```c
/*****************************************************************************
  Copyright (c) 2026 Private Perimeter LLC. All rights reserved.

  PRODUCT HARDWARE License:
  Permission is hereby granted, free of charge, to any person acquiring a copy
  of this software and associated documentation files (the "Software") to use,
  modify, and run the Software solely in conjunction with the genuine FRAM 128K
  hardware manufactured by Private Perimeter LLC.

  Any redistribution, packaging, or incorporation of this Software into a
  competing software product, service, or hardware platform is strictly prohibited
  without the prior written consent of the copyright holder.

  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND...
*****************************************************************************/
#include <stdio.h>
#include <stdbool.h>
#include <unistd.h>
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/i2c_master.h"
#include "driver/gpio.h"

static const char *TAG = "I2C";
static const char *FRAMTAG = "FRAM";

#define FRAM_ADDR_L 0x50
#define FRAM_ADDR_H 0x51
#define FRAM_PAGE_SIZE 0x10000                // 64K

#define I2C_MASTER_SCL_IO           39         // Set this to match your board
#define I2C_MASTER_SDA_IO           40         // Set this to match your board
#define I2C_MASTER_NUM              I2C_NUM_0  // I2C port number for master dev
#define I2C_MASTER_FREQ_HZ          100000     // I2C master clock frequency
#define I2C_MASTER_TIMEOUT_MS       1000

i2c_master_bus_handle_t bus_handle;
// The FRAM needs two device handles for lower and upper banks
i2c_master_dev_handle_t dev_handle_L, dev_handle_H;

/**
 * i2c master initialization
 */
esp_err_t i2c_master_init(i2c_master_bus_handle_t *bus_handle)
{
  esp_err_t err;

  i2c_master_bus_config_t bus_config = {
    .i2c_port = I2C_MASTER_NUM,
    .sda_io_num = I2C_MASTER_SDA_IO,
    .scl_io_num = I2C_MASTER_SCL_IO,
    .clk_source = I2C_CLK_SRC_DEFAULT,
    .glitch_ignore_cnt = 7,
    .flags.enable_internal_pullup = true,
  };

  err = i2c_new_master_bus(&bus_config, bus_handle);
  if (err==ESP_OK) {
    ESP_LOGI(TAG,"i2c_new_master_bus initialized");
  }
  else {
    ESP_LOGE(TAG,"Failed i2c_new_master_bus initialization");
```

```c
  }
  return err;
}  // i2c_master_init


esp_err_t i2c_add_device(i2c_master_bus_handle_t *bus_handle, i2c_master_dev_handle_t *dev_handle, uint8_t dev_address)
{
  esp_err_t err;

  i2c_device_config_t dev_config = {
    .dev_addr_length = I2C_ADDR_BIT_LEN_7,
    .device_address = dev_address,                      // varies with whatever device we are asking to talk with
    .scl_speed_hz = I2C_MASTER_FREQ_HZ,
  };

  err = i2c_master_bus_add_device(*bus_handle, &dev_config, dev_handle);
  if (err==ESP_OK) {
    ESP_LOGI(TAG,"Successful i2c_master_bus_add_device");
  }
  else {
    ESP_LOGE(TAG,"Failed i2c_master_bus_add_device");
  }
  return err;
}  // i2c_add_device

/**
 * @brief  Writes a single byte to any location in the 128KB FRAM memory space.
 * @param  mem_addr: 17-bit internal memory address (0x00000 to 0x1FFFF)
 * @param  data: The byte of data to write
 * @return ESP_OK if successful
 */
esp_err_t FRAM_WriteByte(uint32_t mem_addr, uint8_t data)
{
  esp_err_t err=ESP_OK;

  if (mem_addr > 0x1FFFF) return ESP_FAIL; // Address out of bounds for 128KB

  // Calculate Device Address (Bit 16 of mem_addr dictates the I2C page)
  i2c_master_dev_handle_t dev_handle = (mem_addr & 0x10000) ? dev_handle_H : dev_handle_L;

  // Extract lower 16-bit register address bytes
  uint8_t addr_high = (uint8_t)((mem_addr >> 8) & 0xFF);
  uint8_t addr_low  = (uint8_t)(mem_addr & 0xFF);

  uint8_t write_buf[3] = {addr_high, addr_low, data};
  // writes to whatever bank was determined above
  err = i2c_master_transmit(dev_handle, write_buf, sizeof(write_buf), I2C_MASTER_TIMEOUT_MS / portTICK_PERIOD_MS);
  if (err!=ESP_OK) {
    ESP_LOGE(FRAMTAG, "i2c_master_transmit returned %s", esp_err_to_name(err));
  }
  return err;
}  // FRAM_WriteByte

/**
 * @brief  Reads a byte from any location in the 128KB FRAM using ESP-IDF master driver.
