Embedded Device Control and Monitoring Firmware with Remote Configuration Support

Generated by Doxygen 1.13.2

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Data Structure Documentation

3.1 ADCSensor Struct Reference

Structure for ADC sensor variables.

`#include <variables.h>`

Data Fields

- Variable base
  Base variable structure.
- char * sensor_type
  Type of the sensor.
- char * pd_sck
  Pin for power-down and serial clock.
- char * dout
  Data output pin.
- double map_low
  Lower mapping range for the sensor value.
- double map_high
  Upper mapping range for the sensor value.
- double gain
  Gain factor for the sensor.
- char * sampling_rate
  Sampling rate for the sensor.
- double value
  Current value of the ADC sensor.

3.1.1 Detailed Description

Structure for ADC sensor variables.

Definition at line 55 of file variables.h.

3.1.2 Field Documentation

3.1.2.1 base

`Variable ADCSensor::base`

Base variable structure.

Definition at line 56 of file variables.h.

3.1.2.2 dout

`char* ADCSensor::dout`

Data output pin.

Definition at line 59 of file variables.h.

3.1.2.3 gain

`double ADCSensor::gain`

Gain factor for the sensor.

Definition at line 62 of file variables.h.

3.1.2.4 map_high

`double ADCSensor::map_high`

Upper mapping range for the sensor value.

Definition at line 61 of file variables.h.

3.1.2.5 map_low

`double ADCSensor::map_low`

Lower mapping range for the sensor value.

Definition at line 60 of file variables.h.

3.1.2.6 pd_sck

`char* ADCSensor::pd_sck`

Pin for power-down and serial clock.

Definition at line 58 of file variables.h.

3.1.2.7 sampling_rate

```
char* ADCSensor::sampling_rate
```

Sampling rate for the sensor.

Definition at line 63 of file variables.h.

3.1.2.8 sensor_type

```
char* ADCSensor::sensor_type
```

Type of the sensor.

Definition at line 57 of file variables.h.

3.1.2.9 value

```
double ADCSensor::value
```

Current value of the ADC sensor.

Definition at line 64 of file variables.h.

The documentation for this struct was generated from the following file:

- main/variables.h

3.2 ADCSensorState Struct Reference

Structure to hold the state of an ADC sensor. This structure stores information about the sensor's name, last read value, whether it has a valid value, and a buffer for recent values.

```
#include <adc_sensor.h>
```

Data Fields

- char * name
- double last_value
- bool has_value
- double value_buffer [VALUE_BUFFER_SIZE]
- int buffer_index
- int buffer_count

3.2.1 Detailed Description

Structure to hold the state of an ADC sensor. This structure stores information about the sensor's name, last read value, whether it has a valid value, and a buffer for recent values.

Definition at line 23 of file adc_sensor.h.

3.2.2 Field Documentation

3.2.2.1 buffer_count

```
int ADCSensorState::buffer_count
```

Definition at line 29 of file adc_sensor.h.

3.2.2.2 buffer_index

```
int ADCSensorState::buffer_index
```

Definition at line 28 of file adc_sensor.h.

3.2.2.3 has_value

```
bool ADCSensorState::has_value
```

Definition at line 26 of file adc_sensor.h.

3.2.2.4 last_value

```
double ADCSensorState::last_value
```

Definition at line 25 of file adc_sensor.h.

3.2.2.5 name

```
char* ADCSensorState::name
```

Definition at line 24 of file adc_sensor.h.

3.2.2.6 value_buffer

```
double ADCSensorState::value_buffer[VALUE_BUFFER_SIZE]
```

Definition at line 27 of file adc_sensor.h.

The documentation for this struct was generated from the following file:

- main/adc_sensor.h

3.3 Boolean Struct Reference

Structure for boolean variables.

```
#include <variables.h>
```

Data Fields

- Variable base

  Base variable structure.
- bool value

  Boolean value.

3.3.1 Detailed Description

Structure for boolean variables.

Definition at line 70 of file variables.h.

3.3.2 Field Documentation

3.3.2.1 base

```
Variable Boolean::base
```

Base variable structure.

Definition at line 71 of file variables.h.

3.3.2.2 value

```
bool Boolean::value
```

Boolean value.

Definition at line 72 of file variables.h.

The documentation for this struct was generated from the following file:

- main/variables.h

3.4 Counter Struct Reference

Structure for counter variables.

```
#include <variables.h>
```

Data Fields

- Variable base
  Base variable structure.
- double pv
  Preset value.
- double cv
  Current value.
- bool cu
  Count up flag.
- bool cd
  Count down flag.
- bool qu
  Output for count up.
- bool qd
  Output for count down.

3.4.1 Detailed Description

Structure for counter variables.

Definition at line 86 of file variables.h.

3.4.2 Field Documentation

3.4.2.1 base

```
Variable Counter::base
```

Base variable structure.

Definition at line 87 of file variables.h.

3.4.2.2 cd

```
bool Counter::cd
```

Count down flag.

Definition at line 91 of file variables.h.

3.4.2.3 cu

```
bool Counter::cu
```

Count up flag.

Definition at line 90 of file variables.h.

3.4.2.4 cv

```
double Counter::cv
```

Current value.

Definition at line 89 of file variables.h.

3.4.2.5 pv

```
double Counter::pv
```

Preset value.

Definition at line 88 of file variables.h.

3.4.2.6 qd

```
bool Counter::qd
```

Output for count down.

Definition at line 93 of file variables.h.

3.4.2.7 qu

```
bool Counter::qu
```

Output for count up.

Definition at line 92 of file variables.h.

The documentation for this struct was generated from the following file:

- main/variables.h

3.5 Device Struct Reference

Structure defining the device configuration.

```
#include <device_config.h>
```

Data Fields

- char * device_name

  Name of the device.
- double logic_voltage

  Logic voltage level of the device.
- int * digital_inputs

  Array of digital input GPIO pins.
- size_t digital_inputs_len

  Length of the digital inputs array.
- char ** digital_inputs_names

  Array of names for digital inputs.
- size_t digital_inputs_names_len

  Length of the digital inputs names array.
- int * digital_outputs

  Array of digital output GPIO pins.
- size_t digital_outputs_len

  Length of the digital outputs array.
- char ** digital_outputs_names

  Array of names for digital outputs.
- size_t digital_outputs_names_len

  Length of the digital outputs names array.
- int * analog_inputs

  Array of analog input GPIO pins.
- size_t analog_inputs_len

  Length of the analog inputs array.
- char ** analog_inputs_names

  Array of names for analog inputs.
- size_t analog_inputs_names_len

  Length of the analog inputs names array.
- int * dac_outputs

  Array of DAC output GPIO pins.
- size_t dac_outputs_len

  Length of the DAC outputs array.
- char ** dac_outputs_names

  Array of names for DAC outputs.
- size_t dac_outputs_names_len

  Length of the DAC outputs names array.
- int * one_wire_inputs

  Array of one-wire input GPIO pins.
- size_t one_wire_inputs_len

  Length of the one-wire inputs array.
- char *** one_wire_inputs_names

  Array of arrays of names for one-wire inputs.
- size_t * one_wire_inputs_names_len

  Array of lengths for one-wire inputs names.
- char *** one_wire_inputs_devices_types

  Array of arrays of device types for one-wire inputs.
- size_t * one_wire_inputs_devices_types_len

  Array of lengths for one-wire device types.
- char *** one_wire_inputs_devices_addresses

Array of arrays of device addresses for one-wire inputs.

- size_t * one_wire_inputs_devices_addresses_len

  Array of lengths for one-wire device addresses.
- int pwm_channels

  Number of PWM channels available.
- int max_hardware_timers

  Maximum number of hardware timers.
- bool has_rtos

  Indicates if the device has an RTOS.
- int * uart

  Array of UART interface pins.
- size_t uart_len

  Length of the UART array.
- int * i2c

  Array of I2C interface pins.
- size_t i2c_len

  Length of the I2C array.
- int * spi

  Array of SPI interface pins.
- size_t spi_len

  Length of the SPI array.
- bool usb

  Indicates if the device has USB support.
- char ** parent_devices

  Array of parent device identifiers.
- size_t parent_devices_len

  Length of the parent devices array.

3.5.1 Detailed Description

Structure defining the device configuration.

Definition at line 11 of file device_config.h.

3.5.2 Field Documentation

3.5.2.1 analog_inputs

```
int* Device::analog_inputs
```

Array of analog input GPIO pins.

Definition at line 26 of file device_config.h.

3.5.2.2 analog_inputs_len

```
size_t Device::analog_inputs_len
```

Length of the analog inputs array.

