ECE 4999 Electromechanical Lock

1.2

Generated by Doxygen 1.8.20

Chapter 1

ECE 4999 Electromechanical Lock

ESP32 based lock that features RFID and keypad to unlock

Author

Matthew Hait

Remington Davids

Version

1.2

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

File Documentation

3.1 G:/ESP-IDF/projects/esp_4999_Main/main/CMakeLists.txt File Reference

3.2 G:/ESP-IDF/projects/esp_4999_Main/main/main.c File Reference

```
#include <stdio.h>
#include "rc522.c"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/queue.h"
#include "esp_system.h"
#include "driver/adc.h"
#include "esp_adc_cal.h"
#include "nvs_flash.h"
#include "nvs.h"
#include <string.h>
#include "driver/ledc.h"
#include "driver/mcpwm.h"
#include "soc/mcpwm_periph.h"
```

Include dependency graph for main.c:

Macros

- #define ESP_INTR_FLAG_LEVEL 0

  Interupt vector level ref: *https://github.com/anoochit/esp32-example/blob/master/2080_G↩ PIO_Interrupt/main/gpio_intr.c*.

- #define MULTISAMPLING (50)

  ADC samples to collect for keypad

- #define defaultKEY 1234

  Default key pin from factory. Randomize on every product.

- #define serialLen (5)

  Length of RFID serial

- #define GPIO_KEYPAD (GPIO_SEL_34)

  Keypad IO Pin

- #define GPIO_KEYPAD_IO 34

  Keypad IO Pin

- #define GPIO_BATV (GPIO_SEL_35)

  Battery Monitor IO Pin

- #define GPIO_BATV_IO 35

  Battery Monitor IO Pin

- #define KEYPAD_BUF_SZ (17)

  Max pin size is KEYPAD_BUF_SZ - 1.
- #define LEDC_HS_TIMER LEDC_TIMER_0

  HW RTC for RGB control.
- #define LEDC_HS_MODE LEDC_HIGH_SPEED_MODE

  RGB PWM timing mode.
- #define LEDC_HS_RED_GPIO (18)

  Status RGB RED output pin.
- #define LEDC_HS_RED_CHANNEL LEDC_CHANNEL_0

  Status RGB RED LEDC channel.
- #define RED 0
- #define LEDC_HS_GREEN_GPIO (19)

  Status RGB GREEN output pin.
- #define LEDC_HS_GREEN_CHANNEL LEDC_CHANNEL_1

  Status RGB GREEN LEDC channel.
- #define GREEN 1
- #define LEDC_HS_BLUE_GPIO (20)

  Status RGB BLUE output pin.
- #define LEDC_HS_BLUE_CHANNEL LEDC_CHANNEL_2

  Status RGB BLUE LEDC channel.
- #define Blue 2
- #define LEDC_TEST_CH_NUM 3

  Status RGB channel quantity.
- #define LEDC_TEST_DUTY (4000)

  Duty cycle for quick LED flash.
- #define MTR_PWM0A_OUT 15

  PWM H-Bridge Input A.
- #define MTR_PWM0B_OUT 16

PWM H-Bridge Input B.

- #define MTR_Sleep (GPIO_SEL_33)

  PWM H-Bridge Sleep Control.
- #define MTR_SleepC (GPIO_NUM_33)

  PWM H-Bridge Sleep Control.
- #define MTR_PWM_FREQ 1000

  PWM H-Bridge PWM Frequency.
- #define MTR_PERIOD 3000

  PWM H-Bridge lock cycle time.
- #define MTR_duty 100

  PWM H-Bridge duty cycle.

Functions

- void pairTagHandler (uint8_t ∗serial_no)

  Callback function for rc522 when pairing new NFC tags.
- void tag_handler (uint8_t ∗serial_no)

  Callback function for rc522 library under normal operation.
- void toggle_Lock ()

  Handles motor control and position logic.
- void unpairNFC ()

  Removes NFC keys from heap and NVS.
- void writePinNVS (uint64_t secret)

  Stores keypad pin into non-voltile storage.
- unsigned long runTime ()

  Grabs FreeRTOS runtime to determine timeouts.
- void batCheck ()

  Flashes LED if battery is low.
- static void IRAM_ATTR gpio_isr_handler (void ∗arg)

  Keypad GPIO xQueue translate.
- static void keypadCallback (void ∗arg)

  Keypad GPIO interupt handler.
- static void keypadPINCallback (void ∗arg)

  Keypad GPIO interupt handler for setting new pin.
- void initGPIO ()

  Initialize GPIO pins. Ref: `https://github.com/espressif/esp-idf/blob/master/examples/peripherals/gpio/_gpio/main/gpio_example_main.c`.
- void init_NVS ()

  Initialize NVS; pulls key qty and pin from storage into memory. Ref: `https://github.com/espressif/esp-idf/tree/1d706_rw_value`.
- void grabKeyList (uint8_t ∗buf, int count)

  Grabs NFC keylist from NVS.
- void writeNFCKey (uint8_t ∗buf, int count)

  Writes NFC key to NVS. Only writes one key, not all NFC keys at once. (For use in pairing new NFC Tag)
- void writeKeyQtyNVS (uint8_t qty)

  Stores NFC key qty into non-voltile storage.
- void app_main (void)

Variables

- uint8_t storedKey_QTY = 0

  QTY of stored NFC keys

- uint64_t keypadPin = 0

  Secret pin, shhhhhhhh

- bool keyPinValid = false

  Logic check to insure password cannot be entered from corrupted memory

- esp_adc_cal_characteristics_t characteristics
- static xQueueHandle gpio_evt_queue = NULL

  FreeRTOS queue for keypad events.
- bool mtrForward = true

  Motor direction control. Default Forward

- int thresholds [12] = {195, 305, 375, 405, 447, 477, 491, 512, 530, 538, 551, 563}

  ADC values for keypad reference.
- char keypad [12] = {'1', '2', '3', '4', '5', '6', '7', '8', '9', '∗', '0', '#'}

  Translated keypad values.
- char keypadInputBuffer [KEYPAD_BUF_SZ]

  Buffer storage for keypad input.
- int keypadInputCnt = 0

  Keypad buffer index.
- unsigned long keypadLastUpdate = 0

  Last keypad input tick [ms].
- const unsigned long keypadInvalidAfter = 5000

  Keypad timeout period [ms].
- const unsigned long pairInvalidAfter = 5000

  NFC pair timeout period [ms].
- bool nfcEnable = true

  NFC enabler.
- uint8_t ∗ keyList [serialLen]

  NFC key list.
- ledc_channel_config_t ledc_channel [LEDC_TEST_CH_NUM]

  Status RGB LEDC config.
- const rc522_start_args_t pairNFC_args

  rc522 config when pairing NFC tags.
- const rc522_start_args_t running_args

  rc522 config when under normal operation.
- mcpwm_config_t pwm_config

  H-Bridge Motor config.
- bool noPairDect = true

  Poorly designed timeout bypass for NFC pair.
- TaskHandle_t xPinHandle

  FreeRTOS task handle for changing xqueue on keypad.
- TaskHandle_t xKeypadHandle

  FreeRTOS task handle for normal xqueue on keypad.

3.2.1 Macro Definition Documentation

3.2.1.1 Blue

```
#define Blue 2
```

Definition at line 54 of file main.c.

3.2.1.2 defaultKEY

```
#define defaultKEY 1234
```

Default key pin from factory. Randomize on every product.

Definition at line 22 of file main.c.

3.2.1.3 ESP_INTR_FLAG_LEVEL

```
#define ESP_INTR_FLAG_LEVEL 0
```

Interupt vector level ref: https://github.com/anoochit/esp32-example/blob/master/2080_GPIO_Interrupt/main/gpio_intr.c.

Definition at line 18 of file main.c.

3.2.1.4 GPIO_BATV

```
#define GPIO_BATV (GPIO_SEL_35)
```

Battery Monitor IO Pin

Definition at line 31 of file main.c.

3.2.1.5 GPIO_BATV_IO

```
#define GPIO_BATV_IO 35
```

Battery Monitor IO Pin

Definition at line 33 of file main.c.