 * @param  mem_addr: 17-bit memory address (0x00000 to 0x1FFFF)
 * @param  p_data: Pointer to variables where byte will be saved
 * @return esp_err_t (ESP_OK if success)
 */
esp_err_t FRAM_ReadByte(uint32_t mem_addr, uint8_t *p_data)
{
```

```c
  if (mem_addr > 0x1FFFF || p_data == NULL) return ESP_ERR_INVALID_ARG;

  // Calculate Device Address (Bit 16 of mem_addr dictates the I2C page)
  i2c_master_dev_handle_t dev_handle = (mem_addr & 0x10000) ? dev_handle_H : dev_handle_L;

  // The register address we want to set up
  uint8_t addr_buf[2];
  addr_buf[0] = (uint8_t)((mem_addr >> 8) & 0xFF);
  addr_buf[1] = (uint8_t)(mem_addr & 0xFF);

  // This built-in IDF call performs a Write (to set register index),
  // issues an implicit Re-Start, and transitions cleanly into a Read sequence.
  return i2c_master_transmit_receive(dev_handle, addr_buf, sizeof(addr_buf), p_data, 1,
I2C_MASTER_TIMEOUT_MS / portTICK_PERIOD_MS);
}

//
// These next routines are for writing and reading many bytes at once but keeping track of what bank we
are in properly
//

/**
 * @brief  Writes a multi-byte buffer to FRAM, handling page boundaries automatically.
 * @param  mem_addr: 17-bit memory address (0x00000 to 0x1FFFF)
 * @param  p_buffer: Pointer to the source data buffer
 * @param  len: Total number of bytes to write
 * @return esp_err_t (ESP_OK if success, or ESP I2C error codes)
 */
esp_err_t FRAM_WriteBuffer(uint32_t mem_addr, const uint8_t *p_buffer, uint32_t len)
{
  if ((mem_addr + len) > 0x20000 || p_buffer == NULL || len == 0) {
    return ESP_ERR_INVALID_ARG;
  }

  uint32_t bytes_written = 0;

  while (bytes_written < len) {
    uint32_t current_addr = mem_addr + bytes_written;

    // 1. Determine active page handle and local offset
    i2c_master_dev_handle_t target_handle = (current_addr & 0x10000) ? dev_handle_H : dev_handle_L;
    uint16_t local_reg_addr = (uint16_t)(current_addr & 0xFFFF);

    // 2. Calculate how many bytes can be written safely before hitting the 64KB page boundary
    uint32_t max_bytes_allowed = FRAM_PAGE_SIZE - local_reg_addr;
    uint32_t chunk_size = len - bytes_written;
    if (chunk_size > max_bytes_allowed) {
      chunk_size = max_bytes_allowed;
    }

    // 3. Allocate a transmission buffer [Addr High, Addr Low, Data0, Data1, ...]
    // Dynamic allocation keeps stack footprint small for large transmissions
    uint8_t *tx_buf = malloc(2 + chunk_size);
    if (tx_buf == NULL) return ESP_ERR_NO_MEM;

    tx_buf[0] = (uint8_t)((local_reg_addr >> 8) & 0xFF);
    tx_buf[1] = (uint8_t)(local_reg_addr & 0xFF);
    memcpy(&tx_buf[2], &p_buffer[bytes_written], chunk_size);

    // 4. Stream data payload to the hardware
    esp_err_t ret = i2c_master_transmit(target_handle, tx_buf, 2 + chunk_size, -1);
    free(tx_buf);

    if (ret != ESP_OK) {
```

```c
      return ret; // Terminate early if bus error occurs
    }

    bytes_written += chunk_size;
  }  // while more bytes

  return ESP_OK;
}  // FRAM_WriteBuffer


/**
 * @brief  Reads a multi-byte buffer from FRAM, handling page boundaries automatically.