Definition at line 27 of file device_config.h.

3.5.2.3 analog_inputs_names

```
char** Device::analog_inputs_names
```

Array of names for analog inputs.

Definition at line 28 of file device_config.h.

3.5.2.4 analog_inputs_names_len

```
size_t Device::analog_inputs_names_len
```

Length of the analog inputs names array.

Definition at line 29 of file device_config.h.

3.5.2.5 dac_outputs

```
int* Device::dac_outputs
```

Array of DAC output GPIO pins.

Definition at line 30 of file device_config.h.

3.5.2.6 dac_outputs_len

```
size_t Device::dac_outputs_len
```

Length of the DAC outputs array.

Definition at line 31 of file device_config.h.

3.5.2.7 dac_outputs_names

```
char** Device::dac_outputs_names
```

Array of names for DAC outputs.

Definition at line 32 of file device_config.h.

3.5.2.8 dac_outputs_names_len

```
size_t Device::dac_outputs_names_len
```

Length of the DAC outputs names array.

Definition at line 33 of file device_config.h.

3.5.2.9 device_name

```
char* Device::device_name
```

Name of the device.

Definition at line 12 of file device_config.h.

3.5.2.10 digital_inputs

```
int* Device::digital_inputs
```

Array of digital input GPIO pins.

Definition at line 16 of file device_config.h.

3.5.2.11 digital_inputs_len

```
size_t Device::digital_inputs_len
```

Length of the digital inputs array.

Definition at line 17 of file device_config.h.

3.5.2.12 digital_inputs_names

```
char** Device::digital_inputs_names
```

Array of names for digital inputs.

Definition at line 18 of file device_config.h.

3.5.2.13 digital_inputs_names_len

```
size_t Device::digital_inputs_names_len
```

Length of the digital inputs names array.

Definition at line 19 of file device_config.h.

3.5.2.14 digital_outputs

```
int* Device::digital_outputs
```

Array of digital output GPIO pins.

Definition at line 20 of file device_config.h.

3.5.2.15 digital_outputs_len

```
size_t Device::digital_outputs_len
```

Length of the digital outputs array.

Definition at line 21 of file device_config.h.

3.5.2.16 digital_outputs_names

```
char** Device::digital_outputs_names
```

Array of names for digital outputs.

Definition at line 22 of file device_config.h.

3.5.2.17 digital_outputs_names_len

```
size_t Device::digital_outputs_names_len
```

Length of the digital outputs names array.

Definition at line 23 of file device_config.h.

3.5.2.18 has_rtos

```
bool Device::has_rtos
```

Indicates if the device has an RTOS.

Definition at line 48 of file device_config.h.

3.5.2.19 i2c

```
int* Device::i2c
```

Array of I2C interface pins.

Definition at line 51 of file device_config.h.

3.5.2.20 i2c_len

```
size_t Device::i2c_len
```

Length of the I2C array.

Definition at line 52 of file device_config.h.

3.5.2.21 logic_voltage

```
double Device::logic_voltage
```

Logic voltage level of the device.

Definition at line 13 of file device_config.h.

3.5.2.22 max_hardware_timers

```
int Device::max_hardware_timers
```

Maximum number of hardware timers.

Definition at line 47 of file device_config.h.

3.5.2.23 one_wire_inputs

```
int* Device::one_wire_inputs
```

Array of one-wire input GPIO pins.

Definition at line 36 of file device_config.h.

3.5.2.24 one_wire_inputs_devices_addresses

```
char*** Device::one_wire_inputs_devices_addresses
```

Array of arrays of device addresses for one-wire inputs.

Definition at line 42 of file device_config.h.

3.5.2.25 one_wire_inputs_devices_addresses_len

```
size_t* Device::one_wire_inputs_devices_addresses_len
```

Array of lengths for one-wire device addresses.

Definition at line 43 of file device_config.h.

3.5.2.26 one_wire_inputs_devices_types

```
char*** Device::one_wire_inputs_devices_types
```

Array of arrays of device types for one-wire inputs.

Definition at line 40 of file device_config.h.

3.5.2.27 one_wire_inputs_devices_types_len

```
size_t* Device::one_wire_inputs_devices_types_len
```

Array of lengths for one-wire device types.

Definition at line 41 of file device_config.h.

3.5.2.28 one_wire_inputs_len

```
size_t Device::one_wire_inputs_len
```

Length of the one-wire inputs array.

Definition at line 37 of file device_config.h.

3.5.2.29 one_wire_inputs_names

```
char*** Device::one_wire_inputs_names
```

Array of arrays of names for one-wire inputs.

Definition at line 38 of file device_config.h.

3.5.2.30 one_wire_inputs_names_len

```
size_t* Device::one_wire_inputs_names_len
```

Array of lengths for one-wire inputs names.

Definition at line 39 of file device_config.h.

3.5.2.31 parent_devices

```
char** Device::parent_devices
```

Array of parent device identifiers.

Definition at line 57 of file device_config.h.

3.5.2.32 parent_devices_len

```
size_t Device::parent_devices_len
```

Length of the parent devices array.

Definition at line 58 of file device_config.h.

3.5.2.33 pwm_channels

```
int Device::pwm_channels
```

Number of PWM channels available.

Definition at line 46 of file device_config.h.

3.5.2.34 spi

```
int* Device::spi
```

Array of SPI interface pins.

Definition at line 53 of file device_config.h.

3.5.2.35 spi_len

```
size_t Device::spi_len
```

Length of the SPI array.

Definition at line 54 of file device_config.h.

3.5.2.36 uart

```
int* Device::uart
```

Array of UART interface pins.

Definition at line 49 of file device_config.h.

3.5.2.37 uart_len

```
size_t Device::uart_len
```

Length of the UART array.

Definition at line 50 of file device_config.h.

3.5.2.38 usb

```
bool Device::usb
```

Indicates if the device has USB support.

Definition at line 55 of file device_config.h.

The documentation for this struct was generated from the following file:

- main/device_config.h

3.6 DigitalAnalogInputOutput Struct Reference

Structure for digital/analog input/output variables.

`#include <variables.h>`

Data Fields

- Variable base
  Base variable structure.
- char * pin_number
  Pin number for the I/O.

3.6.1 Detailed Description

Structure for digital/analog input/output variables.

Definition at line 38 of file variables.h.

3.6.2 Field Documentation

3.6.2.1 base

```
Variable DigitalAnalogInputOutput::base
```

Base variable structure.

Definition at line 39 of file variables.h.

3.6.2.2 pin_number

```
char* DigitalAnalogInputOutput::pin_number
```

Pin number for the I/O.

Definition at line 40 of file variables.h.

The documentation for this struct was generated from the following file:

- main/variables.h

3.7 Number Struct Reference

Structure for numeric variables.

`#include <variables.h>`

Data Fields

- Variable base
  Base variable structure.
- double value
  Numeric value.

3.7.1 Detailed Description

Structure for numeric variables.

Definition at line 78 of file variables.h.

3.7.2 Field Documentation

3.7.2.1 base

```
Variable Number::base
```

Base variable structure.

Definition at line 79 of file variables.h.

3.7.2.2 value

```
double Number::value
```

Numeric value.

Definition at line 80 of file variables.h.

The documentation for this struct was generated from the following file:

- main/variables.h

3.8 OneShotState Struct Reference

Structure to track the previous state for one-shot positive coils.

Data Fields

- char var_name [MAX_VAR_NAME_LENGTH]
  Name of the variable.
- bool prev_state
  Previous state of the variable.

3.8.1 Detailed Description

Structure to track the previous state for one-shot positive coils.

Definition at line 21 of file ladder_elements.c.

3.8.2 Field Documentation

3.8.2.1 prev_state

```
bool OneShotState::prev_state
```

Previous state of the variable.

Definition at line 23 of file ladder_elements.c.

3.8.2.2 var_name

```
char OneShotState::var_name[MAX_VAR_NAME_LENGTH]
```

Name of the variable.

Definition at line 22 of file ladder_elements.c.

The documentation for this struct was generated from the following file:

- main/ladder_elements.c

3.9 OneWireInput Struct Reference

Structure for one-wire input variables.

```
#include <variables.h>
```

Data Fields

- Variable base
  Base variable structure.
- char * pin_number
  Pin number for the one-wire device.
- double value
  Current value of the one-wire sensor.

3.9.1 Detailed Description

Structure for one-wire input variables.

Definition at line 46 of file variables.h.

3.9.2 Field Documentation

3.9.2.1 base

```
Variable OneWireInput::base
```

Base variable structure.

Definition at line 47 of file variables.h.

3.9.2.2 pin_number

```
char* OneWireInput::pin_number
```

Pin number for the one-wire device.