3.2.1.6 GPIO_KEYPAD

```
#define GPIO_KEYPAD (GPIO_SEL_34)
```

Keypad IO Pin

Definition at line 26 of file main.c.

3.2.1.7 GPIO_KEYPAD_IO

```
#define GPIO_KEYPAD_IO 34
```

Keypad IO Pin

Definition at line 28 of file main.c.

3.2.1.8 GREEN

```
#define GREEN 1
```

Definition at line 49 of file main.c.

3.2.1.9 KEYPAD_BUF_SZ

```
#define KEYPAD_BUF_SZ (17)
```

Max pin size is KEYPAD_BUF_SZ - 1.

Definition at line 35 of file main.c.

3.2.1.10 LEDC_HS_BLUE_CHANNEL

```
#define LEDC_HS_BLUE_CHANNEL LEDC_CHANNEL_2
```

Status RGB BLUE LEDC channel.

Definition at line 53 of file main.c.

3.2.1.11 LEDC_HS_BLUE_GPIO

```
#define LEDC_HS_BLUE_GPIO (20)
```

Status RGB BLUE output pin.

Definition at line 51 of file main.c.

3.2.1.12 LEDC_HS_GREEN_CHANNEL

```
#define LEDC_HS_GREEN_CHANNEL LEDC_CHANNEL_1
```

Status RGB GREEN LEDC channel.

Definition at line 48 of file main.c.

3.2.1.13 LEDC_HS_GREEN_GPIO

```
#define LEDC_HS_GREEN_GPIO (19)
```

Status RGB GREEN output pin.

Definition at line 46 of file main.c.

3.2.1.14 LEDC_HS_MODE

```
#define LEDC_HS_MODE LEDC_HIGH_SPEED_MODE
```

RGB PWM timing mode.

Definition at line 39 of file main.c.

3.2.1.15 LEDC_HS_RED_CHANNEL

```
#define LEDC_HS_RED_CHANNEL LEDC_CHANNEL_0
```

Status RGB RED LEDC channel.

Definition at line 43 of file main.c.

3.2.1.16 LEDC_HS_RED_GPIO

```
#define LEDC_HS_RED_GPIO (18)
```

Status RGB RED output pin.

Definition at line 41 of file main.c.

3.2.1.17 LEDC_HS_TIMER

```
#define LEDC_HS_TIMER LEDC_TIMER_0
```

HW RTC for RGB control.

Definition at line 37 of file main.c.

3.2.1.18 LEDC_TEST_CH_NUM

```
#define LEDC_TEST_CH_NUM 3
```

Status RGB channel quantity.

Definition at line 56 of file main.c.

3.2.1.19 LEDC_TEST_DUTY

```
#define LEDC_TEST_DUTY (4000)
```

Duty cycle for quick LED flash.

Definition at line 58 of file main.c.

3.2.1.20 MTR_duty

```
#define MTR_duty 100
```

PWM H-Bridge duty cycle.

Definition at line 72 of file main.c.

3.2.1.21 MTR_PERIOD

```
#define MTR_PERIOD 3000
```

PWM H-Bridge lock cycle time.

Definition at line 70 of file main.c.

3.2.1.22 MTR_PWM0A_OUT

```
#define MTR_PWM0A_OUT 15
```

PWM H-Bridge Input A.

Definition at line 60 of file main.c.

3.2.1.23 MTR_PWM0B_OUT

```
#define MTR_PWM0B_OUT 16
```

PWM H-Bridge Input B.

Definition at line 62 of file main.c.

3.2.1.24 MTR_PWM_FREQ

```
#define MTR_PWM_FREQ 1000
```

PWM H-Bridge PWM Frequency.

Definition at line 68 of file main.c.

3.2.1.25 MTR_Sleep

```
#define MTR_Sleep (GPIO_SEL_33)
```

PWM H-Bridge Sleep Control.

Definition at line 64 of file main.c.

3.2.1.26 MTR_SleepC

```
#define MTR_SleepC (GPIO_NUM_33)
```

PWM H-Bridge Sleep Control.

Definition at line 66 of file main.c.

3.2.1.27 MULTISAMPLING

```
#define MULTISAMPLING (50)
```

ADC samples to collect for keypad

Definition at line 20 of file main.c.

3.2.1.28 RED

```
#define RED 0
```

Definition at line 44 of file main.c.

3.2.1.29 serialLen

```
#define serialLen (5)
```

Length of RFID serial

Definition at line 24 of file main.c.

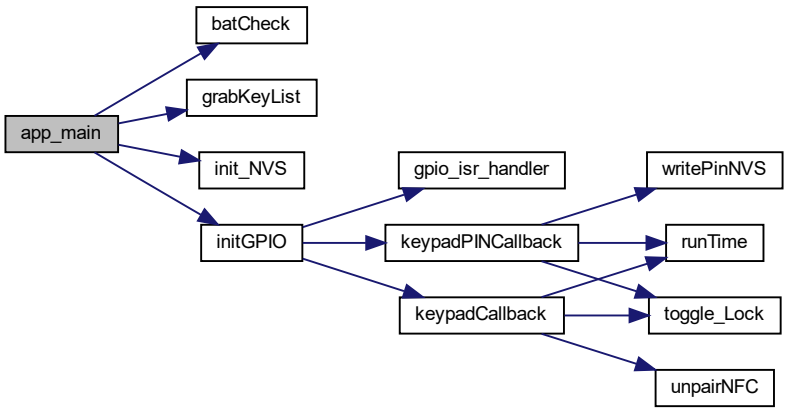
3.2.2 Function Documentation

3.2.2.1 app_main()

```
void app_main (
            void  )
```

Definition at line 846 of file main.c.

Here is the call graph for this function:



3.2.2.2 batCheck()

```
void batCheck ( )
```

Flashes LED if battery is low.

Definition at line 196 of file main.c.

Here is the caller graph for this function:

3.2.2.3 gpio_isr_handler()

```
static void IRAM_ATTR gpio_isr_handler (
            void * arg )  [static]
```

Keypad GPIO xQueue translate.

Parameters

∗arg	Optional switch on GPIO pin.

Definition at line 222 of file main.c.

Here is the caller graph for this function:

3.2.2.4 grabKeyList()

```
void grabKeyList (
            uint8_t * buf,
            int count )
```

Grabs NFC keylist from NVS.

Parameters

∗buf	Key Output
count	Key Index

Definition at line 627 of file main.c.

Here is the caller graph for this function:

3.2.2.5 init_NVS()

```
void init_NVS ( )
```

Initialize NVS; pulls key qty and pin from storage into memory. Ref: https://github.com/espressif/esp-idf/tree/1d7
_rw_value.

Definition at line 543 of file main.c.

Here is the caller graph for this function:

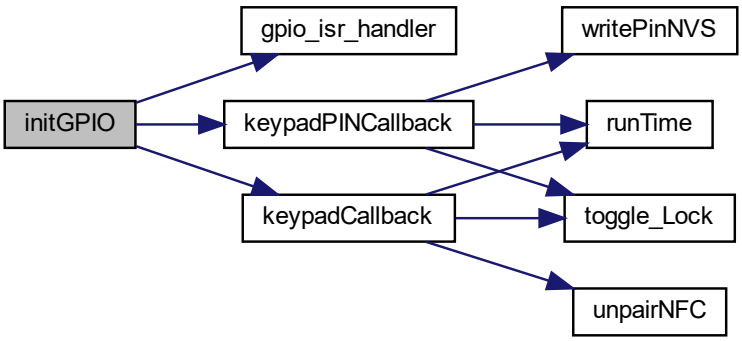
3.2.2.6 initGPIO()

```
void initGPIO ( )
```

Initialize GPIO pins. Ref: https://github.com/espressif/esp-idf/blob/master/examples/peripherals/gp
_gpio/main/gpio_example_main.c.

Definition at line 466 of file main.c.