 * @param  mem_addr: 17-bit memory address (0x00000 to 0x1FFFF)
 * @param  p_buffer: Pointer to the destination buffer to save data
 * @param  len: Total number of bytes to read
 * @return esp_err_t (ESP_OK if success, or ESP I2C error codes)
 */
esp_err_t FRAM_ReadBuffer(uint32_t mem_addr, uint8_t *p_buffer, uint32_t len)
{
  if ((mem_addr + len) > 0x20000 || p_buffer == NULL || len == 0) {
    return ESP_ERR_INVALID_ARG;
  }

  uint32_t bytes_read = 0;

  while (bytes_read < len) {
    uint32_t current_addr = mem_addr + bytes_read;

    // 1. Determine active page handle and local offset
    i2c_master_dev_handle_t target_handle = (current_addr & 0x10000) ? dev_handle_H : dev_handle_L;
    uint16_t local_reg_addr = (uint16_t)(current_addr & 0xFFFF);

    // 2. Bound checking to make sure we don't roll over past 0xFFFF on this handle
    uint32_t max_bytes_allowed = FRAM_PAGE_SIZE - local_reg_addr;
    uint32_t chunk_size = len - bytes_read;
    if (chunk_size > max_bytes_allowed) {
      chunk_size = max_bytes_allowed;
    }

    // 3. Set up memory register pointers
    uint8_t addr_buf[2];
    addr_buf[0] = (uint8_t)((local_reg_addr >> 8) & 0xFF);
    addr_buf[1] = (uint8_t)(local_reg_addr & 0xFF);

    // 4. Execute standard write-then-read sequence for the current chunk
    esp_err_t ret = i2c_master_transmit_receive(
        target_handle,
        addr_buf, sizeof(addr_buf),
        &p_buffer[bytes_read], chunk_size,
        -1
    );

    if (ret != ESP_OK) {
      return ret;
    }

    bytes_read += chunk_size;
  }  // while more bytes to write

  return ESP_OK;
}  // FRAM_ReadBuffer

//
// FRAM_SingleByteTest
```

```c
//
// Write then read back a single byte from a given address
//
esp_err_t FRAM_SingleByteTest(uint32_t mem_addr, uint8_t testWrite)
{
  esp_err_t err=ESP_OK;
  uint8_t data8;

  ESP_LOGI(FRAMTAG, "FRAM_SingleByteTest adr 0x%x data 0x%x",mem_addr,testWrite);

  ESP_LOGI(FRAMTAG, "Write 0x%x",testWrite);
  err =  FRAM_WriteByte(mem_addr, testWrite);
  if (err==ESP_OK) {
    err =  FRAM_ReadByte(mem_addr, &data8);
    if (err==ESP_OK) {
          if (data8==testWrite) {
        ESP_LOGI(FRAMTAG, "Write/read matches 0x%x==0x%x",testWrite,data8);
          }
          else {
        ESP_LOGE(FRAMTAG, "Write/read mismatch 0x%x <> 0x%x",testWrite,data8);
          }
        }
    else {
      ESP_LOGE(FRAMTAG, "FRAM_ReadByte returned %s",esp_err_to_name(err));
    }
  }
  else {
    ESP_LOGE(FRAMTAG, "FRAM_WriteByte returned %s",esp_err_to_name(err));
  }
  return err;
}  // FRAM_SingleByteTest


//
// fram_test
//
//  Exercise all of our routines and see that we can read and write arbitrarily to this FRAM chip
//
esp_err_t fram_test(void)
{
  esp_err_t err=ESP_OK;
  uint32_t mem_addr;
  #define BUFFER_SIZE 256
  uint8_t writeBuffer[BUFFER_SIZE];
  uint8_t readBuffer[BUFFER_SIZE];

  ESP_LOGI(FRAMTAG, "FRAM lower bank byte test");
  mem_addr=0;
  for (int i=0; (i<10) && (err==ESP_OK); i++) {
        // write and overwrite
    err = FRAM_SingleByteTest(mem_addr+i, 0xA5);
    if (err==ESP_OK)
      err = FRAM_SingleByteTest(mem_addr+i+1, 0x5A);
  }  // for
  if (err==ESP_OK) {
    ESP_LOGI(FRAMTAG, "FRAM upper bank byte test");
