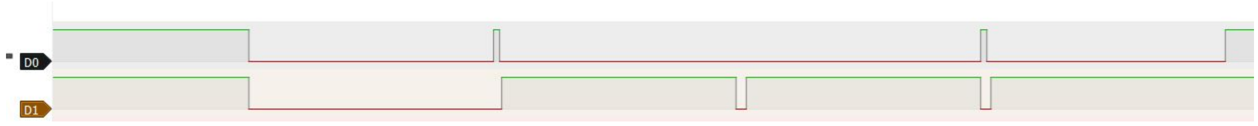
```
entry_point(): Entering main()
Starting RTOS
I2c_1 Slave init
I2C:Starting transfer with device address: 0x70
I2C: i2c2  HW State: 0x08
I2C: i2c2  HW State: 0x18
I2C: i2c1  HW State: 0x60
I2C: i2c2  HW State: 0x28
I2C: i2c1  HW State: 0x80
I2C: i2c2  HW State: 0x28
I2C: i2c1  HW State: 0x80
I2C: i2c1  HW State: 0x80 Data: 0x77 at:0x0
I2C: i2c1  HW State: 0xA0
I2C:  Finished with state: 0x00
I2C:Starting transfer with device address: 0x71
I2C: i2c2  HW State: 0x08
I2C: i2c2  HW State: 0x18
I2C: i2c1  HW State: 0x60
I2C: i2c2  HW State: 0x28
I2C: i2c1  HW State: 0x80
I2C: i2c1  HW State: 0xA0
I2C: i2c2  HW State: 0x10
I2C: i2c2  HW State: 0x40
I2C: i2c1  HW State: 0xA8
I2C: i2c1  HW State: 0xC0
I2C: i2c2  HW State: 0x58
I2C:  Finished with state: 0x00
Read: 0x77 from 0x70
Checking address: 0x70
I2C:Starting transfer with device address: 0x70
I2C: i2c2  HW State: 0x08
I2C: i2c2  HW State: 0x18
I2C: i2c1  HW State: 0x60
I2C: i2c1  HW State: 0xA0
I2C:  Finished with state: 0x00
I2C slave detected at address: 0x70
Main done

List of commands (use help <name> to get full help if you see ...):
             crash : Deliberately crashes the system to demonstrate how ...
               i2c : i2c read 0xDD 0xRR <n>...
          tasklist : Outputs list of RTOS tasks, CPU and stack usage....
--------------------------------------------------------------------------------
```

Once we were able to read and write via the i2c cli, we needed to try to capture and analyze the logic analyzer output. This was rather difficult, and has been difficult throughout the semester because of the quality of the logic analyzer, but with enough time and trial and error we were able to see waveforms generated.

Read Logic Analyzer Output [SCL → D0, SDA → D1]



Write Logic Analyzer Output [SCL → D0, SDA → D1]



i2c.c

```
#include "i2c.h"

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

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

/// Set to non-zero to enable debugging, and then you can use I2C__DEBUG_PRINTF()
#define I2C__ENABLE_DEBUGGING 0

#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)
```

```c
#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;

 volatile size_t slave_mem_loc;
 volatile uint8_t slave_memory[256];
} 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.