Here is the call graph for this function:



Here is the caller graph for this function:

3.2.2.7 keypadCallback()

```
static void keypadCallback (
            void * arg )  [static]
```

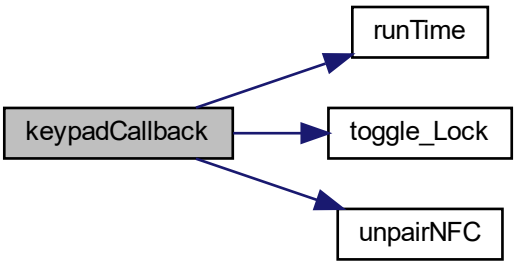
Keypad GPIO interupt handler.

Parameters

∗arg	Optional switch on GPIO pin.

Definition at line 232 of file main.c.

Here is the call graph for this function:



Here is the caller graph for this function:

3.2.2.8 keypadPINCallback()

```
static void keypadPINCallback (
            void * arg )  [static]
```

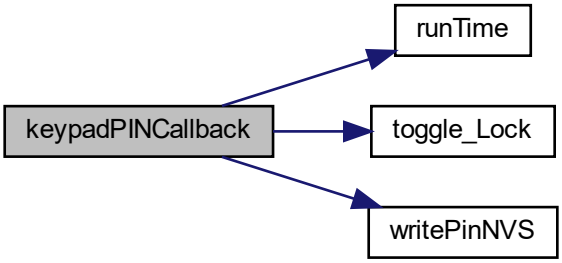
Keypad GPIO interupt handler for setting new pin.

Parameters

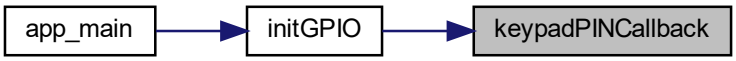
∗arg	Optional switch on GPIO pin.

Definition at line 398 of file main.c.

Here is the call graph for this function:



Here is the caller graph for this function:



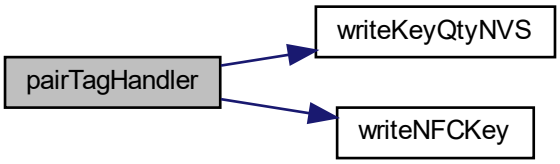
3.2.2.9 pairTagHandler()

```
void pairTagHandler (
            uint8_t * serial_no )
```

Callback function for rc522 when pairing new NFC tags.

Definition at line 811 of file main.c.

Here is the call graph for this function:



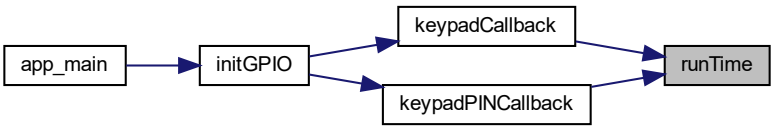
3.2.2.10 runTime()

```
unsigned long runTime ( )
```

Grabs FreeRTOS runtime to determine timeouts.

Definition at line 191 of file main.c.

Here is the caller graph for this function:



3.2.2.11 tag_handler()

```
void tag_handler (
            uint8_t * serial_no )
```

Callback function for rc522 library under normal operation.

Definition at line 781 of file main.c.

Here is the call graph for this function:

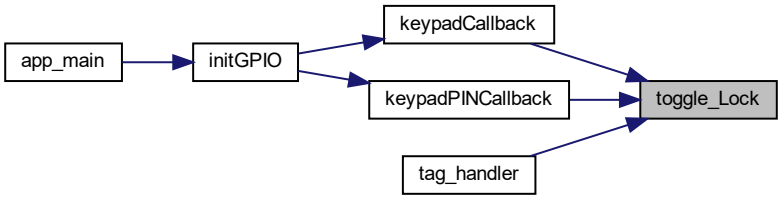
3.2.2.12 toggle_Lock()

```
void toggle_Lock ( )
```

Handles motor control and position logic.

Definition at line 825 of file main.c.

Here is the caller graph for this function:



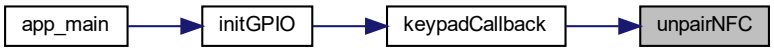
3.2.2.13 unpairNFC()

```
void unpairNFC ( )
```

Removes NFC keys from heap and NVS.

Definition at line 714 of file main.c.

Here is the caller graph for this function:



3.2.2.14 writeKeyQtyNVS()

```
void writeKeyQtyNVS (
            uint8_t qty )
```

Stores NFC key qty into non-voltile storage.

Parameters

qty	Value to be stored

Definition at line 758 of file main.c.

Here is the caller graph for this function:

3.2.2.15 writeNFCKey()

```
void writeNFCKey (
            uint8_t * buf,
            int count )
```

Writes NFC key to NVS. Only writes one key, not all NFC keys at once. (For use in pairing new NFC Tag)

Parameters

∗buf	Key Input
count	Key Index

Definition at line 677 of file main.c.

Here is the caller graph for this function:

3.2.2.16 writePinNVS()

```
void writePinNVS (
            uint64_t secret )
```

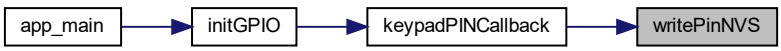
Stores keypad pin into non-voltile storage.

Parameters

secret	Input pin to be stored

Definition at line 601 of file main.c.

Here is the caller graph for this function:



3.2.3 Variable Documentation

3.2.3.1 characteristics

```
esp_adc_cal_characteristics_t characteristics
```

Definition at line 81 of file main.c.

3.2.3.2 gpio_evt_queue

```
xQueueHandle gpio_evt_queue = NULL  [static]
```

FreeRTOS queue for keypad events.

Definition at line 83 of file main.c.

3.2.3.3 keyList

```
uint8_t* keyList[serialLen]
```

NFC key list.

Definition at line 103 of file main.c.

3.2.3.4 keypad

```
char keypad[12] = {'1', '2', '3', '4', '5', '6', '7', '8', '9', '*', '0', '#'}
```

Translated keypad values.

Definition at line 89 of file main.c.

3.2.3.5 keypadInputBuffer

```
char keypadInputBuffer[KEYPAD_BUF_SZ]
```

Buffer storage for keypad input.

Definition at line 91 of file main.c.

3.2.3.6 keypadInputCnt

```
int keypadInputCnt = 0
```

Keypad buffer index.

Definition at line 93 of file main.c.

3.2.3.7 keypadInvalidAfter

```
const unsigned long keypadInvalidAfter = 5000
```

Keypad timeout period [ms].

Definition at line 97 of file main.c.

3.2.3.8 keypadLastUpdate

```
unsigned long keypadLastUpdate = 0
```

Last keypad input tick [ms].

Definition at line 95 of file main.c.

3.2.3.9 keypadPin

```
uint64_t keypadPin = 0
```

Secret pin, shhhhhhhh

Definition at line 78 of file main.c.

3.2.3.10 keyPinValid

```
bool keyPinValid = false
```

Logic check to insure password cannot be entered from corrupted memory

Definition at line 80 of file main.c.

3.2.3.11 ledc_channel

```
ledc_channel_config_t ledc_channel[LEDC_TEST_CH_NUM]
```

Initial value:

```
= {
    {
        .channel    = LEDC_HS_RED_CHANNEL,
        .duty       = 0,
        .gpio_num   = LEDC_HS_RED_GPIO,
        .speed_mode = LEDC_HS_MODE,
        .hpoint     = 0,
        .timer_sel  = LEDC_HS_TIMER
    },
    {
        .channel    = LEDC_HS_GREEN_CHANNEL,
        .duty       = 0,
        .gpio_num   = LEDC_HS_GREEN_GPIO,
        .speed_mode = LEDC_HS_MODE,
        .hpoint     = 0,
        .timer_sel  = LEDC_HS_TIMER
    },
    {
        .channel    = LEDC_HS_BLUE_CHANNEL,
        .duty       = 0,
        .gpio_num   = LEDC_HS_BLUE_GPIO,
        .speed_mode = LEDC_HS_MODE,
        .hpoint     = 0,
        .timer_sel  = LEDC_HS_TIMER
    }
}
```

Status RGB LEDC config.

Definition at line 105 of file main.c.

3.2.3.12 mtrForward

```
bool mtrForward = true
```

Motor direction control. Default Forward

Definition at line 85 of file main.c.