    mem_addr=0x10000;
    for (int i=0; (i<10) && (err==ESP_OK); i++) {
          // write and overwrite
      err = FRAM_SingleByteTest(mem_addr+i, 0xA5);
      if (err==ESP_OK)
        err = FRAM_SingleByteTest(mem_addr+i+1, 0x5A);
    }  // for
  }
```

```c
  else {
    ESP_LOGE(FRAMTAG, "FRAM lower bank byte test failed %s",esp_err_to_name(err));
    return err;
  }

  if (err==ESP_OK) {
    ESP_LOGI(FRAMTAG, "FRAM lower bank buffer test");
    for (int k=0; k<BUFFER_SIZE; k++)
      writeBuffer[k]=k;
    // write buffer to an odd arbitrary address
    err = FRAM_WriteBuffer(0x1003, writeBuffer, BUFFER_SIZE);
    if (err==ESP_OK) {
      err = FRAM_ReadBuffer(0x1003, readBuffer,BUFFER_SIZE);
      if (err==ESP_OK) {
        for (int j=0; (j<BUFFER_SIZE) && (err==ESP_OK); j++) {
                  if (writeBuffer[j] != readBuffer[j]) {
                        err = ESP_FAIL;
                  }
                }
      }
    }
  }
  else {
    ESP_LOGE(FRAMTAG, "FRAM lower bank buffer test failed %s",esp_err_to_name(err));
    return err;
  }

  if (err==ESP_OK) {
    ESP_LOGI(FRAMTAG, "FRAM upper bank buffer test");
    for (int k=0; k<BUFFER_SIZE; k++)
      writeBuffer[k]=k;
    // write buffer to an odd arbitrary address
    err = FRAM_WriteBuffer(0x14007, writeBuffer, BUFFER_SIZE);
    if (err==ESP_OK) {
      err = FRAM_ReadBuffer(0x14007, readBuffer,BUFFER_SIZE);
      if (err==ESP_OK) {
        for (int j=0; (j<BUFFER_SIZE) && (err==ESP_OK); j++) {
                  if (writeBuffer[j] != readBuffer[j]) {
                        err = ESP_FAIL;
                  }
                }
      }
    }
  }
  else {
    ESP_LOGE(FRAMTAG, "FRAM upper bank buffer test failed %s",esp_err_to_name(err));
    return err;
  }

  if (err==ESP_OK) {
    ESP_LOGI(FRAMTAG, "FRAM span bank buffer test");
    for (int k=0; k<BUFFER_SIZE; k++)
      writeBuffer[k]=k;
    // write buffer to an odd arbitrary address near the page boundary so that we exceed it into the next page
    err = FRAM_WriteBuffer(0x7fe0, writeBuffer, BUFFER_SIZE);
    if (err==ESP_OK) {
      err = FRAM_ReadBuffer(0x7fe0, readBuffer,BUFFER_SIZE);
      if (err==ESP_OK) {
        for (int j=0; (j<BUFFER_SIZE) && (err==ESP_OK); j++) {
                  if (writeBuffer[j] != readBuffer[j]) {
                        err = ESP_FAIL;
                  }
                }
```

```
      }
    }
  }
  else {
    ESP_LOGE(FRAMTAG, "FRAM span bank buffer test failed %s",esp_err_to_name(err));
    return err;
  }

  // always dump values so that we can test after writing and also after a power down reset
  err = FRAM_ReadBuffer(0x0000, readBuffer,32);
  ESP_LOGI(FRAMTAG, "addr 0x0000");
  ESP_LOG_BUFFER_HEXDUMP(TAG, readBuffer, 32, ESP_LOG_INFO);
  err = FRAM_ReadBuffer(0x1003, readBuffer,BUFFER_SIZE);
  ESP_LOGI(FRAMTAG, "addr 0x1003");
  ESP_LOG_BUFFER_HEXDUMP(TAG, readBuffer, BUFFER_SIZE, ESP_LOG_INFO);
  err = FRAM_ReadBuffer(0x7fe0, readBuffer,32);
  ESP_LOGI(FRAMTAG, "addr 0x7fe0");
  ESP_LOG_BUFFER_HEXDUMP(TAG, readBuffer, 32, ESP_LOG_INFO);
  err = FRAM_ReadBuffer(0x10000, readBuffer,32);
  ESP_LOGI(FRAMTAG, "addr 0x10000");
  ESP_LOG_BUFFER_HEXDUMP(TAG, readBuffer, 32, ESP_LOG_INFO);