Definition at line 48 of file variables.h.

3.9.2.3 value

```
double OneWireInput::value
```

Current value of the one-wire sensor.

Definition at line 49 of file variables.h.

The documentation for this struct was generated from the following file:

- main/variables.h

3.10 SensorState Struct Reference

Data Fields

- int pin
  GPIO pin connected to the one-wire bus.
- char address [17]
  Hex string representation of the sensor address.
- int detection_count
  Positive for detections, negative for misses.

3.10.1 Detailed Description

Definition at line 22 of file one_wire_detect.c.

3.10.2 Field Documentation

3.10.2.1 address

```
char SensorState::address[17]
```

Hex string representation of the sensor address.

Definition at line 24 of file one_wire_detect.c.

3.10.2.2 detection_count

```
int SensorState::detection_count
```

Positive for detections, negative for misses.

Definition at line 25 of file one_wire_detect.c.

3.10.2.3 pin

```
int SensorState::pin
```

GPIO pin connected to the one-wire bus.

Definition at line 23 of file one_wire_detect.c.

The documentation for this struct was generated from the following file:

- main/one_wire_detect.c

3.11 TaskInfo Struct Reference

Structure to store task information.

Data Fields

- TaskHandle_t handle
  Handle of the FreeRTOS task.
- cJSON * wire_copy
  Copy of the JSON wire configuration.

3.11.1 Detailed Description

Structure to store task information.

Definition at line 42 of file conf_task_manager.c.

3.11.2 Field Documentation

3.11.2.1 handle

```
TaskHandle_t TaskInfo::handle
```

Handle of the FreeRTOS task.

Definition at line 43 of file conf_task_manager.c.

3.11.2.2 wire_copy

```
cJSON* TaskInfo::wire_copy
```

Copy of the JSON wire configuration.

Definition at line 44 of file conf_task_manager.c.

The documentation for this struct was generated from the following file:

- main/conf_task_manager.c

3.12 Time Struct Reference

Structure for time variables.

```
#include <variables.h>
```

Data Fields

- Variable base
  Base variable structure.
- double value
  Time value.

3.12.1 Detailed Description

Structure for time variables.

Definition at line 110 of file variables.h.

3.12.2 Field Documentation

3.12.2.1 base

```
Variable Time::base
```

Base variable structure.

Definition at line 111 of file variables.h.

3.12.2.2 value

```
double Time::value
```

Time value.

Definition at line 112 of file variables.h.

The documentation for this struct was generated from the following file:

- main/variables.h

3.13 Timer Struct Reference

Structure for timer variables.

```
#include <variables.h>
```

Data Fields

- Variable base
  Base variable structure.
- double pt
  Preset time.
- double et
  Elapsed time.
- bool in
  Input state.
- bool q
  Output state.

3.13.1 Detailed Description

Structure for timer variables.

Definition at line 99 of file variables.h.

3.13.2 Field Documentation

3.13.2.1 base

```
Variable Timer::base
```

Base variable structure.

Definition at line 100 of file variables.h.

3.13.2.2 et

```
double Timer::et
```

Elapsed time.

Definition at line 102 of file variables.h.

3.13.2.3 in

```
bool Timer::in
```

Input state.

Definition at line 103 of file variables.h.

3.13.2.4 pt

```
double Timer::pt
```

Preset time.

Definition at line 101 of file variables.h.

3.13.2.5 q

```
bool Timer::q
```

Output state.

Definition at line 104 of file variables.h.

The documentation for this struct was generated from the following file:

- main/variables.h

3.14 TimerState Struct Reference

Structure to track the state of timers.

Data Fields

- char var_name [MAX_VAR_NAME_LENGTH]
  Name of the timer variable.
- int64_t start_time
  Start time in microseconds.
- bool running
  Indicates if the timer is active.

3.14.1 Detailed Description

Structure to track the state of timers.

Definition at line 29 of file ladder_elements.c.

3.14.2 Field Documentation

3.14.2.1 running

```
bool TimerState::running
```

Indicates if the timer is active.

Definition at line 32 of file ladder_elements.c.

3.14.2.2 start_time

```
int64_t TimerState::start_time
```

Start time in microseconds.

Definition at line 31 of file ladder_elements.c.

3.14.2.3 var_name

```
char TimerState::var_name[MAX_VAR_NAME_LENGTH]
```

Name of the timer variable.

Definition at line 30 of file ladder_elements.c.

The documentation for this struct was generated from the following file:

- main/ladder_elements.c

3.15 Variable Struct Reference

Base structure for a variable.

```
#include <variables.h>
```

Data Fields

- char * name
  Variable name.
- char * type
  Variable type as a string.

3.15.1 Detailed Description

Base structure for a variable.

Definition at line 30 of file variables.h.

3.15.2 Field Documentation

3.15.2.1 name

```
char* Variable::name
```

Variable name.

Definition at line 31 of file variables.h.

3.15.2.2 type

```
char* Variable::type
```

Variable type as a string.

Definition at line 32 of file variables.h.

The documentation for this struct was generated from the following file:

- main/variables.h

3.16 VariableNode Struct Reference

Structure for a variable node in the variables list.

```
#include <variables.h>
```

Data Fields

- VariableType type
  Type of the variable.
- void * data
  Pointer to the variable data.

3.16.1 Detailed Description

Structure for a variable node in the variables list.

Definition at line 118 of file variables.h.

3.16.2 Field Documentation

3.16.2.1 data

```
void* VariableNode::data
```

Pointer to the variable data.

Definition at line 120 of file variables.h.

3.16.2.2 type

```
VariableType VariableNode::type
```

Type of the variable.

Definition at line 119 of file variables.h.

The documentation for this struct was generated from the following file:

- main/variables.h

3.17 VariablesList Struct Reference

Structure for managing a list of variables.

```
#include <variables.h>
```

Data Fields

- VariableNode * nodes
  Array of variable nodes.
- size_t count
  Current number of variables.
- size_t capacity
  Capacity of the array.

3.17.1 Detailed Description

Structure for managing a list of variables.

Definition at line 126 of file variables.h.

3.17.2 Field Documentation

3.17.2.1 capacity

```
size_t VariablesList::capacity
```

Capacity of the array.

Definition at line 129 of file variables.h.

3.17.2.2 count

```
size_t VariablesList::count
```

Current number of variables.

Definition at line 128 of file variables.h.

3.17.2.3 nodes

```
VariableNode* VariablesList::nodes
```

Array of variable nodes.

Definition at line 127 of file variables.h.

The documentation for this struct was generated from the following file:

- main/variables.h

Chapter 4

File Documentation

4.1 config/config.h File Reference

Configuration header for Wi-Fi and MQTT settings.

Macros

- #define WIFI_SSID "wifi_ssid"

  Wi-Fi SSID (network name) used for connecting the device to a wireless network.
- #define WIFI_PASS "wifi_pass"

  Wi-Fi password associated with the SSID.
- #define MQTT_BROKER_URI "mqtt://user:pass@broker"

  URI of the MQTT broker, including optional credentials.

4.1.1 Detailed Description

Configuration header for Wi-Fi and MQTT settings.

This file defines preprocessor macros for Wi-Fi credentials and the MQTT broker URI. These values are used throughout the application to establish network and broker connections.

Definition in file config.h.

4.1.2 Macro Definition Documentation

4.1.2.1 MQTT_BROKER_URI

```
#define MQTT_BROKER_URI "mqtt://user:pass@broker"
```

URI of the MQTT broker, including optional credentials.

Format: mqtt://username:password@broker_address

Definition at line 28 of file config.h.

4.1.2.2 WIFI_PASS

```
#define WIFI_PASS "wifi_pass"
```

Wi-Fi password associated with the SSID.

Definition at line 21 of file config.h.

4.1.2.3 WIFI_SSID

```
#define WIFI_SSID "wifi_ssid"
```

Wi-Fi SSID (network name) used for connecting the device to a wireless network.

Definition at line 15 of file config.h.

4.2 config.h

Go to the documentation of this file.

```
00001
00008
00009 #ifndef CONFIG_H
00010 #define CONFIG_H
00011
00015 #define WIFI_SSID "wifi_ssid"
00016
00017
00021 #define WIFI_PASS "wifi_pass"
00022
00028 #define MQTT_BROKER_URI "mqtt://user:pass@broker"
00029
00030 #endif // CONFIG_H
```

4.3 main/adc_sensor.c File Reference

```
#include "adc_sensor.h"
#include <string.h>
#include "device_config.h"
#include "TM7711.h"
```

Functions

- double map_value (double value, double fromLow, double fromHigh, double toLow, double toHigh)

  Maps a value from one range to another.
- static ADCSensorState * find_or_add_sensor_state (const char *sensor_name)

  Finds or adds a sensor state based on the sensor name.
- esp_err_t adc_sensor_init (char *sensor_type, char *pd_sck, char *dout)

  Initializes an ADC sensor with the specified configuration.
- double adc_sensor_read (char *sensor_type, char *pd_sck, char *dout, double map_low, double map_high, double gain, char *sampling_rate, const char *sensor_name)

  Reads the value from an ADC sensor and maps it to a specified range.