3.2.3.13 nfcEnable

```
bool nfcEnable = true
```

NFC enabler.

Definition at line 101 of file main.c.

3.2.3.14 noPairDect

```
bool noPairDect = true
```

Poorly designed timeout bypass for NFC pair.

Definition at line 159 of file main.c.

3.2.3.15 pairInvalidAfter

```
const unsigned long pairInvalidAfter = 5000
```

NFC pair timeout period [ms].

Definition at line 99 of file main.c.

3.2.3.16 pairNFC_args

```
const rc522_start_args_t pairNFC_args
```

Initial value:

```
= {
    .miso_io = 25,
    .mosi_io = 23,
    .sck_io = 19,
    .sda_io = 22,
    .callback = &pairTagHandler
}
```

rc522 config when pairing NFC tags.

Definition at line 134 of file main.c.

3.2.3.17 pwm_config

```
mcpwm_config_t pwm_config
```

Initial value:

```
= {
    .frequency = MTR_PWM_FREQ,
    .cmpr_a = 0,
    .cmpr_b = 0,
    .counter_mode = MCPWM_UP_COUNTER,
    .duty_mode = MCPWM_DUTY_MODE_0
}
```

H-Bridge Motor config.

Definition at line 151 of file main.c.

3.2.3.18 running_args

```
const rc522_start_args_t running_args
```

Initial value:

```
= {
    .miso_io = 25,
    .mosi_io = 23,
    .sck_io = 19,
    .sda_io = 22,
    .callback = &tag_handler
}
```

rc522 config when under normal operation.

Definition at line 143 of file main.c.

3.2.3.19 storedKey_QTY

```
uint8_t storedKey_QTY = 0
```

QTY of stored NFC keys

Definition at line 76 of file main.c.

3.2.3.20 thresholds

```
int thresholds[12] = {195, 305, 375, 405, 447, 477, 491, 512, 530, 538, 551, 563}
```

ADC values for keypad reference.

Definition at line 87 of file main.c.

3.2.3.21 xKeypadHandle

```
TaskHandle_t xKeypadHandle
```

FreeRTOS task handle for normal xqueue on keypad.

Definition at line 163 of file main.c.

3.2.3.22 xPinHandle

```
TaskHandle_t xPinHandle
```

FreeRTOS task handle for changing xqueue on keypad.

Definition at line 161 of file main.c.

