I2C SELF TEST

① read external BH1750 ambient light sensor

② self I2C master —— I2C slave

I2C master gpio18 gpio 19

I2C slave gpio25 gpio 26

(external pull-up resistors not required for SDA/SCL as driver will enable internal pull-up resistors)

```
make menuconfig / idf.py menuconfig
```

I2C master -> pins, port, frequency

I2C slave -> pins, port, address

ADDR low

BH1750 sensor -> slave address (0x23 / 0x5c),

operation mode (one time L-resolution)

(4 lux, time = 16ms)

```
make -j4 flash monitor
```

```
#include <stdio.h>
#include "esp_log.h"
#include "driver/i2c.h"
#include "sdkconfig.h"


static const char *TAG = "i2c-example";

~~#de~~

#define DATA_LENGTH 512
#define RW_TEST_LENGTH 128
#define DELAY_TIME_BETWEEN_ITEMS_MS 1000
```

```
#define I2C_SLAVE_SCL_IO   CONFIG_I2C_SLAVE_SCL
#define I2C_SLAVE_SDA_IO   CONFIG_I2C_SLAVE_SDA
#define I2C_SLAVE_NUM   I2C_NUMBER(CONFIG_I2C_SLAVE_PORT_NUM)
#define I2C_SLAVE_TX_BUF_LEN   (2* DATA_LENGTH)
#define I2C_SLAVE_RX_BUF_LEN   (2* DATA_LENGTH)

#define I2C_MASTER_SCL_IO   CONFIG_I2C_MASTER_SCL
#define I2C_MASTER_SDA_IO   CONFIG_I2C_MASTER_SDA
#define I2C_MASTER_NUM   I2C_NUMBER(CONFIG_I2C_MASTER_PORT_NUM)
#define I2C_MASTER_FREQ_HZ   CONFIG_I2C_MASTER_FREQUENCY
#define I2C_MASTER_TX_BUF_DISABLE   0
#define I2C_MASTER_RX_BUF_DISABLE   0

#define BH1750_SENSOR_ADDR   CONFIG_BH1750_ADDR
#define BH1750_CMD_START   CONFIG_BH1750_OPMODE
#define ESP_SLAVE_ADDR   CONFIG_I2C_SLAVE_ADDRESS
#define WRITE_BIT   I2C_MASTER_WRITE
#define READ_BIT   I2C_MASTER_READ
#define ACK_CHECK_EN   0x1
#define ACK_CHECK_DIS   0x0
#define ACK_VAL   0x0
#define NACK_VAL   0x1

SemaphoreHandle_t print_mux = NULL;
```

```
Static esp-err-t i2c-master-read-slave ( i2c-port-t
  i2c_num, uint8_t *data_rd , size_t size) {
  if (size == 0) return ESP-OK;
  i2c-cmd-handle-t cmd = i2c_cmd_link_create();
  i2c_master_start (cmd);
  i2c-master_write-byte (cmd, (ESP_SLAVE_ADDR <<1)
     | READ_BIT, ACK_CHECK_EN);
  if (size >1) {
     i2c_master_read (cmd, data-rd, size-1, ACK-VAL);
  }
  i2c master_read_byte (cmd, data_rd+size -1, NACK-VAL);
  i2c-master-stop(cmd);
                          -cmd
  esp_err_t ret = i2c. master_begin
                          ^
      (i2c-num, cmd, 1000/portTICK_RATE_MS);
  i2c- cmd-link delete (cmd);
  return ret;
}
```

```
static esp-err-t i2c-master-write-slave ( i2c port_t
  i2c_num, uint8_t *data_wr, size_t size) {
  i2c-cmd-handle-t cmd = i2c-cmd-link-create();
  i2c-master-start (cmd);
  i2c-master-write-byte(cmd, (ESP_SLAVE_ADDR <<1)
  | WRITE-BIT, ACK-CHECK_EN);
  i2c-master_write (cmd, data-wr, size, ACK_CHECK_EN);
  i2c-master- stop (cmd);
  esp-err-t ret = i2c_master_cmd-begin (i2c-num,
    cmd, 1000 / portTICK_RATE_MS);
    i2c_cmd_link_delete (cmd);
    return ret;
}


static esp_err_t i2c_master_sensor_test (i2c_port_t i2c_num,
      uint8_t *data_h, uint8_t *data_l) {
    i2c_cmd_handle_t cmd = i2c_cmd_link_create();