Variables

- static const char * TAG = "ADC_SENSOR"

  Tag for logging messages from the ADC sensor module.
- ADCSensorState sensor_states [MAX_ADC_SENSORS]

  Array to store the state of all ADC sensors. This array holds the state for up to MAX_ADC_SENSORS sensors.
- int sensor_state_count = 0

  Counter for the number of configured ADC sensors. Tracks the total number of sensors currently in use.

4.3.1 Function Documentation

4.3.1.1 adc_sensor_init()

```
esp_err_t adc_sensor_init (
            char * sensor_type,
            char * pd_sck,
            char * dout)
```

Initializes an ADC sensor with the specified configuration.

Parameters

sensor_type	Type of the sensor (e.g., model or identifier).
pd_sck	Pin used for the clock signal.
dout	Pin used for data output.

Returns

esp_err_t Error code indicating success or failure of initialization.

Definition at line 67 of file adc_sensor.c.

4.3.1.2 adc_sensor_read()

```
double adc_sensor_read (
            char * sensor_type,
            char * pd_sck,
            char * dout,
            double map_low,
            double map_high,
            double gain,
            char * sampling_rate,
            const char * sensor_name)
```

Reads the value from an ADC sensor and maps it to a specified range.

Parameters

sensor_type	Type of the sensor (e.g., model or identifier).
pd_sck	Pin used for the clock signal.
dout	Pin used for data output.

map_low	Lower bound of the mapped output range.
map_high	Upper bound of the mapped output range.
gain	Gain factor to apply to the sensor reading.
sampling_rate	Sampling rate for reading the sensor.
sensor_name	Name of the sensor for identification.

Returns

double The mapped sensor value.

Definition at line 96 of file adc_sensor.c.

4.3.1.3 find_or_add_sensor_state()

```
static ADCSensorState * find_or_add_sensor_state (
            const char * sensor_name)  [static]
```

Finds or adds a sensor state based on the sensor name.

Parameters

sensor_name	The name of the sensor to find or add.

Returns

ADCSensorState* Pointer to the sensor state, or NULL if capacity is exceeded.

Definition at line 44 of file adc_sensor.c.

4.3.1.4 map_value()

```
double map_value (
            double value,
            double fromLow,
            double fromHigh,
            double toLow,
            double toHigh)
```

Maps a value from one range to another.

Parameters

value	The input value to map.
fromLow	The lower bound of the input range.
fromHigh	The upper bound of the input range.
toLow	The lower bound of the output range.
toHigh	The upper bound of the output range.

Returns

double The mapped value in the target range.

Definition at line 32 of file adc_sensor.c.

4.3.2 Variable Documentation

4.3.2.1 sensor_state_count

```
int sensor_state_count = 0
```

Counter for the number of configured ADC sensors. Tracks the total number of sensors currently in use.

Counter for the number of configured ADC sensors. This external variable tracks the total number of sensors in use.

Definition at line 21 of file adc_sensor.c.

4.3.2.2 sensor_states

```
ADCSensorState sensor_states[MAX_ADC_SENSORS]
```

Array to store the state of all ADC sensors. This array holds the state for up to MAX_ADC_SENSORS sensors.

Array to store the state of all ADC sensors. This external array holds the state for each configured ADC sensor.

Definition at line 15 of file adc_sensor.c.

4.3.2.3 TAG

```
const char* TAG = "ADC_SENSOR"  [static]
```

Tag for logging messages from the ADC sensor module.

Definition at line 9 of file adc_sensor.c.

4.4 adc_sensor.c

Go to the documentation of this file.

```
00001 #include "adc_sensor.h"
00002 #include <string.h>
00003 #include "device_config.h"
00004 #include "TM7711.h"
00005
00009 static const char *TAG = "ADC_SENSOR";
00010
00015 ADCSensorState sensor_states[MAX_ADC_SENSORS];
00016
00021 int sensor_state_count = 0;
00022
00032 double map_value(double value, double fromLow, double fromHigh, double toLow, double toHigh) {
00033     if (fromHigh == fromLow) {
00034         return toLow; // Prevents division by zero
00035     }
00036     return (value - fromLow) * (toHigh - toLow) / (fromHigh - fromLow) + toLow;
00037 }
00038
00044 static ADCSensorState *find_or_add_sensor_state(const char *sensor_name) {
00045     // Check if sensor already exists
00046     for (int i = 0; i < sensor_state_count; i++) {
00047         if (sensor_states[i].name && strcmp(sensor_states[i].name, sensor_name) == 0) {
00048             return &sensor_states[i];
00049         }
00050     }
00051     // Add new sensor if capacity allows
00052     if (sensor_state_count < MAX_ADC_SENSORS) {
```

```
00053         sensor_states[sensor_state_count].name = strdup(sensor_name); // Allocate memory for sensor
      name
00054         sensor_states[sensor_state_count].last_value = 0.0;           // Initialize last value
00055         sensor_states[sensor_state_count].has_value = false;          // Initialize value flag
00056         sensor_states[sensor_state_count].buffer_index = 0;           // Initialize buffer index
00057         sensor_states[sensor_state_count].buffer_count = 0;           // Initialize buffer count
00058         for (int j = 0; j < VALUE_BUFFER_SIZE; j++) {
00059             sensor_states[sensor_state_count].value_buffer[j] = 0.0;  // Clear value buffer
00060         }
00061         return &sensor_states[sensor_state_count++];                 // Return new sensor state and
      increment count
00062     }
00063     ESP_LOGE(TAG, "Sensor capacity exceeded");
00064     return NULL;
00065 }
00066
00067 esp_err_t adc_sensor_init(char *sensor_type, char *pd_sck, char *dout){
00068     gpio_num_t dout_pin, pd_sck_pin;
00069     esp_err_t ret;
00070
00071     // Find GPIO pins by name
00072     if (!find_pin_by_name(pd_sck, &pd_sck_pin)) {
00073         ESP_LOGE(TAG, "PD_SCK pin %s not found", pd_sck);
00074         return ESP_ERR_INVALID_ARG;
00075     }
00076     if (!find_pin_by_name(dout, &dout_pin)) {
00077         ESP_LOGE(TAG, "DOUT pin %s not found", dout);
00078         return ESP_ERR_INVALID_ARG;
00079     }
00080
00081     // Initialize TM7711 sensor if specified
00082     if (strcmp(sensor_type, "TM7711") == 0) {
00083         // Initialize TM7711 with specified pins
00084         ret = tm7711_init(dout_pin, pd_sck_pin);
00085         if (ret != ESP_OK) {
00086             ESP_LOGE(TAG, "TM7711 initialization failed: %d", ret);
00087             return ret;
00088         }
00089         return ESP_OK;
00090     }
00091
00092     // Return error for unsupported sensor types
00093     return ESP_ERR_NOT_SUPPORTED;
00094 }
00095
00096 double adc_sensor_read(char *sensor_type, char *pd_sck, char *dout, double map_low, double map_high,
      double gain, char *sampling_rate, const char *sensor_name) {
00097     gpio_num_t dout_pin, pd_sck_pin;
00098     unsigned long data = 0;
00099     esp_err_t ret;
00100
00101     // Find GPIO pins by name
00102     if (!find_pin_by_name(pd_sck, &pd_sck_pin)) {
00103         ESP_LOGE(TAG, "PD_SCK pin %s not found", pd_sck);
00104         return 0.0;
00105     }
00106     if (!find_pin_by_name(dout, &dout_pin)) {
00107         ESP_LOGE(TAG, "DOUT pin %s not found", dout);
00108         return 0.0;
00109     }
00110
00111     // Validate mapping parameters and gain
00112     if (map_low == map_high || gain < 0) {
00113         ESP_LOGE(TAG, "Invalid mapping parameters or gain");
00114         return 0.0;
00115     }
00116
00117     // Handle TM7711 sensor reading
00118     if (strcmp(sensor_type, "TM7711") == 0) {
00119         unsigned char next_select;
00120
00121         // Map sampling rate string to corresponding constant
00122         if (strcmp(sampling_rate, "10Hz") == 0) {
00123             next_select = CH1_10HZ;
00124         } else if (strcmp(sampling_rate, "40Hz") == 0) {
00125             next_select = CH1_40HZ;
00126         } else if (strcmp(sampling_rate, "Temperature") == 0) {
00127             next_select = CH2_TEMP;
00128         } else {
00129             ESP_LOGE(TAG, "Unsupported sampling_rate value: %s", sampling_rate);
00130             return 0.0;
00131         }
00132
00133         // Read data from TM7711 sensor
00134         ret = tm7711_read(next_select, dout_pin, pd_sck_pin, &data);
00135         if (ret != ESP_OK) {
00136             ESP_LOGE(TAG, "TM7711 read failed: %d", ret);
```

```
00137             return 0.0;
00138         }
00139
00140         // Find or add sensor state
00141         ADCSensorState *state = find_or_add_sensor_state(sensor_name);
00142         if (!state) {
00143             return 0.0;
00144         }
00145
00146         // Check for extreme values (min=0, max=16777215 for 24-bit ADC)
00147         if (data == 0 || data == 16777215) {
00148             ESP_LOGW(TAG, "Extreme value detected for %s: %lu, returning last value", sensor_name,
      data);
00149             return state->has_value ? state->last_value : 0.0;
00150         }
00151
00152         // Map the raw data to the specified range
00153         double mapped_value = map_value((double)data, 0, 16777215, map_low, map_high);
00154
00155         // Update the sensor value buffer
00156         state->value_buffer[state->buffer_index] = mapped_value;
00157         state->buffer_index = (state->buffer_index + 1) % VALUE_BUFFER_SIZE;
00158         if (state->buffer_count < VALUE_BUFFER_SIZE) {
00159             state->buffer_count++;
00160         }
00161
00162         // Calculate the average of buffered values
00163         double sum = 0.0;
00164         for (int i = 0; i < state->buffer_count; i++) {
00165             sum += state->value_buffer[i];
00166         }
00167         double avg_value = sum / state->buffer_count;
00168
00169         // Update the last value and validity flag
00170         state->last_value = avg_value;
00171         state->has_value = true;
00172
00173         return avg_value;
00174     }
00175
00176     // Log error for unsupported sensor types
00177     ESP_LOGE(TAG, "Unsupported sensor type: %s", sensor_type);
00178     return 0.0;
00179 }
```