3.3 main.c

```
00001 #include <stdio.h>
00002 #include "rc522.c"
00003 #include "freertos/FreeRTOS.h"
00004 #include "freertos/task.h"
00005 #include "freertos/queue.h"
00006 #include "esp_system.h"
00007 #include "driver/adc.h"
00008 #include "esp_adc_cal.h"
00009 #include "nvs_flash.h"
00010 #include "nvs.h"
00011 #include <string.h>
00012 #include "driver/ledc.h"
00013 #include "driver/mcpwm.h"
00014 #include "soc/mcpwm_periph.h"
00015
00016
00018 #define ESP_INTR_FLAG_LEVEL 0
00019
00020 #define MULTISAMPLING       (50)
00021
00022 #define defaultKEY  1234
00023
00024 #define serialLen   (5)
00025
00026 #define GPIO_KEYPAD (GPIO_SEL_34)
00027
00028 #define GPIO_KEYPAD_IO  34
00029
00030 //Might need to move to ADC 2 if confict with Keypad trigger
00031 #define GPIO_BATV   (GPIO_SEL_35)
00032
00033 #define GPIO_BATV_IO    35
00034
00035 #define KEYPAD_BUF_SZ   (17)
00036
00037 #define LEDC_HS_TIMER          LEDC_TIMER_0
00038
00039 #define LEDC_HS_MODE           LEDC_HIGH_SPEED_MODE
00040
00041 #define LEDC_HS_RED_GPIO       (18)
00042
00043 #define LEDC_HS_RED_CHANNEL    LEDC_CHANNEL_0
00044 #define RED                     0
00045
00046 #define LEDC_HS_GREEN_GPIO     (19)
00047
00048 #define LEDC_HS_GREEN_CHANNEL  LEDC_CHANNEL_1
00049 #define GREEN                   1
00050
00051 #define LEDC_HS_BLUE_GPIO      (20)
00052
00053 #define LEDC_HS_BLUE_CHANNEL   LEDC_CHANNEL_2
00054 #define Blue                    2
00055
00056 #define LEDC_TEST_CH_NUM       3
00057
00058 #define LEDC_TEST_DUTY         (4000)
00059
00060 #define MTR_PWM0A_OUT 15
00061
00062 #define MTR_PWM0B_OUT 16
00063
00064 #define MTR_Sleep (GPIO_SEL_33)
```

```
00065
00066 #define MTR_SleepC (GPIO_NUM_33)
00067
00068 #define MTR_PWM_FREQ 1000
00069
00070 #define MTR_PERIOD 3000
00071
00072 #define MTR_duty 100
00073     //If falling trigger doesn't work with ADC, attach pin to keypad circuit.
00074     //#define GPIO_SENSE    (GPIO_SEL_39)
00076 uint8_t storedKey_QTY = 0;
00078 uint64_t keypadPin = 0;
00080 bool keyPinValid = false;
00081 esp_adc_cal_characteristics_t characteristics;
00083 static xQueueHandle gpio_evt_queue = NULL;
00085 bool mtrForward = true;
00087 int thresholds[12] = {195, 305, 375, 405, 447, 477, 491, 512, 530, 538, 551, 563};
00089 char keypad[12] = {'1', '2', '3', '4', '5', '6', '7', '8', '9', '*', '0', '#'};
00091 char keypadInputBuffer[KEYPAD_BUF_SZ];
00093 int keypadInputCnt = 0;
00095 unsigned long keypadLastUpdate = 0;
00097 const unsigned long keypadInvalidAfter = 5000;
00099 const unsigned long pairInvalidAfter = 5000;
00101 bool nfcEnable = true;
00103 uint8_t *keyList[serialLen];
00105 ledc_channel_config_t ledc_channel[LEDC_TEST_CH_NUM] = {
00106     {
00107         .channel    = LEDC_HS_RED_CHANNEL,
00108         .duty       = 0,
00109         .gpio_num   = LEDC_HS_RED_GPIO,
00110         .speed_mode = LEDC_HS_MODE,
00111         .hpoint     = 0,
00112         .timer_sel  = LEDC_HS_TIMER
00113     },
00114     {
00115         .channel    = LEDC_HS_GREEN_CHANNEL,
00116         .duty       = 0,
00117         .gpio_num   = LEDC_HS_GREEN_GPIO,
00118         .speed_mode = LEDC_HS_MODE,
00119         .hpoint     = 0,
00120         .timer_sel  = LEDC_HS_TIMER
00121     },
00122     {
00123         .channel    = LEDC_HS_BLUE_CHANNEL,
00124         .duty       = 0,
00125         .gpio_num   = LEDC_HS_BLUE_GPIO,
00126         .speed_mode = LEDC_HS_MODE,
00127         .hpoint     = 0,
00128         .timer_sel  = LEDC_HS_TIMER
00129     }
00130 };
00131
00132 void pairTagHandler(uint8_t* serial_no);
00134 const rc522_start_args_t pairNFC_args = {
00135     .miso_io = 25,
00136     .mosi_io = 23,
00137     .sck_io = 19,
00138     .sda_io = 22,
00139     .callback = &pairTagHandler
00140 };
00141 void tag_handler(uint8_t* serial_no);
00143 const rc522_start_args_t running_args = {
00144     .miso_io = 25,
00145     .mosi_io = 23,
00146     .sck_io = 19,
00147     .sda_io = 22,
00148     .callback = &tag_handler
00149 };
00151 mcpwm_config_t pwm_config = {
00152     .frequency = MTR_PWM_FREQ,
00153     .cmpr_a = 0,
00154     .cmpr_b = 0,
00155     .counter_mode = MCPWM_UP_COUNTER,
00156     .duty_mode = MCPWM_DUTY_MODE_0
00157 };
00159 bool noPairDect = true;
00161 TaskHandle_t xPinHandle;
00163 TaskHandle_t xKeypadHandle;
00164
00165
00177 void toggle_Lock();
```

```
00181 void unpairNFC();
00182
00188 void writePinNVS(uint64_t secret);
00189
00191 unsigned long runTime() {
00192     return xTaskGetTickCount() * portTICK_PERIOD_MS;
00193 }
00194
00196 void batCheck() {
00197     TickType_t xLastWakeTime;
00198     const TickType_t xFrequency = 5000 / portTICK_PERIOD_MS;
00199     xLastWakeTime = xTaskGetTickCount ();
00200     uint32_t adc1_gpio35 = 0;
00201     while(true) {
00202         vTaskDelayUntil( &xLastWakeTime, xFrequency );
00203         for(int i = 0; i < MULTISAMPLING; i++) {
00204             adc1_gpio35 = adc1_get_raw(ADC1_CHANNEL_7);
00205         }
00206         adc1_gpio35 /= MULTISAMPLING;
00207         // Read ADC as mV
00208         uint32_t rightDampVoltage = esp_adc_cal_raw_to_voltage(adc1_gpio35, &characteristics);
00209         if (rightDampVoltage < 3000) {
00210             ledc_set_duty(ledc_channel[RED].speed_mode, ledc_channel[RED].channel, LEDC_TEST_DUTY);
00211             ledc_update_duty(ledc_channel[RED].speed_mode, ledc_channel[RED].channel);
00212             vTaskDelay(125 / portTICK_PERIOD_MS);
00213             ledc_set_duty(ledc_channel[RED].speed_mode, ledc_channel[RED].channel, 0);
00214             ledc_update_duty(ledc_channel[RED].speed_mode, ledc_channel[RED].channel);
00215         }
00216     }
00217 }
00218
00222 static void IRAM_ATTR gpio_isr_handler(void* arg)
00223 {
00224     uint32_t gpio_num = (uint32_t) arg;
00225     xQueueSendFromISR(gpio_evt_queue, &gpio_num, NULL);
00226 }
00227
00232 static void keypadCallback(void* arg)
00233 {
00234     uint32_t adc1_gpio34 = 0;
00235     uint32_t io_num;
00236     uint64_t bufTranslate;
00237     for(;;) {
00238         if(xQueueReceive(gpio_evt_queue, &io_num, portMAX_DELAY)) {
00239             if(keypadLastUpdate != 0 && (keypadLastUpdate + keypadInvalidAfter) > runTime()) {
00240                 //Timeout period reached; reset buffer before input
00241                 memset(keypadInputBuffer, 0, KEYPAD_BUF_SZ);
00242                 keypadInputCnt = 0;
00243             }
00244             // Wait 5ms for input to settle
00245             vTaskDelay(5 / portTICK_PERIOD_MS);
00246             keypadLastUpdate = runTime();
00247             adc1_gpio34 = 0;
00248             for(int i = 0; i < MULTISAMPLING; i++) {
00249                 adc1_gpio34 = adc1_get_raw(ADC1_CHANNEL_6);
00250             }
00251             adc1_gpio34 /= MULTISAMPLING;
00252             for (int i=0; i<12; i++) {
00253                 if(adc1_gpio34 < thresholds[i]) {
00254                     keypadInputBuffer[keypadInputCnt] = keypad[i];
00255                 }
00256             }
00257             if(adc1_gpio34 > thresholds[11]) {
00258                 keypadInputBuffer[keypadInputCnt] = keypad[11];
00259             }
00260             keypadInputCnt++;
00261             // EX: "1#1234*" with "1234" being the pin code would disable the NFC.
00262             if (keypadInputBuffer[keypadInputCnt - 1] == '*' && keypadInputBuffer[1] == '#') {
00263                 //Menu Mode
00264                 //Check Pin
00265                     //Cut out * & keypadInputCnt[0] through '#
00266                     //Double check "-3" and not "-2"
00267                     int numBytes = sizeof(char) * (keypadInputCnt - 3);
00268                     char *pinParse = malloc(numBytes);
00269                     memcpy(pinParse, keypadInputBuffer + 2, numBytes);
00270                     bufTranslate = atoi(pinParse);
00271                     free(pinParse);
00272                 if (bufTranslate == keypadPin && keyPinValid) {
00273                     switch(keypadInputBuffer[0]) {
00274                         case '1':
00275                             //NFC Disable
```

```
00276                           nfcEnable = false;
00277                           break;
00278                       case '2':
00279                           //NFC Enable
00280                           nfcEnable = true;
00281                           break;
00282                       case '3':
00283                           //Unpare All NFC
00284                           unpairNFC();
00285                           break;
00286                       case '4':
00287                           //Pair NFC Tag