    i2c_master_start (cmd);
    i2c_master_write_byte (cmd, (BH1750_SENSOR_ADDR << 1)
      | WRITE_BIT, ACK_CHECK_EN);
    i2c_master_write_byte (cmd, BH1750_CMD_START, ACK_CHECK_EN);
    i2c_master_stop (cmd);
    int ret = i2c_master_cmd_begin (i2c_num, cmd, 1000/portTICK_RATE_MS);
    i2c_cmd_link_delete (cmd);
    if (ret != ESP_OK) return ret;
    vTaskDelay (30 / portTICK_RATE_MS);
    cmd = i2c_cmd_link_create ();
    i2c_master_start (cmd);
    i2c_master_write_byte (cmd, (BH1750_SENSOR_ADDR << 1)
      | READ_BIT, ACK_CHECK_EN);
    i2c_master_read_byte (cmd, data_h, ACK_VAL);
    i2c_master_read_byte (cmd, data_l, NACK_VAL);
    i2c_master_stop (cmd);
    ret = i2c_master_cmd_begin (i2c_num, cmd, 1000/ portTICK_RATE_MS);
    i2c_cmd_link_delete (cmd);
    return ret;
}
```

```
static esp_err_t i2c_master_init() {
  int i2c_master_port = I2C_MASTER_NUM;
  i2c_config_t conf;
  Conf. mode = I2C_MODE_MASTER;
  conf. sda_io_num = I2C_MASTER_SDA_IO;
  conf. sda_pullup_en = GPIO_PULLUP_ENABLE;
  Conf. scl_io_num = I2C_MASTER_SCL_IO;
  conf. scl_pullup_en = GPIO_PULLUP_ENABLE;
  conf. master. clk_speed = I2C_MASTER_FREQ_HZ;
  i2c_param_config (i2c_master_port, &conf);
  return i2c_driver_install (i2c_master_port,
        Conf. mode, I2C_MASTER_RX_BUF_DISABLE,
        I2C_MASTER_TX_BUF_DISABLE, 0);
}


static esp_err_t i2c_slave_init() {
  int i2c_slave_port = I2C_SLAVE_NUM;
  i2c_config_t conf_slave;
  conf_slave. sda_io_num = I2C_SLAVE_SDA_IO;
  conf_slave. sda_pullup_en = GPIO_PULLUP_ENABLE;
  conf_slave. scl_io_num = I2C_SLAVE_SCL_IO;
  conf_slave. scl_pullup_en = GPIO_PULLUP_ENABLE;
  conf_slave. mode = I2C_MODE_SLAVE;
  conf_slave. slave. addr_10bit_en = 0;
  conf_Slave. slave. slave_addr = ESP_SLAVE_ADDR;
  i2c_param_config (i2c_slave_port, &conf_slave);
  return i2c_driver_install (i2c_slave_port, conf_slave. mode,
      I2C_SLAVE_RX_BUF_LEN, I2C_SLAVE_TX_BUF_LEN, 0);
```

```
static void disp_buf(uint8_t *buf, int len) {
    int i;
    for (i=0; i<len; i++) {
        printf("%02x ", buf[i]);
        if ((i+1) % 16 == 0) printf("\n");
    }
    printf("\n");
}


static void i2c_test_task(void *arg) {
    int i=0;
    int ret;
    uint32_t task_idx = (uint32_t)arg;
    uint8_t *data = (uint8_t*) malloc(DATA_LENGTH);
    uint8_t *data_wr = (uint8_t*) malloc (DATA_LENGTH);
    uint8_t *data_rd = (uint8_t*) malloc(DATA_LENGTH);
    uint8_t sensor_data_h, sensor_data_l;
    int cnt = 0;
    while (1) {
        ESP_LOGI(TAG, "TASK[%d] test cnt: %d", task_idx, cnt++);
        ret = i2c_master_sensor_test(I2C_MASTER_NUM,
                &sensor_data_h, &sensor_data_l);
        xSemaphoreTake(print_mux, portMAX_DELAY);
        if (ret == ESP_ERR_TIMEOUT) {
            ESP_LOGE(TAG, "I2C timeout");
        } else if (ret == ESP_OK) {
            printf("*****************\n");
            printf("TASK[%d] MASTER READ SENSOR (BH1750)\n", task_idx);