4.5 main/adc_sensor.h File Reference

```
#include "driver/gpio.h"
```

Data Structures

- struct ADCSensorState

  Structure to hold the state of an ADC sensor. This structure stores information about the sensor's name, last read value, whether it has a valid value, and a buffer for recent values.

Macros

- #define MAX_ADC_SENSORS 10

  Maximum number of ADC sensors supported. This macro defines the maximum number of ADC sensors that can be managed.
- #define VALUE_BUFFER_SIZE 3

  Size of the buffer to store the last sensor values. This macro specifies that the last 3 values are stored for each sensor.

Functions

- esp_err_t adc_sensor_init (char ∗sensor_type, char ∗pd_sck, char ∗dout)

  Initializes an ADC sensor with the specified configuration.
- double adc_sensor_read (char ∗sensor_type, char ∗pd_sck, char ∗dout, double map_low, double map_high, double gain, char ∗sampling_rate, const char ∗sensor_name)

  Reads the value from an ADC sensor and maps it to a specified range.

Variables

- ADCSensorState sensor_states [ ]

  Array to store the state of all ADC sensors. This external array holds the state for each configured ADC sensor.
- int sensor_state_count

  Counter for the number of configured ADC sensors. This external variable tracks the total number of sensors in use.

4.5.1 Macro Definition Documentation

4.5.1.1 MAX_ADC_SENSORS

```
#define MAX_ADC_SENSORS 10
```

Maximum number of ADC sensors supported. This macro defines the maximum number of ADC sensors that can be managed.

Definition at line 10 of file adc_sensor.h.

4.5.1.2 VALUE_BUFFER_SIZE

```
#define VALUE_BUFFER_SIZE 3
```

Size of the buffer to store the last sensor values. This macro specifies that the last 3 values are stored for each sensor.

Definition at line 16 of file adc_sensor.h.

4.5.2 Function Documentation

4.5.2.1 adc_sensor_init()

```
esp_err_t adc_sensor_init (
            char * sensor_type,
            char * pd_sck,
            char * dout)
```

Initializes an ADC sensor with the specified configuration.

Parameters

sensor_type	Type of the sensor (e.g., model or identifier).
pd_sck	Pin used for the clock signal.
dout	Pin used for data output.

Returns

esp_err_t Error code indicating success or failure of initialization.

Definition at line 67 of file adc_sensor.c.

4.5.2.2 adc_sensor_read()

```
double adc_sensor_read (
            char * sensor_type,
            char * pd_sck,
            char * dout,
            double map_low,
            double map_high,
            double gain,
            char * sampling_rate,
            const char * sensor_name)
```

Reads the value from an ADC sensor and maps it to a specified range.

Parameters

sensor_type	Type of the sensor (e.g., model or identifier).
pd_sck	Pin used for the clock signal.
dout	Pin used for data output.
map_low	Lower bound of the mapped output range.
map_high	Upper bound of the mapped output range.
gain	Gain factor to apply to the sensor reading.
sampling_rate	Sampling rate for reading the sensor.
sensor_name	Name of the sensor for identification.

Returns

double The mapped sensor value.

Definition at line 96 of file adc_sensor.c.

4.5.3 Variable Documentation

4.5.3.1 sensor_state_count

```
int sensor_state_count  [extern]
```

Counter for the number of configured ADC sensors. This external variable tracks the total number of sensors in use.

Counter for the number of configured ADC sensors. This external variable tracks the total number of sensors in use.

Definition at line 21 of file adc_sensor.c.

4.5.3.2 sensor_states

```
ADCSensorState sensor_states[]  [extern]
```

Array to store the state of all ADC sensors. This external array holds the state for each configured ADC sensor.

Array to store the state of all ADC sensors. This external array holds the state for each configured ADC sensor.

Definition at line 15 of file adc_sensor.c.

4.6 adc_sensor.h

Go to the documentation of this file.

```
00001 #ifndef _ADC_SENSOR_H_
00002 #define _ADC_SENSOR_H_
00003
00004 #include "driver/gpio.h"
00005
00010 #define MAX_ADC_SENSORS 10 // Adjust based on the number of sensors
00011
00016 #define VALUE_BUFFER_SIZE 3 // Store the last 3 values
00017
00023 typedef struct {
00024     char *name;                    // Name of the sensor
00025     double last_value;             // Last recorded sensor value
00026     bool has_value;                // Flag indicating if the sensor has a valid value
00027     double value_buffer[VALUE_BUFFER_SIZE]; // Buffer to store the last few sensor values
00028     int buffer_index;              // Current index in the value buffer
00029     int buffer_count;              // Number of values currently stored in the buffer
00030 } ADCSensorState;
00031
00036 extern ADCSensorState sensor_states[];
00037
00042 extern int sensor_state_count;
00043
00051 esp_err_t adc_sensor_init(char *sensor_type, char *pd_sck, char *dout);
00052
00065 double adc_sensor_read(char *sensor_type, char *pd_sck, char *dout, double map_low, double map_high,
      double gain, char *sampling_rate, const char *sensor_name);
00066
00067 #endif
```

4.7 main/ble.c File Reference

```
#include "ble.h"
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "nvs_flash.h"
#include "esp_log.h"
#include "nimble/nimble_port.h"
#include "nimble/nimble_port_freertos.h"
#include "host/ble_hs.h"
#include "services/gap/ble_svc_gap.h"
#include "services/gatt/ble_svc_gatt.h"
#include "driver/gpio.h"
#include "esp_mac.h"
#include "conf_task_manager.h"
#include "nvs_utils.h"
#include "variables.h"
#include "one_wire_detect.h"
```

Functions

- void ble_app_advertise (void)

  Function prototype for starting BLE advertising.
- static int configuration_read (uint16_t conn_handle, uint16_t attr_handle, struct ble_gatt_access_ctxt *ctxt, void *arg)

  Handles read requests for the configuration characteristic. Loads configuration from NVS and sends it in chunks based on the MTU size.
- static int configuration_write (uint16_t conn_handle, uint16_t attr_handle, struct ble_gatt_access_ctxt *ctxt, void *arg)

  Handles write requests for the configuration characteristic. Applies the received configuration data.
- static int monitor_read (uint16_t conn_handle, uint16_t attr_handle, struct ble_gatt_access_ctxt *ctxt, void *arg)

  Handles read requests for the monitor characteristic. Reads variable data as JSON and sends it in chunks based on the MTU size.
- static int one_wire_read (uint16_t conn_handle, uint16_t attr_handle, struct ble_gatt_access_ctxt *ctxt, void *arg)

  Handles read requests for the one-wire sensor characteristic. Reads one-wire sensor data and sends it in chunks based on the MTU size.
- static int ble_gap_event (struct ble_gap_event *event, void *arg)

  Handles BLE GAP events such as connection, disconnection, and MTU updates.
- void ble_app_on_sync (void)

  Callback function called when the BLE stack is synchronized. Initiates advertising after setting the BLE address type.
- void host_task (void *param)

  Task function for running the NimBLE host. Runs indefinitely until nimble_port_stop() is called.
- static void set_ble_name_from_mac ()

  Sets the BLE device name based on the Bluetooth MAC address. Creates a device name in the format "ESP_↩ XXYYZZ" using the first three bytes of the MAC address.
- void ble_init (void)

  Initializes the BLE stack and services. Sets up the NimBLE host, GAP, GATT, and custom services, and starts the host task.