*/
```

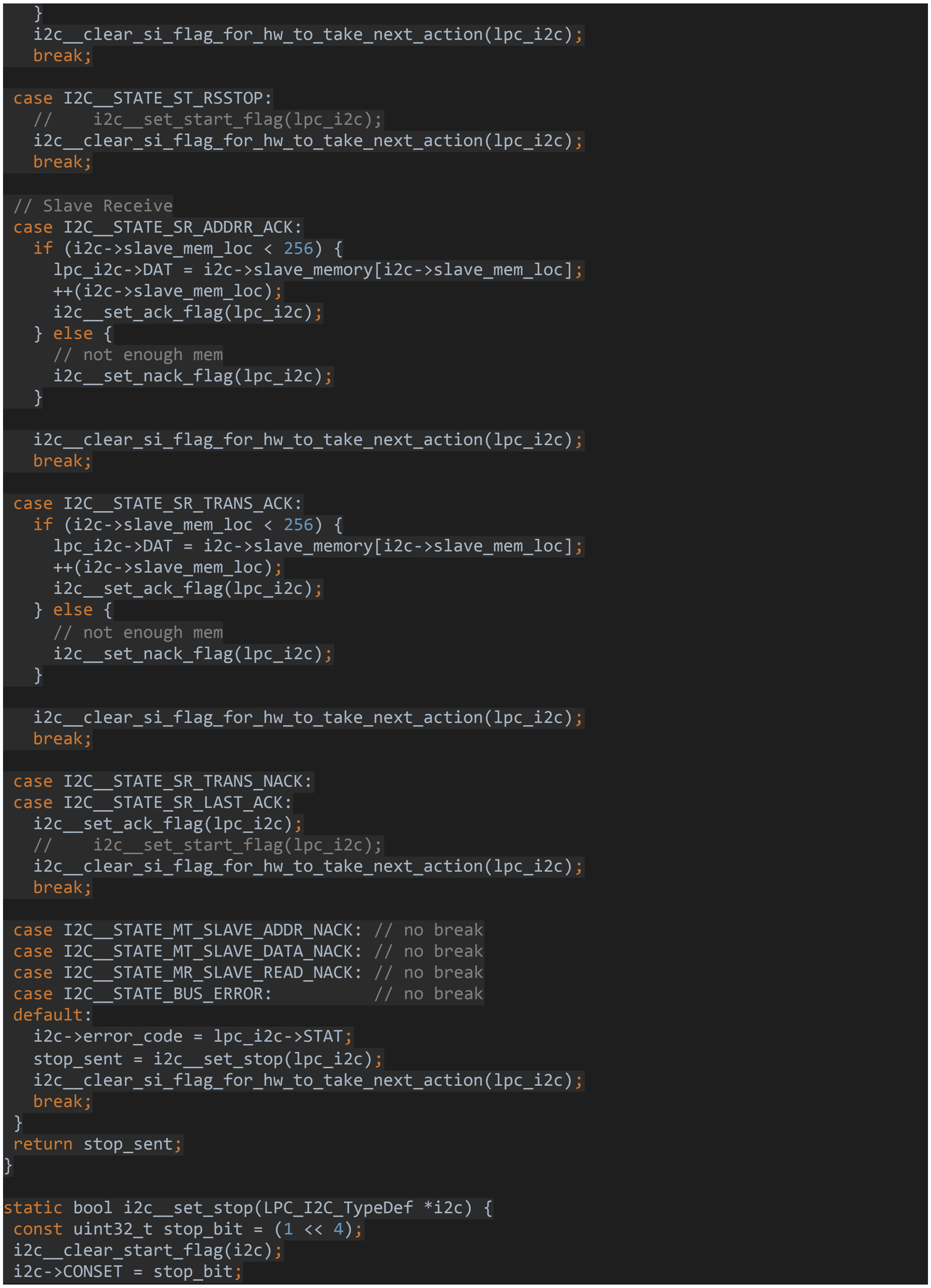
```c
static void i2c__flag_read_mode(uint8_t *slave_address) { *slave_address |= 0x01; }
static void i2c__flag_write_mode(uint8_t *slave_address) { *slave_address &= ~0x01; }
static uint8_t i2c__read_address(uint8_t slave_address) { return (slave_address | 0x01); }
static uint8_t i2c__write_address(uint8_t 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);
}

void i2c__initialize_slave(i2c_e i2c_number) {
 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};

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

 lpc_peripheral__turn_on_power_to(peripheral_id);
 lpc_i2c->CONCLR = 0x6C; // Clear ALL I2C Flags

 lpc_i2c->ADR0 = 0x70; // Edits
                       //  lpc_i2c->ADR1 = 0x74; // Edits
                       //  lpc_i2c->ADR2 = 0x78; // Edits
                       //  lpc_i2c->ADR3 = 0x7c; // Edits

 // Enable I2C and the interrupt for it
 lpc_i2c->CONSET = 0x44; // Edits
 lpc_peripheral__enable_interrupt(peripheral_id, isrs[i2c_number],
i2c_structs[i2c_number].rtos_isr_trace_name);
}

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,
```

```c
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_ARBITRATION_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:
   I2C__STATE_ST_ADDRW_ACK = 0x60,
   I2C__STATE_ST_ADDRD_ACK = 0x80,
   I2C__STATE_ST_ADDRD_NACK = 0x88,
   I2C__STATE_ST_RSSTOP = 0xA0,

   // Slave Reciever States:
   I2C__STATE_SR_ADDRR_ACK = 0xA8,
   I2C__STATE_SR_TRANS_ACK = 0xB8,
   I2C__STATE_SR_TRANS_NACK = 0xC0,
   I2C__STATE_SR_LAST_ACK = 0xC8,
 };

 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)