00288                           rc522_destroy();            //Remove running callback
00289                           rc522_start(pairNFC_args);  //Attach pairing callback
00290                           //Setup pair timeout. Reuse keypadLastUpdate because...
00291                           keypadLastUpdate = runTime();
00292                           //Wait until pair or NFC timeout
00293                           while (runTime() < (keypadLastUpdate + keypadInvalidAfter) && noPairDect) {
00294                               vTaskDelay(250 / portTICK_PERIOD_MS);
00295                               //Check every 1/4 sec for result
00296                           }
00297                           if (noPairDect) {
00298                               //No tag was detected
00299                               //Blink Red thrice
00300                               for (int i = 0; i < 6; i++) {
00301                                   if ( i % 2) {
00302                                       ledc_set_duty(ledc_channel[RED].speed_mode,
    ledc_channel[RED].channel, 0);
00303                                   } else {
00304                                       ledc_set_duty(ledc_channel[RED].speed_mode,
    ledc_channel[RED].channel, LEDC_TEST_DUTY);
00305                                   }
00306                                   ledc_update_duty(ledc_channel[RED].speed_mode,
    ledc_channel[RED].channel);
00307                                   vTaskDelay(250 / portTICK_PERIOD_MS);
00308                               }
00309                           }
00310                           noPairDect = true;            //Reset for next time.
00311                           rc522_destroy();            //Remove pairing callback
00312                           rc522_start(running_args);  //Attach running callback
00313                           break;
00314                       case '5':
00315                           //Change Pin
00316
00317                           //Clear buffer for new process
00318                           memset(keypadInputBuffer, 0, KEYPAD_BUF_SZ);
00319                           keypadInputCnt = 0;
00320                           //Resume change pin task and suspend this task
00321                           vTaskResume( xPinHandle );
00322                           vTaskSuspend( NULL );
00323                           break;
00324                       default:
00325                           //A proper menu code was not selected.
00326                           //Blink Red thrice
00327                           for (int i = 0; i < 6; i++) {
00328                               if ( i % 2) {
00329                                   ledc_set_duty(ledc_channel[RED].speed_mode, ledc_channel[RED].channel,
    0);
00330                               } else {
00331                                   ledc_set_duty(ledc_channel[RED].speed_mode, ledc_channel[RED].channel,
    LEDC_TEST_DUTY);
00332                               }
00333                               ledc_update_duty(ledc_channel[RED].speed_mode, ledc_channel[RED].channel);
00334                               vTaskDelay(250 / portTICK_PERIOD_MS);
00335                           }
00336                           break;
00337                   }
00338                   //Flash Green
00339                   ledc_set_duty(ledc_channel[GREEN].speed_mode, ledc_channel[GREEN].channel,
    LEDC_TEST_DUTY);
00340                   ledc_update_duty(ledc_channel[GREEN].speed_mode, ledc_channel[GREEN].channel);
00341                   vTaskDelay(250 / portTICK_PERIOD_MS);
00342                   ledc_set_duty(ledc_channel[GREEN].speed_mode, ledc_channel[GREEN].channel, 0);
00343                   ledc_update_duty(ledc_channel[GREEN].speed_mode, ledc_channel[GREEN].channel);
00344               } else {
00345                   //Blink Red twice
00346                   for (int i = 0; i < 4; i++) {
00347                       if ( i % 2) {
00348                           ledc_set_duty(ledc_channel[RED].speed_mode, ledc_channel[RED].channel, 0);
00349                       } else {
00350                           ledc_set_duty(ledc_channel[RED].speed_mode, ledc_channel[RED].channel,
```

```
LEDC_TEST_DUTY);
00351                         }
00352                         ledc_update_duty(ledc_channel[RED].speed_mode, ledc_channel[RED].channel);
00353                         vTaskDelay(125 / portTICK_PERIOD_MS);
00354                     }
00355                 }
00356                 memset(keypadInputBuffer, 0, KEYPAD_BUF_SZ);
00357                 keypadInputCnt = 0;
00358             }
00359             if (keypadInputBuffer[keypadInputCnt - 1] == '*') {
00360                 //Check Pin
00361                 keypadInputBuffer[keypadInputCnt - 1] = 0;
00362                 bufTranslate = atoi(keypadInputBuffer);
00363                 if (bufTranslate == keypadPin && keyPinValid) {
00364                     ledc_set_duty(ledc_channel[GREEN].speed_mode, ledc_channel[GREEN].channel,
LEDC_TEST_DUTY);
00365                     ledc_update_duty(ledc_channel[GREEN].speed_mode, ledc_channel[GREEN].channel);
00366                     toggle_Lock();
00367                     ledc_set_duty(ledc_channel[GREEN].speed_mode, ledc_channel[GREEN].channel, 0);
00368                     ledc_update_duty(ledc_channel[GREEN].speed_mode, ledc_channel[GREEN].channel);
00369                 } else {
00370                     //Blink Red twice
00371                     for (int i = 0; i < 4; i++) {
00372                         if ( i % 2) {
00373                             ledc_set_duty(ledc_channel[RED].speed_mode, ledc_channel[RED].channel, 0);
00374                         } else {
00375                             ledc_set_duty(ledc_channel[RED].speed_mode, ledc_channel[RED].channel,
LEDC_TEST_DUTY);
00376                         }
00377                         ledc_update_duty(ledc_channel[RED].speed_mode, ledc_channel[RED].channel);
00378                         vTaskDelay(125 / portTICK_PERIOD_MS);
00379                     }
00380                 }
00381                 memset(keypadInputBuffer, 0, KEYPAD_BUF_SZ);
00382                 keypadInputCnt = 0;
00383             }
00384             if (keypadInputCnt == KEYPAD_BUF_SZ) {
00385                 //buffer limit reached; wrong code or menu option entered.
00386                 memset(keypadInputBuffer, 0, KEYPAD_BUF_SZ);
00387                 keypadInputCnt = 0;
00388                 //Do error message or something
00389             }
00390         }
00391     }
00392 }
00393
00398 static void keypadPINCallback(void* arg)
00399 {
00400     uint32_t adc1_gpio34 = 0;
00401     uint32_t io_num;
00402     uint64_t bufTranslate;
00403     for(;;) {
00404         if(xQueueReceive(gpio_evt_queue, &io_num, portMAX_DELAY)) {
00405             if(keypadLastUpdate != 0 && (keypadLastUpdate + keypadInvalidAfter) > runTime()) {
00406                 //Timeout period reached; reset buffer before input
00407                 memset(keypadInputBuffer, 0, KEYPAD_BUF_SZ);
00408                 keypadInputCnt = 0;
00409             }
00410             // Wait 5ms for input to settle
00411             vTaskDelay(5 / portTICK_PERIOD_MS);
00412             keypadLastUpdate = runTime();
00413             adc1_gpio34 = 0;
00414             for(int i = 0; i < MULTISAMPLING; i++) {
00415                 adc1_gpio34 = adc1_get_raw(ADC1_CHANNEL_6);
00416             }
00417             adc1_gpio34 /= MULTISAMPLING;
00418             for (int i=0; i<12; i++) {
00419                 if(adc1_gpio34 < thresholds[i]) {
00420                     keypadInputBuffer[keypadInputCnt] = keypad[i];
00421                 }
00422             }
00423             if(adc1_gpio34 > thresholds[11]) {
00424                 keypadInputBuffer[keypadInputCnt] = keypad[11];
00425             }
00426             keypadInputCnt++;
00427             if (keypadInputBuffer[keypadInputCnt - 1] == '*') {
00428                 //Check Pin
00429                 keypadInputBuffer[keypadInputCnt - 1] = 0;
00430                 bufTranslate = atoi(keypadInputBuffer);
00431                 //Update pin in RAM
00432                 keypadPin = bufTranslate;
```

```
00433                 //Update pin in NVS
00434                 writePinNVS(keypadPin);
00435                 //Flash Blue
00436                 ledc_set_duty(ledc_channel[Blue].speed_mode, ledc_channel[Blue].channel, LEDC_TEST_DUTY);
00437                 ledc_update_duty(ledc_channel[Blue].speed_mode, ledc_channel[Blue].channel);