Variables

- char * TAG = "BLE-Server"

  Tag for logging messages from the BLE server module.
- static uint16_t ble_mtu = 23

  Default MTU size for BLE communication. Initially set to the minimum value of 23 bytes.
- uint8_t ble_addr_type

  BLE address type (public or random).
- uint16_t conn_handle = BLE_HS_CONN_HANDLE_NONE

  Connection handle for the active BLE connection. Initialized to BLE_HS_CONN_HANDLE_NONE when no connection exists.
- bool app_connected_ble = false

  Flag indicating whether the application is connected via BLE.
- static const struct ble_gatt_svc_def gatt_svcs [ ]

  GATT service definitions for the BLE server. Defines a primary service with characteristics for configuration read/write, monitor data read, and one-wire sensor data read.

4.7.1 Function Documentation

4.7.1.1 ble_app_advertise()

```
void ble_app_advertise (
            void )
```

Function prototype for starting BLE advertising.

Starts BLE advertising with the device name and service UUID.

Definition at line 314 of file ble.c.

4.7.1.2 ble_app_on_sync()

```
void ble_app_on_sync (
            void )
```

Callback function called when the BLE stack is synchronized. Initiates advertising after setting the BLE address type.

Definition at line 359 of file ble.c.

4.7.1.3 ble_gap_event()

```
static int ble_gap_event (
            struct ble_gap_event * event,
            void * arg)  [static]
```

Handles BLE GAP events such as connection, disconnection, and MTU updates.

Parameters

event	The GAP event structure.
arg	Unused argument.

Returns

int 0 on success.

Definition at line 269 of file ble.c.

4.7.1.4 ble_init()

```
void ble_init (
            void )
```

Initializes the BLE stack and services. Sets up the NimBLE host, GAP, GATT, and custom services, and starts the host task.

Initializes the BLE subsystem.

Definition at line 397 of file ble.c.

4.7.1.5 configuration_read()

```
static int configuration_read (
            uint16_t conn_handle,
            uint16_t attr_handle,
            struct ble_gatt_access_ctxt * ctxt,
            void * arg)  [static]
```

Handles read requests for the configuration characteristic. Loads configuration from NVS and sends it in chunks based on the MTU size.

Parameters

conn_handle	Connection handle for the BLE connection.
attr_handle	Attribute handle of the characteristic.
ctxt	Context for the GATT access operation.
arg	Unused argument.

Returns

int 0 on success, or a BLE_ATT error code on failure.

Definition at line 63 of file ble.c.

4.7.1.6 configuration_write()

```
static int configuration_write (
            uint16_t conn_handle,
            uint16_t attr_handle,
            struct ble_gatt_access_ctxt * ctxt,
            void * arg)  [static]
```

Handles write requests for the configuration characteristic. Applies the received configuration data.

Parameters

conn_handle	Connection handle for the BLE connection.
attr_handle	Attribute handle of the characteristic.
ctxt	Context for the GATT access operation.
arg	Unused argument.

Returns

int 0 on success.

Definition at line 115 of file ble.c.

4.7.1.7 host_task()

```
void host_task (
            void * param)
```

Task function for running the NimBLE host. Runs indefinitely until nimble_port_stop() is called.

Parameters

param	Unused parameter.

Definition at line 375 of file ble.c.

4.7.1.8 monitor_read()

```
static int monitor_read (
            uint16_t conn_handle,
            uint16_t attr_handle,
            struct ble_gatt_access_ctxt * ctxt,
            void * arg)  [static]
```

Handles read requests for the monitor characteristic. Reads variable data as JSON and sends it in chunks based on the MTU size.

Parameters

conn_handle	Connection handle for the BLE connection.
attr_handle	Attribute handle of the characteristic.
ctxt	Context for the GATT access operation.
arg	Unused argument.

Returns

int 0 on success, or a BLE_ATT error code on failure.

Definition at line 130 of file ble.c.

4.7.1.9 one_wire_read()

```
static int one_wire_read (
            uint16_t conn_handle,
            uint16_t attr_handle,
            struct ble_gatt_access_ctxt * ctxt,
            void * arg)  [static]
```

Handles read requests for the one-wire sensor characteristic. Reads one-wire sensor data and sends it in chunks based on the MTU size.

Parameters

conn_handle	Connection handle for the BLE connection.
attr_handle	Attribute handle of the characteristic.
ctxt	Context for the GATT access operation.
arg	Unused argument.

Returns

int 0 on success, or a BLE_ATT error code on failure.

Definition at line 183 of file ble.c.

4.7.1.10 set_ble_name_from_mac()

```
static void set_ble_name_from_mac ()  [static]
```

Sets the BLE device name based on the Bluetooth MAC address. Creates a device name in the format "ESP_↩XXYYZZ" using the first three bytes of the MAC address.

Definition at line 384 of file ble.c.

4.7.2 Variable Documentation

4.7.2.1 app_connected_ble

```
bool app_connected_ble = false
```

Flag indicating whether the application is connected via BLE.

Flag indicating if the BLE application is connected.

Definition at line 52 of file ble.c.

4.7.2.2 ble_addr_type

```
uint8_t ble_addr_type
```

BLE address type (public or random).

Definition at line 36 of file ble.c.

4.7.2.3 ble_mtu

```
uint16_t ble_mtu = 23  [static]
```

Default MTU size for BLE communication. Initially set to the minimum value of 23 bytes.

Definition at line 31 of file ble.c.

4.7.2.4 conn_handle

```
uint16_t conn_handle = BLE_HS_CONN_HANDLE_NONE
```

Connection handle for the active BLE connection. Initialized to BLE_HS_CONN_HANDLE_NONE when no connection exists.

Definition at line 47 of file ble.c.

4.7.2.5 gatt_svcs

```
const struct ble_gatt_svc_def gatt_svcs[]  [static]
```

Initial value:

```
= {
    {
        .type = BLE_GATT_SVC_TYPE_PRIMARY,
        .uuid = BLE_UUID16_DECLARE(SERVICE_UUID),
        .characteristics = (struct ble_gatt_chr_def[]){
            {
                .uuid = BLE_UUID16_DECLARE(READ_CONFIGURATION_CHAR_UUID),
                .flags = BLE_GATT_CHR_F_READ,
                .access_cb = configuration_read
            },
            {
                .uuid = BLE_UUID16_DECLARE(WRITE_CONFIGURATION_CHAR_UUID),
                .flags = BLE_GATT_CHR_F_WRITE | BLE_GATT_CHR_F_WRITE_NO_RSP,
                .access_cb = configuration_write
            },
            {
                .uuid = BLE_UUID16_DECLARE(READ_MONITOR_CHAR_UUID),
                .flags = BLE_GATT_CHR_F_READ,
                .access_cb = monitor_read
            },
            {
                .uuid = BLE_UUID16_DECLARE(READ_ONE_WIRE_CHAR_UUID),
                .flags = BLE_GATT_CHR_F_READ,
                .access_cb = one_wire_read
            },
            {0}
        }
    },
    {0}
}
```

GATT service definitions for the BLE server. Defines a primary service with characteristics for configuration read/write, monitor data read, and one-wire sensor data read.

Definition at line 232 of file ble.c.

4.7.2.6 TAG

```
char* TAG = "BLE-Server"
```

Tag for logging messages from the BLE server module.

Definition at line 25 of file ble.c.