*********************************************************************************************
***************
  */

 LPC_I2C_TypeDef *lpc_i2c = i2c->registers;
 const char *i2cname = i2c->rtos_isr_trace_name;

 const unsigned i2c_state = lpc_i2c->STAT;
 I2C__DEBUG_PRINTF(" %s  HW State: 0x%02X", i2cname, 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:
```

```c
   *(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
   *(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;

 // Slave Transfer
 case I2C__STATE_ST_ADDRW_ACK:
   i2c->slave_mem_loc = -1;
   i2c__set_ack_flag(lpc_i2c);
   i2c__clear_si_flag_for_hw_to_take_next_action(lpc_i2c);
   break;

 case I2C__STATE_ST_ADDRD_ACK:
   if (i2c->slave_mem_loc == -1) {
     i2c->slave_mem_loc = lpc_i2c->DAT;
     i2c__set_ack_flag(lpc_i2c);
   } else if (i2c->slave_mem_loc < 256) {
     i2c->slave_memory[i2c->slave_mem_loc] = lpc_i2c->DAT;
     ++(i2c->slave_mem_loc);
     i2c__set_ack_flag(lpc_i2c);
   } else {
     // not enough mem
     i2c__set_nack_flag(lpc_i2c);
   }

   i2c__clear_si_flag_for_hw_to_take_next_action(lpc_i2c);
   break;

 case I2C__STATE_ST_ADDRD_NACK:
   i2c__set_start_flag(lpc_i2c);
   if (i2c->slave_mem_loc < 256) {
     i2c->slave_memory[i2c->slave_mem_loc] = lpc_i2c->DAT;
     ++(i2c->slave_mem_loc);
     i2c__set_ack_flag(lpc_i2c);
   } else {
     // not enough mem
     i2c__set_nack_flag(lpc_i2c);
```

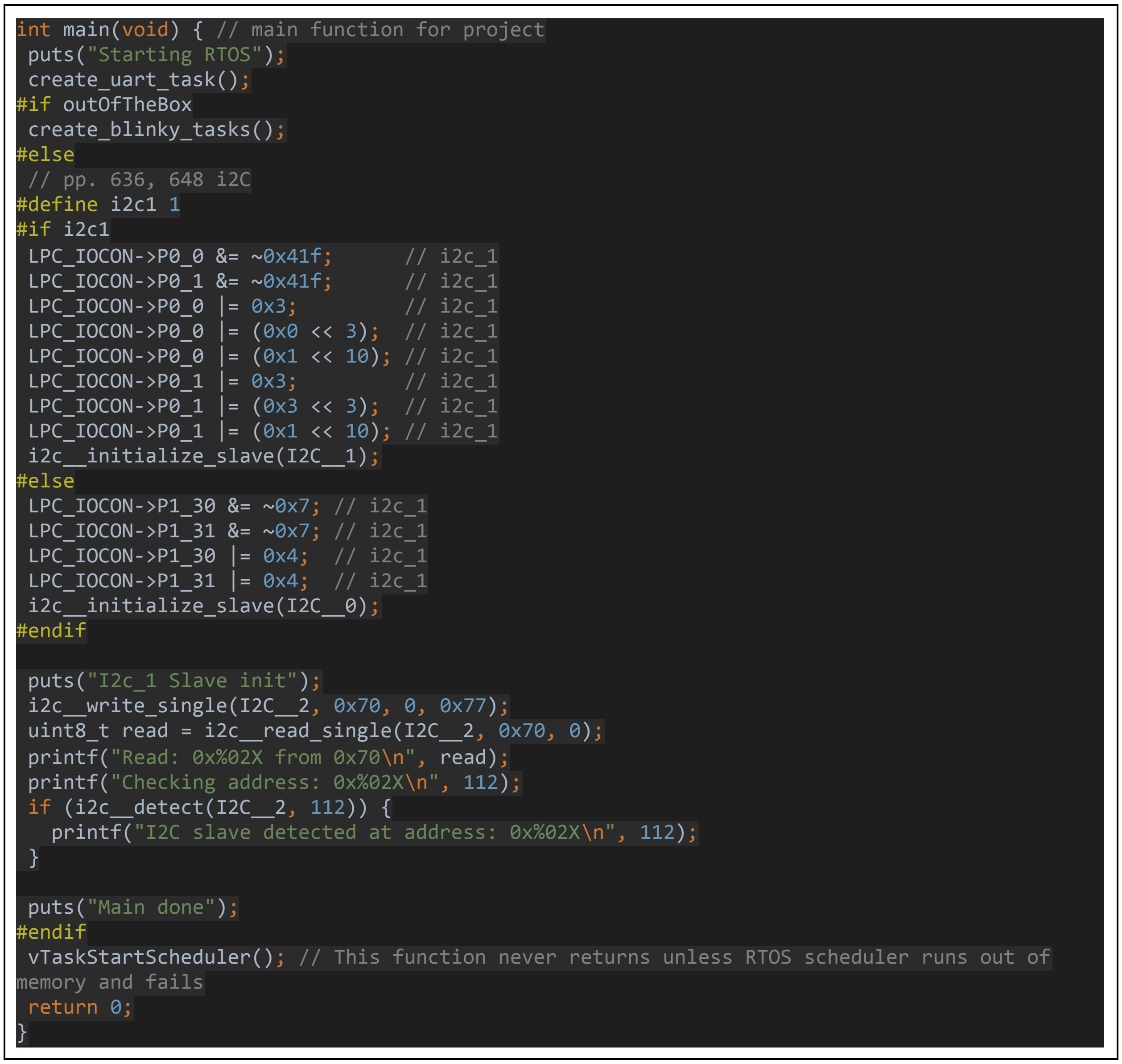


```
   }
   i2c__clear_si_flag_for_hw_to_take_next_action(lpc_i2c);
   break;

 case I2C__STATE_ST_RSSTOP:
   //    i2c__set_start_flag(lpc_i2c);