  err = FRAM_ReadBuffer(0x14007, readBuffer,BUFFER_SIZE);
  ESP_LOGI(FRAMTAG, "addr 0x14007");
  ESP_LOG_BUFFER_HEXDUMP(TAG, readBuffer, BUFFER_SIZE, ESP_LOG_INFO);

  if (err==ESP_OK) {
    ESP_LOGI(FRAMTAG, "Clear FRAM back to 0s");
    for (int k=0; k<BUFFER_SIZE; k++)
      writeBuffer[k]=0;
    // write a buffer at a time until we write to the entire FRAM
    for (int m=0; (m<(0x20000/BUFFER_SIZE)) && (err==ESP_OK); m++) {
         if (m%32==0) {
       printf("Clearing offset 0x%x\r\n",m*BUFFER_SIZE);
         }
      // write buffer to an odd arbitrary address near the page boundary so that we exceed it into the
next page
      err = FRAM_WriteBuffer(m * BUFFER_SIZE, writeBuffer, BUFFER_SIZE);
      if (err==ESP_OK) {
        err = FRAM_ReadBuffer(m * BUFFER_SIZE, readBuffer,BUFFER_SIZE);
        if (err==ESP_OK) {
          for (int j=0; (j<BUFFER_SIZE) && (err==ESP_OK); j++) {
               if (writeBuffer[j] != readBuffer[j]) {
                        err = ESP_FAIL;
                   }
                 }
               }
      }
    }  // for each buffer of data
  }

  if (err != ESP_OK) {
    ESP_LOGE(FRAMTAG, "FRAM test failed %s",esp_err_to_name(err));
  }

  return err;
}  // fram_test



void app_main(void)
{
  esp_err_t err;

  printf("Hello world!\n");
```

```
441
442   err = i2c_master_init(&bus_handle);
443   if (err!=ESP_OK) {
444     ESP_LOGE(TAG, "i2c_master_init returned %s",esp_err_to_name(err));
445   }
446   else {
447     ESP_LOGI(TAG, "I2C initialized successfully");
448
449     // device handle for the FRAM low bank
450     err = i2c_add_device(&bus_handle, &dev_handle_L, FRAM_ADDR_L);
451     if (err!=ESP_OK) {
452       ESP_LOGE(TAG, "i2c_add_device returned %s",esp_err_to_name(err));
453     }
454     else {
455           // we need a second device handle for the FRAM high bank
456       err = i2c_add_device(&bus_handle, &dev_handle_H, FRAM_ADDR_H);
457       if (err!=ESP_OK) {
458         ESP_LOGE(TAG, "i2c_add_device returned %s",esp_err_to_name(err));
459       }
460       else {
461         err = fram_test();
462         if (err==ESP_OK) {
463           ESP_LOGI(TAG, "FRAM 128K test passed");
464                 }
465
466         err = i2c_master_bus_rm_device(dev_handle_L);
467         if (err!=ESP_OK) {
468           ESP_LOGE(TAG, "i2c_master_bus_rm_device returned %s for handle 1",esp_err_to_name(err));
469         }
470         else {
471           ESP_LOGI(TAG, "i2c_master_bus_rm_device OK");
472         }
473
474         err = i2c_master_bus_rm_device(dev_handle_H);
475         if (err!=ESP_OK) {
476           ESP_LOGE(TAG, "i2c_master_bus_rm_device returned %s for handle 2",esp_err_to_name(err));
477         }
478         else {
479           ESP_LOGI(TAG, "i2c_master_bus_rm_device OK");
480         }
481       }
482     }  // else run FRAM tests
483
484     // now shutdown I2C altogether
485     err = i2c_del_master_bus(bus_handle);
486     if (err!=ESP_OK) {
487       ESP_LOGE(TAG, "i2c_del_master_bus returned %s",esp_err_to_name(err));
488     }
489     else {
490       ESP_LOGI(TAG, "I2C de-initialized successfully");
491     }
492   }  // I2C initialized
493 }  // app_main
494
```