4.8 ble.c

Go to the documentation of this file.

```
00001 #include "ble.h"
00002 #include <stdio.h>
00003 #include "freertos/FreeRTOS.h"
00004 #include "freertos/task.h"
00005 #include "nvs_flash.h"
00006 #include "esp_log.h"
00007 #include "nimble/nimble_port.h"
00008 #include "nimble/nimble_port_freertos.h"
00009 #include "host/ble_hs.h"
00010 #include "services/gap/ble_svc_gap.h"
00011 #include "services/gatt/ble_svc_gatt.h"
00012 #include "driver/gpio.h"
00013
00014 #include "esp_mac.h"
00015
00016 #include "conf_task_manager.h"
00017 #include "nvs_utils.h"
```

```
00018
00019 #include "variables.h"
00020 #include "one_wire_detect.h"
00021
00025 char *TAG = "BLE-Server";
00026
00031 static uint16_t ble_mtu = 23; // Default minimum MTU
00032
00036 uint8_t ble_addr_type;
00037
00041 void ble_app_advertise(void);
00042
00047 uint16_t conn_handle = BLE_HS_CONN_HANDLE_NONE;
00048
00052 bool app_connected_ble = false;
00053
00063 static int configuration_read(uint16_t conn_handle, uint16_t attr_handle, struct ble_gatt_access_ctxt
      *ctxt, void *arg) {
00064     static char *nvs_data = NULL;
00065     static size_t nvs_data_len = 0;
00066     static size_t offset = 0;
00067
00068     // Load configuration from NVS if not already loaded
00069     if (nvs_data == NULL) {
00070         ESP_LOGI(TAG, "Client requested configuration");
00071         esp_err_t ret = load_config_from_nvs(&nvs_data, &nvs_data_len);
00072         if (ret != ESP_OK || nvs_data == NULL) {
00073             ESP_LOGE(TAG, "Failed to load config from NVS");
00074             return 0;
00075         }
00076     }
00077
00078     // Check if all data has been sent
00079     if (offset >= nvs_data_len) {
00080         // All data sent, return empty response to signal end
00081         ESP_LOGI(TAG, "Configuration sent successfully. (End of data reached, sending empty
      response)");
00082         if (nvs_data != NULL) {
00083             free(nvs_data); // Free allocated memory
00084             nvs_data = NULL;
00085             nvs_data_len = 0;
00086             offset = 0;
00087         }
00088         return 0; // Success, no more data to send
00089     }
00090
00091     // Calculate chunk size based on remaining data and MTU
00092     size_t remaining = nvs_data_len - offset;
00093     size_t current_chunk_size = (remaining > (ble_mtu - 3)) ? (ble_mtu - 3) : remaining;
00094
00095     // Append chunk to response buffer
00096     int rc = os_mbuf_append(ctxt->om, &nvs_data[offset], current_chunk_size);
00097     if (rc != 0) {
00098         ESP_LOGE(TAG, "Failed to append data to mbuf: %d", rc);
00099         return BLE_ATT_ERR_INSUFFICIENT_RES;
00100     }
00101
00102     offset += current_chunk_size; // Update offset for next read
00103     return 0;
00104 }
00105
00115 static int configuration_write(uint16_t conn_handle, uint16_t attr_handle, struct ble_gatt_access_ctxt
      *ctxt, void *arg) {
00116     const char *data = (const char *)ctxt->om->om_data;
00117     configure(data, ctxt->om->om_len, false); // Apply configuration
00118     return 0;
00119 }
00120
00130 static int monitor_read(uint16_t conn_handle, uint16_t attr_handle, struct ble_gatt_access_ctxt *ctxt,
      void *arg) {
00131     static char *monitor_data = NULL;
00132     static size_t monitor_data_len = 0;
00133     static size_t monitor_offset = 0;
00134
00135     // Load monitor data if not already loaded
00136     if (monitor_data == NULL) {
00137         monitor_data = read_variables_json(); // Read variables as JSON
00138         if (monitor_data == NULL) {
00139             ESP_LOGI(TAG, "No monitor data available");
00140             return 0;
00141         }
00142         monitor_data_len = strlen(monitor_data);
00143         monitor_offset = 0;
00144     }
00145
00146     // Check if all data has been sent
00147     if (monitor_offset >= monitor_data_len) {
```

```
00148         free(monitor_data); // Free allocated memory
00149         monitor_data = NULL;
00150         monitor_data_len = 0;
00151         monitor_offset = 0;
00152         return 0; // Empty response signals end
00153     }
00154 
00155     // Calculate chunk size based on remaining data and MTU
00156     size_t remaining = monitor_data_len - monitor_offset;
00157     size_t chunk_size = (remaining > (ble_mtu - 3)) ? (ble_mtu - 3) : remaining;
00158 
00159     // Append chunk to response buffer
00160     int rc = os_mbuf_append(ctxt->om, &monitor_data[monitor_offset], chunk_size);
00161     if (rc != 0) {
00162         ESP_LOGE(TAG, "Failed to append monitor data to mbuf: %d", rc);
00163         free(monitor_data); // Free memory on error
00164         monitor_data = NULL;
00165         monitor_data_len = 0;
00166         monitor_offset = 0;
00167         return BLE_ATT_ERR_INSUFFICIENT_RES;
00168     }
00169 
00170     monitor_offset += chunk_size; // Update offset for next read
00171     return 0;
00172 }
00173 
00183 static int one_wire_read(uint16_t conn_handle, uint16_t attr_handle, struct ble_gatt_access_ctxt
    *ctxt, void *arg) {
00184     static char *one_wire_data = NULL;
00185     static size_t one_wire_data_len = 0;
00186     static size_t one_wire_offset = 0;
00187 
00188     // Load one-wire sensor data if not already loaded
00189     if (one_wire_data == NULL) {
00190         one_wire_data = search_for_one_wire_sensors(); // Read one-wire sensor data
00191         if (one_wire_data == NULL) {
00192             ESP_LOGI(TAG, "No one-wire data available");
00193             return 0;
00194         }
00195         one_wire_data_len = strlen(one_wire_data);
00196         one_wire_offset = 0;
00197     }
00198 
00199     // Check if all data has been sent
00200     if (one_wire_offset >= one_wire_data_len) {
00201         free(one_wire_data); // Free allocated memory
00202         one_wire_data = NULL;
00203         one_wire_data_len = 0;
00204         one_wire_offset = 0;
00205         return 0; // Empty response signals end
00206     }
00207 
00208     // Calculate chunk size based on remaining data and MTU
00209     size_t remaining = one_wire_data_len - one_wire_offset;
00210     size_t chunk_size = (remaining > (ble_mtu - 3)) ? (ble_mtu - 3) : remaining;
00211 
00212     // Append chunk to response buffer
00213     int rc = os_mbuf_append(ctxt->om, &one_wire_data[one_wire_offset], chunk_size);
00214     if (rc != 0) {
00215         ESP_LOGE(TAG, "Failed to append one-wire data to mbuf: %d", rc);
00216         free(one_wire_data); // Free memory on error
00217         one_wire_data = NULL;
00218         one_wire_data_len = 0;
00219         one_wire_offset = 0;
00220         return BLE_ATT_ERR_INSUFFICIENT_RES;
00221     }
00222 
00223     one_wire_offset += chunk_size; // Update offset for next read
00224     return 0;
00225 }
00226 
00232 static const struct ble_gatt_svc_def gatt_svcs[] = {
00233     {
00234         .type = BLE_GATT_SVC_TYPE_PRIMARY,
00235         .uuid = BLE_UUID16_DECLARE(SERVICE_UUID),
00236         .characteristics = (struct ble_gatt_chr_def[]){
00237             {
00238                 .uuid = BLE_UUID16_DECLARE(READ_CONFIGURATION_CHAR_UUID),
00239                 .flags = BLE_GATT_CHR_F_READ,
00240                 .access_cb = configuration_read // Read configuration
00241             },
00242             {
00243                 .uuid = BLE_UUID16_DECLARE(WRITE_CONFIGURATION_CHAR_UUID),
00244                 .flags = BLE_GATT_CHR_F_WRITE | BLE_GATT_CHR_F_WRITE_NO_RSP,
00245                 .access_cb = configuration_write // Write configuration
00246             },
00247             {
```

```
00248                 .uuid = BLE_UUID16_DECLARE(READ_MONITOR_CHAR_UUID),
00249                 .flags = BLE_GATT_CHR_F_READ,
00250                 .access_cb = monitor_read // Read monitor data
00251             },
00252             {
00253                 .uuid = BLE_UUID16_DECLARE(READ_ONE_WIRE_CHAR_UUID),
00254                 .flags = BLE_GATT_CHR_F_READ,
00255                 .access_cb = one_wire_read // Read one-wire sensor data
00256             },
00257             {0} // Terminator for characteristics array
00258         }
00259     },
00260     {0} // Terminator for services array
00261 };
00262
00269 static int ble_gap_event(struct ble_gap_event *event, void *arg) {
00270     switch (event->type) {
00271     case BLE_GAP_EVENT_CONNECT:
00272         // Handle connection event
00273         ESP_LOGI(TAG, "EVENT CONNECT %s, conn_handle=%d", event->connect.status == 0 ? "OK!" :
      "FAILED!", event->connect.conn_handle);
00274         if (event->connect.status == 0) {
00275             conn_handle = event->connect.conn_handle;  // Store connection handle
00276             ESP_LOGI(TAG, "Client connected successfully");
00277             app_connected_ble = true; // Set BLE connection flag
00278         } else {
00279             ble_app_advertise(); // Restart advertising on connection failure
00280         }
00281         break;
00282     case BLE_GAP_EVENT_DISCONNECT:
00283         // Handle disconnection event
00284         if (event->disconnect.conn.conn_handle == conn_handle) {
00285             ESP_LOGI(TAG, "EVENT DISCONNECT, reason=%d, conn_handle=%d", event->disconnect.reason,
      event->disconnect.conn.conn_handle);
00286             conn_handle = BLE_HS_CONN_HANDLE_NONE;  // Reset connection handle
00287             app_connected_ble = false; // Clear BLE connection flag
00288         } else {
00289             ESP_LOGW(TAG, "Other disconnect, conn_handle: %d", conn_handle);
00290         }
00291         ble_app_advertise(); // Restart advertising
00292         break;
00293     case BLE_GAP_EVENT_ADV_COMPLETE:
00294         // Handle advertising completion event
00295         ESP_LOGI(TAG, "EVENT ADV_COMPLETE");
00296         ble_app_advertise(); // Restart advertising
00297         break;
00298     case BLE_GAP_EVENT_MTU:
00299         // Handle MTU update event
00300         ESP_LOGI(TAG, "MTU updated: %d", event->mtu.value);
00301         ble_mtu = event->mtu.value; // Update MTU size
00302         break;
00303     default:
00304         // Log unhandled GAP events
00305         ESP_LOGI(TAG, "Unhandled GAP event: %d", event->type);
00306         break;
00307     }
00308     return 0;
00309 }
00310
00314 void ble_app_advertise(void)
00315 {
00316     struct ble_hs_adv_fields fields;
00317
00318     // Initialize advertising fields
00319     memset(&fields, 0, sizeof(fields));
00320     const char *device_name = ble_svc_gap_device_name();
00321     fields.name = (uint8_t *)device_name;
00322     fields.name_len = strlen(device_name);
00323     fields.name_is_complete = 1;
00324
00325     // Set service UUID for advertising
00326     ble_uuid16_t uuid16 = BLE_UUID16_INIT(SERVICE_UUID);
00327     fields.uuids16 = &uuid16;
00328     fields.num_uuids16 = 1;
00329     fields.uuids16_is_complete = 1;
00330
00331     // Set advertising fields
00332     int rc = ble_gap_adv_set_fields(&fields);
00333     if (rc != 0) {
00334         ESP_LOGE(TAG, "Failed to set advertising fields: %d", rc);
00335         return;
00336     }
00337
00338     // Configure advertising parameters
00339     struct ble_gap_adv_params adv_params;
00340     memset(&adv_params, 0, sizeof(adv_params));
00341     adv_params.conn_mode = BLE_GAP_CONN_MODE_UND; // Undirected connectable mode
```

```
00342     adv_params.disc_mode = BLE_GAP_DISC_MODE_GEN; // General discoverable mode
00343     adv_params.itvl_min = 0x20;  // Minimum advertising interval
00344     adv_params.itvl_max = 0x40;  // Maximum advertising interval
00345
00346     // Start advertising
00347     rc = ble_gap_adv_start(ble_addr_type, NULL, BLE_HS_FOREVER, &adv_params, ble_gap_event, NULL);
00348     if (rc != 0) {
00349         ESP_LOGE(TAG, "Advertising start failed: %d", rc);
00350     } else {
00351         ESP_LOGI(TAG, "Advertising started successfully");
00352     }
00353 }
00354
00359 void ble_app_on_sync(void)
00360 {
00361     ESP_LOGI(TAG, "BLE sync completed");
00362     vTaskDelay(pdMS_TO_TICKS(1000)); // Delay to ensure stability
00363     if (ble_hs_id_infer_auto(0, &ble_addr_type) != 0) {
00364         ESP_LOGE(TAG, "Failed to infer BLE address type!");
00365         return;
00366     }
00367     ble_app_advertise(); // Start advertising
00368 }
00369
00375 void host_task(void *param)
00376 {
00377     nimble_port_run(); // Run the NimBLE host stack
00378 }
00379
00384 static void set_ble_name_from_mac() {
00385     uint8_t mac[6];
00386     esp_read_mac(mac, ESP_MAC_BT); // Read Bluetooth MAC address
00387     char dev_name[11];
00388     snprintf(dev_name, sizeof(dev_name), "ESP_%02X%02X%02X", mac[0], mac[1], mac[2]); // Format device
      name
00389     int rc = ble_svc_gap_device_name_set(dev_name); // Set device name
00390     if (rc != 0) ESP_LOGE(TAG, "Failed to set device name: %d", rc);
00391 }
00392
00397 void ble_init(void) {
00398     // Initialize NimBLE stack
00399     ESP_ERROR_CHECK(nimble_port_init());
00400     set_ble_name_from_mac(); // Set device name based on MAC
00401     ble_svc_gap_init(); // Initialize GAP service
00402     ble_svc_gatt_init(); // Initialize GATT service
00403     ble_gatts_count_cfg(gatt_svcs); // Configure GATT services
00404     ESP_ERROR_CHECK(ble_gatts_add_svcs(gatt_svcs)); // Add GATT services
00405     ble_hs_cfg.sync_cb = ble_app_on_sync; // Set synchronization callback
00406
00407     // Start NimBLE host task
00408     nimble_port_freertos_init(host_task);
00409 }
```