   i2c__clear_si_flag_for_hw_to_take_next_action(lpc_i2c);
   break;

 // Slave Receive
 case I2C__STATE_SR_ADDRR_ACK:
   if (i2c->slave_mem_loc < 256) {
     lpc_i2c->DAT = i2c->slave_memory[i2c->slave_mem_loc];
     ++(i2c->slave_mem_loc);
     i2c__set_ack_flag(lpc_i2c);
   } else {
     // not enough mem
     i2c__set_nack_flag(lpc_i2c);
   }

   i2c__clear_si_flag_for_hw_to_take_next_action(lpc_i2c);
   break;

 case I2C__STATE_SR_TRANS_ACK:
   if (i2c->slave_mem_loc < 256) {
     lpc_i2c->DAT = i2c->slave_memory[i2c->slave_mem_loc];
     ++(i2c->slave_mem_loc);
     i2c__set_ack_flag(lpc_i2c);
   } else {
     // not enough mem
     i2c__set_nack_flag(lpc_i2c);
   }

   i2c__clear_si_flag_for_hw_to_take_next_action(lpc_i2c);
   break;

 case I2C__STATE_SR_TRANS_NACK:
 case I2C__STATE_SR_LAST_ACK:
   i2c__set_ack_flag(lpc_i2c);
   //    i2c__set_start_flag(lpc_i2c);
   i2c__clear_si_flag_for_hw_to_take_next_action(lpc_i2c);
   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);
   i2c__clear_si_flag_for_hw_to_take_next_action(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;
```

```c
 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);
 }
}
```

main.c

```c
#if i2c
static volatile uint8_t slave_memory[256];
bool i2c_slave_callback__read_memory(uint8_t memory_index, uint8_t *memory) {
 // TODO: Read the data from slave_memory[memory_index] to *memory pointer
 // TODO: return true if all is well (memory index is within bounds)
 bool status = true;
 // LPC_I2C2->ADR00 |= memory_index;
 *memory = slave_memory[memory_index];
 if (LPC_I2C2->STAT == 0x78 || LPC_I2C2->STAT == 0xB0) {