00438                 toggle_Lock();
00439                 ledc_set_duty(ledc_channel[Blue].speed_mode, ledc_channel[Blue].channel, 0);
00440                 ledc_update_duty(ledc_channel[Blue].speed_mode, ledc_channel[Blue].channel);
00441                 //Clear buffer for other process
00442                 memset(keypadInputBuffer, 0, KEYPAD_BUF_SZ);
00443                 keypadInputCnt = 0;
00444                 //Resume other task and suspend self
00445                 vTaskResume( xKeypadHandle );
00446                 vTaskSuspend( NULL );
00447             }
00448             if (keypadInputCnt == KEYPAD_BUF_SZ) {
00449                 //buffer limit reached;
00450                 memset(keypadInputBuffer, 0, KEYPAD_BUF_SZ);
00451                 keypadInputCnt = 0;
00452                 //Resume other task and suspend self
00453                 vTaskResume( xKeypadHandle );
00454                 vTaskSuspend( NULL );
00455             }
00456         }
00457     }
00458 }
00459
00460
00461
00466 void initGPIO() {
00467     //Start ADC setup
00468         gpio_config_t io_conf;
00469
00470         //Keypad Input
00471         io_conf.mode = GPIO_MODE_INPUT;
00472         io_conf.intr_type = GPIO_INTR_NEGEDGE;
00473         io_conf.pin_bit_mask = GPIO_KEYPAD;
00474         io_conf.pull_down_en = 0;
00475         io_conf.pull_up_en = 0;
00476         if(gpio_config(&io_conf) != ESP_OK)
00477             printf("Error setting up GPIO 34 | Keypad Input");
00478
00479         //Start Battery check ADC
00480         io_conf.mode = GPIO_MODE_INPUT;
00481         io_conf.intr_type = GPIO_INTR_NEGEDGE;
00482         io_conf.pin_bit_mask = GPIO_BATV;
00483         io_conf.pull_down_en = 0;
00484         io_conf.pull_up_en = 0;
00485         if(gpio_config(&io_conf) != ESP_OK)
00486             printf("Error setting up GPIO 35 | BatV Input");
00487
00488         adc_power_on();
00489         adc1_config_width(ADC_WIDTH_BIT_12);
00490         adc1_config_channel_atten(ADC1_CHANNEL_6,ADC_ATTEN_DB_11);  //and attetntuation | Channel 6 =
    gpio34 | 11 db = 0 to 3.9v attentuation
00491         adc1_config_channel_atten(ADC1_CHANNEL_7,ADC_ATTEN_DB_11);  //and attetntuation | Channel 7 =
    gpio35 | 11 db = 0 to 3.9v attentuation
00492         esp_adc_cal_characterize(ADC_UNIT_1, ADC_ATTEN_DB_11, ADC_WIDTH_BIT_12, 1101, &characteristics);
00493         gpio_evt_queue = xQueueCreate(10, sizeof(uint32_t));
00494         //Create FreeRTOS task here for keypad events. Maybe move to MAIN later.
00495         xTaskCreate(keypadCallback, "keypadCallback_task", 2048, NULL, 10, &xKeypadHandle);
00496         //Create FreeRTOS task for changing pin. Immediately suspend.
00497         xTaskCreate(keypadPINCallback, "keypadPINCallback_task", 2048, NULL, 10, &xPinHandle);
00498         vTaskSuspend( xPinHandle );
00499         gpio_install_isr_service(ESP_INTR_FLAG_LEVEL);
00500         gpio_isr_handler_add(GPIO_KEYPAD_IO, gpio_isr_handler, (void*) GPIO_KEYPAD_IO);
00501     //End Keypad Setup
00502     //Start Status RGB Setup
00503         ledc_timer_config_t ledc_timer = {
00504             .duty_resolution = LEDC_TIMER_13_BIT,
00505             .freq_hz = 5000,
00506             .speed_mode = LEDC_HS_MODE,
00507             .timer_num = LEDC_HS_TIMER,
00508             .clk_cfg = LEDC_AUTO_CLK,
00509         };
00510         ledc_timer_config(&ledc_timer);
00511         for (int ch = 0; ch < LEDC_TEST_CH_NUM; ch++) {
00512             ledc_channel_config(&ledc_channel[ch]);
00513         }
00514         //Dance and say hello.
00515         for (int i = 0; i < 6; i++) {
```

```
00516             if ( i % 2) {
00517                 ledc_set_duty(ledc_channel[GREEN].speed_mode, ledc_channel[GREEN].channel, 0);
00518             } else {
00519                 ledc_set_duty(ledc_channel[GREEN].speed_mode, ledc_channel[GREEN].channel,
      LEDC_TEST_DUTY);
00520             }
00521             ledc_update_duty(ledc_channel[GREEN].speed_mode, ledc_channel[GREEN].channel);
00522             vTaskDelay(167 / portTICK_PERIOD_MS);
00523         }
00524     //End Status RGB Setup
00525     //Start Motor Control Setup
00526         mcpwm_gpio_init(MCPWM_UNIT_0, MCPWM0A, MTR_PWM0A_OUT);
00527         mcpwm_gpio_init(MCPWM_UNIT_0, MCPWM0B, MTR_PWM0B_OUT);
00528         io_conf.mode = GPIO_MODE_OUTPUT;
00529         io_conf.pin_bit_mask = MTR_Sleep;
00530         io_conf.pull_down_en = true;
00531         io_conf.pull_up_en = false;
00532         if(gpio_config(&io_conf) != ESP_OK)
00533             printf("Error setting up GPIO 33 | Motor Sleep");
00534         gpio_set_level(MTR_SleepC, true);
00535         mcpwm_init(MCPWM_UNIT_0, MCPWM_TIMER_0, &pwm_config);
00536     //End Motor Control Setup
00537 }
00538
00543 void init_NVS() {
00544     esp_err_t err = nvs_flash_init();
00545     //Error handling as recommended by example. May be able to remove later.
00546     if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
00547         ESP_ERROR_CHECK(nvs_flash_erase());
00548         err = nvs_flash_init();
00549     }
00550     ESP_ERROR_CHECK( err );
00551     nvs_handle_t my_handle;
00552     err = nvs_open("storage", NVS_READWRITE, &my_handle);
00553     if (err != ESP_OK) {
00554         //Error condition if NVS cannot be opened. Turn into LED status.
00555         printf("Error (%s) opening NVS handle!\n", esp_err_to_name(err));
00556     } else {
00557         //NVS handle open, withdraw data.
00558         uint8_t tmpStoredKey_QTY = 0;
00559         err = nvs_get_u8(my_handle, "storedKey_QTY", &tmpStoredKey_QTY);
00560         switch (err) {
00561             case ESP_OK:
00562                 storedKey_QTY = tmpStoredKey_QTY;
00563                 printf("Done\n");
00564                 break;
00565             case ESP_ERR_NVS_NOT_FOUND:
00566                 //Must be first time device is being powered up. YAY!!!
00567                 printf("The value is not initialized yet!\n");
00568                 break;
00569             default :
00570                 //Soda machine broke. Run LED alert and reboot.
00571                 printf("Error (%s) reading!\n", esp_err_to_name(err));
00572         }
00573         uint64_t tmpkeypadPin = 0;
00574         err = nvs_get_u64(my_handle, "keypadPin", &tmpkeypadPin);
00575         switch (err) {
00576             case ESP_OK:
00577                 keypadPin = tmpkeypadPin;
00578                 keyPinValid = true;
00579                 printf("Done\n");
00580                 break;
00581             case ESP_ERR_NVS_NOT_FOUND:
00582                 //Must be first time device is being powered up. YAY!!!
00583                 printf("The value is not initialized yet!\n");
00584                 //Default Key Pin
00585                 keypadPin = defaultKEY;
00586                 keyPinValid = true;
00587                 break;
00588             default :
00589                 //Soda machine broke. Run LED alert and reboot.
00590                 printf("Error (%s) reading!\n", esp_err_to_name(err));
00591         }
00592         nvs_close(my_handle);
00593     }
00594 }
00595
00601 void writePinNVS(uint64_t secret) {
00602     esp_err_t err = nvs_flash_init();
00603     if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
00604         ESP_ERROR_CHECK(nvs_flash_erase());
```

```
00605         err = nvs_flash_init();
00606     }
00607     ESP_ERROR_CHECK( err );
00608     nvs_handle_t place_handle;
00609     err = nvs_open("storage", NVS_READWRITE, &place_handle);
00610     if (err != ESP_OK) {
00611         //Error condition if NVS cannot be opened. Turn into LED status.
00612         printf("Error (%s) opening NVS handle!\n", esp_err_to_name(err));
00613     } else {
00614         err = nvs_set_u64(place_handle, "keypadPin", secret);
00615         err = nvs_commit(place_handle);
00616         printf((err != ESP_OK) ? "Failed!\n" : "Done\n");
00617         nvs_close(place_handle);
00618     }
00619 }
00620
00627 void grabKeyList(uint8_t *buf, int count) {
00628     esp_err_t err = nvs_flash_init();
00629     //Error handling as recommended by example. May be able to remove later.
00630     if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
00631         ESP_ERROR_CHECK(nvs_flash_erase());
00632         err = nvs_flash_init();
00633     }
00634     ESP_ERROR_CHECK( err );
00635     nvs_handle_t my_handle;
00636     err = nvs_open("storage", NVS_READWRITE, &my_handle);
00637     if (err != ESP_OK) {
00638         //Error condition if NVS cannot be opened. Turn into LED status.