printf("*****************\n");
printf("data_h: %02x \n", sensor_data_h);
printf("data_l: %02x \n", sensor_data_l);
printf("sensor_val: %.02f [Lux] \n",
    (sensor_data_h <<8 | sensor_data_l) /1.2);
} else {
  ESP_LOGW(TAG, "%s: No ack, sensor not connected
      ...skip...", esp_err_to_name (ret));
xSemaphoreGive(print_mux);
vTaskDelay((DELAY_TIME_BETWEEN_ITEMS_MS
    *(task_idx+1))/portTICK_RATE_MS);

for(i=0; i<DATA_LENGTH; i++) {
    data[i] = i;
}
xSemaphoreTake(print_mux, portMAX_DELAY);
size_t d_size = i2c_slave_write_buffer(I2C_SLAVE_NUM,
    data, RW_TEST_LENGTH, 1000/portTICK_RATE_MS);
if(d_size ==0) {
    ESP_LOGW(TAG, "i2c slave tx buffer full");
    ret = i2c_master_read_slave(I2C_MASTER_NUM, data_rd, DATA_LENGTH);
} else {
    ret = i2c_master_read_slave(I2C_MASTER_NUM, data_rd, RW_TEST_LENGTH);
}

if (ret == ESP_ERR_TIMEOUT) {
  ESP_LOGE(TAG, "I2C timeout");
} else if (ret == ESP_OK) {
printf("*****************\n");
printf("TASK [%d] MASTER READ FROM SLAVE \n", task_idx);
```

```
    printf("*****************\n");
    printf("==== TASK [%d] slave buffer data ====\n", task_idx);
    disp-buf(data, d-size);
    printf("==== TASK[%d] master read ====\n", task-idx);
    disp_buf(data_rd, d-size);
  } else {
    ESP-LOGW(TAG, "TASK[%d] %s: master read slave error,
      IO not connected ...\n", task-idx, esp-err-to-name(ret));
  xSemaphoreGive(print-mux);
  vTaskDelay((DELAY_TIME-BETWEEN_ITEMS_MS * (task_idx+1)) / portTICK_RATE_MS);

  int size;
  for (i=0; i< DATA_LENGTH; i++) {
    data_wr[i] = i+10;
  }

  xSemaphoreTake(print-mux, portMAX_DELAY);
  ret = i2c-master_write_slave(I2C_MASTER_NUM, data_wr, RW-TEST-LENGTH);
  if (ret == ESP_OK) {
    size = i2c_slave_read_buffer(I2C_SLAVE_NUM, data,
        RW_TEST_LENGTH, 1000/portTICK_RATE_MS);
  }

  if (ret == ESP_ERR_TIMEOUT) {
    ESP_LOGE(TAG, "I2C timeout");
  } else if (ret== ESP_OK) {
    printf("*******************\n");
    printf("TASK [%d] MASTER WRITE TO SLAVE\n", task_idx);
    printf("******************\n");
    printf("---- TASK[%d] ~~slave read~~ master write: [%d] bytes ----\n",
        task_idx, ~~size~~);
    disp_buf (data_wr, RW_TEST_LENGTH);
    printf ("---- TASK [%d] slave read: [%d] bytes ----\n",
        task_idx, size);
    disp_buf (data, size);
    }else {
    ESP_LOGW (TAG, "TASK [%d] %s: master write to slave error,
        IO not connected... \n", task_idx, esp_err_to_name (ret));
    }
    xSemaphoreGive (print_mux);
    vTaskDelay (( DELAY_TIME_BETWEEN_ITEMS_MS * (task_idx+1)) /portTICK_RATE_MS);
   }
  vSemaphoreDelete (print_mux);
  vTaskDelete (NULL);
}


void app_main () {
  print_mux = xSemaphoreCreateMutex ();
  ESP_ERROR_CHECK (i2c_slave_init ());
  ESP_ERROR_CHECK (i2c_master_init ());
  xTaskCreate (i2c_test_task, "i2c_test_task_0", 1024*2, (void*)0, 10, NULL);
  xTaskCreate (i2c_test_task, "i2c_test_task_1", 1024*2, (void*)1, 10, NULL);
}
```