   status = false;
 }
 return status;
}

bool i2c_slave_callback__write_memory(uint8_t memory_index, uint8_t memory_value) {
 // TODO: Write the memory_value at slave_memory[memory_index]
 // TODO: return true if memory_index is within bounds
 bool status = true;
 slave_memory[memory_index] = memory_value;
 if (LPC_I2C2->STAT == 0x78 || LPC_I2C2->STAT == 0xB0) {
   status = false;
 }
 return status;
}
#endif
```



```
int main(void) { // main function for project
 puts("Starting RTOS");
 create_uart_task();
#if outOfTheBox
 create_blinky_tasks();
#else
 // pp. 636, 648 i2C
#define i2c1 1
#if i2c1
 LPC_IOCON->P0_0 &= ~0x41f;       // i2c_1
 LPC_IOCON->P0_1 &= ~0x41f;       // i2c_1
 LPC_IOCON->P0_0 |= 0x3;          // i2c_1
 LPC_IOCON->P0_0 |= (0x0 << 3);   // i2c_1
 LPC_IOCON->P0_0 |= (0x1 << 10);  // i2c_1
 LPC_IOCON->P0_1 |= 0x3;          // i2c_1
 LPC_IOCON->P0_1 |= (0x3 << 3);   // i2c_1
 LPC_IOCON->P0_1 |= (0x1 << 10);  // i2c_1
 i2c__initialize_slave(I2C__1);
#else
 LPC_IOCON->P1_30 &= ~0x7; // i2c_1
 LPC_IOCON->P1_31 &= ~0x7; // i2c_1
 LPC_IOCON->P1_30 |= 0x4;  // i2c_1
 LPC_IOCON->P1_31 |= 0x4;  // i2c_1
 i2c__initialize_slave(I2C__0);
#endif

 puts("I2c_1 Slave init");
 i2c__write_single(I2C__2, 0x70, 0, 0x77);
 uint8_t read = i2c__read_single(I2C__2, 0x70, 0);
 printf("Read: 0x%02X from 0x70\n", read);
 printf("Checking address: 0x%02X\n", 112);
 if (i2c__detect(I2C__2, 112)) {
   printf("I2C slave detected at address: 0x%02X\n", 112);
 }

 puts("Main done");
#endif
 vTaskStartScheduler(); // This function never returns unless RTOS scheduler runs out of
memory and fails
 return 0;
}
```