00639         printf("Error (%s) opening NVS handle!\n", esp_err_to_name(err));
00640     } else {
00641         uint8_t tmp_key[serialLen];
00642         size_t size = sizeof(tmp_key);
00643         char* tmpNvsKey;
00644         tmpNvsKey = "NFCKEY";
00645         strcat(tmpNvsKey, (char*)count);
00646         err = nvs_get_blob(my_handle, tmpNvsKey, tmp_key, &size);
00647         switch (err) {
00648             case ESP_OK:
00649                 *buf = *tmp_key; //double check this line, thx
00650                 printf("Grabbed Key: %u\n",count);
00651                 break;
00652             case ESP_ERR_NVS_NOT_FOUND:
00653                 //Stored keys do not equal qty reported. Bad.
00654                 *buf = 0;
00655                 printf("The key %u is not initialized yet!\n",count);
00656                 break;
00657             default :
00658                 //Soda machine broke. Run LED alert and reboot.
00659                 *buf = 0;
00660                 printf("Error (%s) reading!\n", esp_err_to_name(err));
00661         }
00662         tmpNvsKey[0] = '\0';
00663         if (err != ESP_OK) {
00664             printf("Error (%s) grabbing key!\n", esp_err_to_name(err));
00665         }
00666         nvs_close(my_handle);
00667     }
00668 }
00669
00677 void writeNFCKey(uint8_t *buf, int count) {
00678     esp_err_t err = nvs_flash_init();
00679     if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
00680         ESP_ERROR_CHECK(nvs_flash_erase());
00681         err = nvs_flash_init();
00682     }
00683     ESP_ERROR_CHECK( err );
00684     nvs_handle_t blobHandle;
00685     err = nvs_open("storage", NVS_READWRITE, &blobHandle);
00686     if (err != ESP_OK) {
00687         //Error condition if NVS cannot be opened. Turn into LED status.
00688         printf("Error (%s) opening NVS handle!\n", esp_err_to_name(err));
00689     } else {
00690         uint8_t tmp_key[serialLen];
00691         //Replace line below with known size
00692         size_t size = sizeof(tmp_key);
00693         char* tmpNvsKey;
00694         tmpNvsKey = "NFCKEY";
00695         strcat(tmpNvsKey, (char*)count);
00696         err = nvs_set_blob(blobHandle, tmpNvsKey, buf, size);
00697         err = nvs_commit(blobHandle);
00698         switch (err) {
```

```
00699             case ESP_OK:
00700                 //Successful Write
00701                 printf("Wrote Key: %u\n",count);
00702                 break;
00703             default :
00704                 //Soda machine broke. Run LED alert and reboot.
00705                 printf("Error (%s) Writing!\n", esp_err_to_name(err));
00706         }
00707         nvs_close(blobHandle);
00708     }
00709 }
00710
00714 void unpairNFC() {
00715     //First, free from heap
00716     for (int i=0; i<storedKey_QTY-1; i++) {
00717         free(keyList[i]);
00718     }
00719     esp_err_t err = nvs_flash_init();
00720     if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
00721         ESP_ERROR_CHECK(nvs_flash_erase());
00722         err = nvs_flash_init();
00723     }
00724     ESP_ERROR_CHECK( err );
00725     nvs_handle_t unpairHandle;
00726     err = nvs_open("storage", NVS_READWRITE, &unpairHandle);
00727     if (err != ESP_OK) {
00728         //Error condition if NVS cannot be opened. Turn into LED status.
00729         printf("Error (%s) opening NVS handle!\n", esp_err_to_name(err));
00730     } else {
00731         //Replace lines below with known size
00732         char* tmpNvsKey;
00733         for (int i=0; i<storedKey_QTY-1; i++) {
00734             tmpNvsKey = "NFCKEY";
00735             strcat(tmpNvsKey, (char*)i);
00736             nvs_erase_key(unpairHandle, tmpNvsKey);
00737         }
00738         err = nvs_commit(unpairHandle);
00739         switch (err) {
00740             case ESP_OK:
00741                 //Successful Write
00742                 printf("Successful Clear: \n");
00743                 break;
00744             default :
00745                 //Soda machine broke. Run LED alert and reboot.
00746                 printf("Error (%s) Writing!\n", esp_err_to_name(err));
00747         }
00748         nvs_close(unpairHandle);
00749     }
00750     storedKey_QTY = 0;
00751 }
00752
00758 void writeKeyQtyNVS(uint8_t qty) {
00759     esp_err_t err = nvs_flash_init();
00760     if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
00761         ESP_ERROR_CHECK(nvs_flash_erase());
00762         err = nvs_flash_init();
00763     }
00764     ESP_ERROR_CHECK( err );
00765     nvs_handle_t placeqtyHand;
00766     err = nvs_open("storage", NVS_READWRITE, &placeqtyHand);
00767     if (err != ESP_OK) {
00768         //Error condition if NVS cannot be opened. Turn into LED status.
00769         printf("Error (%s) opening NVS handle!\n", esp_err_to_name(err));
00770     } else {
00771         err = nvs_set_u8(placeqtyHand, "storedKey_QTY", qty);
00772         err = nvs_commit(placeqtyHand);
00773         printf((err != ESP_OK) ? "Failed!\n" : "Done\n");
00774         nvs_close(placeqtyHand);
00775     }
00776 }
00777
00781 void tag_handler(uint8_t* serial_no) {
00782     bool thisIsNotTheKeyYouSeek = true;
00783     //Check every stored NFC key
00784     for(int i = 0; i < storedKey_QTY-1; i++) {
00785         if (serial_no == keyList[i]) {
00786             ledc_set_duty(ledc_channel[GREEN].speed_mode, ledc_channel[GREEN].channel, LEDC_TEST_DUTY);
00787             ledc_update_duty(ledc_channel[GREEN].speed_mode, ledc_channel[GREEN].channel);
00788             toggle_Lock();
00789             ledc_set_duty(ledc_channel[GREEN].speed_mode, ledc_channel[GREEN].channel, 0);
00790             ledc_update_duty(ledc_channel[GREEN].speed_mode, ledc_channel[GREEN].channel);
```

```
00791             thisIsNotTheKeyYouSeek = false;
00792         }
00793     }
00794     if (thisIsNotTheKeyYouSeek) {
00795         //Blink Red thrice
00796         for (int i = 0; i < 6; i++) {
00797             if ( i % 2) {
00798                 ledc_set_duty(ledc_channel[RED].speed_mode, ledc_channel[RED].channel, 0);
00799             } else {
00800                 ledc_set_duty(ledc_channel[RED].speed_mode, ledc_channel[RED].channel, LEDC_TEST_DUTY);
00801             }
00802             ledc_update_duty(ledc_channel[RED].speed_mode, ledc_channel[RED].channel);
00803             vTaskDelay(250 / portTICK_PERIOD_MS);
00804         }
00805     }
00806 }
00807
00811 void pairTagHandler(uint8_t* serial_no) {
00812     //Allocate heap and update
00813     keyList[storedKey_QTY] = (uint8_t *)malloc(serialLen * sizeof(uint8_t));
00814     keyList[storedKey_QTY] = serial_no;
00815     //Update NVS
00816     writeNFCKey(keyList[storedKey_QTY], storedKey_QTY);
00817     storedKey_QTY++;
00818     writeKeyQtyNVS(storedKey_QTY);
00819     noPairDect = false;
00820 }
00821
00825 void toggle_Lock() {
00826     gpio_set_level(MTR_SleepC, false);
00827     vTaskDelay(5 / portTICK_PERIOD_MS);
00828     if (mtrForward) {
00829         mcpwm_set_signal_low(MCPWM_UNIT_0, MCPWM_TIMER_0, MCPWM_OPR_B);
00830         mcpwm_set_duty(MCPWM_UNIT_0, MCPWM_TIMER_0, MCPWM_OPR_A, MTR_duty);
00831         mcpwm_set_duty_type(MCPWM_UNIT_0, MCPWM_TIMER_0, MCPWM_OPR_A, MCPWM_DUTY_MODE_0);
00832         mtrForward = false;
00833     } else {
00834         mcpwm_set_signal_low(MCPWM_UNIT_0, MCPWM_TIMER_0, MCPWM_OPR_A);
00835         mcpwm_set_duty(MCPWM_UNIT_0, MCPWM_TIMER_0, MCPWM_OPR_B, MTR_duty);
00836         mcpwm_set_duty_type(MCPWM_UNIT_0, MCPWM_TIMER_0, MCPWM_OPR_B, MCPWM_DUTY_MODE_0);
00837         mtrForward = true;
00838     }
00839     vTaskDelay(MTR_PERIOD / portTICK_PERIOD_MS);
00840     mcpwm_set_signal_low(MCPWM_UNIT_0, MCPWM_TIMER_0, MCPWM_OPR_A);
00841     mcpwm_set_signal_low(MCPWM_UNIT_0, MCPWM_TIMER_0, MCPWM_OPR_B);
00842     vTaskDelay(5 / portTICK_PERIOD_MS);
00843     gpio_set_level(MTR_SleepC, true);
00844 }
00845
00846 void app_main(void) {
00847     // Runs rc522_init or handles esp32 error.
00848     // rc522_init runs startup, tests, and starts "rc522_task";
00849     // Maybe adjust "rc522_task" to run on FreeRTOS HW interupt and not timer.
00850     init_NVS();
00851     if(storedKey_QTY != 0) {
00852         //Dynamicly cast keylist into heap. Total possible keys: 2^8 = 256
00853         for (int i=0; i<storedKey_QTY-1; i++) {
00854             keyList[i] = (uint8_t *)malloc(serialLen * sizeof(uint8_t));
00855             grabKeyList(keyList[i],i);
00856         }
00857     }
00858     initGPIO();
00859     rc522_start(running_args);
00860     xTaskCreate(batCheck, "Battery_Check_Task", 1024, NULL, 3, NULL);
00861 }
```

Index