4.9 main/ble.h File Reference

```
#include <stdint.h>
#include "nimble/nimble_port.h"
#include "host/ble_hs.h"
```

Macros

- #define SERVICE_UUID 0x1234

  UUID for the BLE service.
- #define READ_CONFIGURATION_CHAR_UUID 0xFFF1

  UUID for the read configuration characteristic.
- #define WRITE_CONFIGURATION_CHAR_UUID 0xFFF2

  UUID for the write configuration characteristic.
- #define READ_MONITOR_CHAR_UUID 0xFFF3

  UUID for the read monitor characteristic.
- #define READ_ONE_WIRE_CHAR_UUID 0xFFF4

  UUID for the read one-wire characteristic.

Functions

- void ble_init (void)

  Initializes the BLE subsystem.

Variables

- char * monitor_data

  Pointer to the monitor data buffer.
- size_t monitor_data_len

  Length of the monitor data buffer.
- size_t monitor_offset

  Current offset in the monitor data buffer.
- bool monitor_reading

  Flag indicating if monitor data is being read.
- bool app_connected_ble

  Flag indicating if the BLE application is connected.

4.9.1 Macro Definition Documentation

4.9.1.1 READ_CONFIGURATION_CHAR_UUID

```
#define READ_CONFIGURATION_CHAR_UUID 0xFFF1
```

UUID for the read configuration characteristic.

Definition at line 16 of file ble.h.

4.9.1.2 READ_MONITOR_CHAR_UUID

```
#define READ_MONITOR_CHAR_UUID 0xFFF3
```

UUID for the read monitor characteristic.

Definition at line 26 of file ble.h.

4.9.1.3 READ_ONE_WIRE_CHAR_UUID

```
#define READ_ONE_WIRE_CHAR_UUID 0xFFF4
```

UUID for the read one-wire characteristic.

Definition at line 31 of file ble.h.

4.9.1.4 SERVICE_UUID

```
#define SERVICE_UUID 0x1234
```

UUID for the BLE service.

Definition at line 11 of file ble.h.

4.9.1.5 WRITE_CONFIGURATION_CHAR_UUID

```
#define WRITE_CONFIGURATION_CHAR_UUID 0xFFF2
```

UUID for the write configuration characteristic.

Definition at line 21 of file ble.h.

4.9.2 Function Documentation

4.9.2.1 ble_init()

```
void ble_init (
            void )
```

Initializes the BLE subsystem.

Initializes the BLE subsystem.

Definition at line 397 of file ble.c.

4.9.3 Variable Documentation

4.9.3.1 app_connected_ble

```
bool app_connected_ble  [extern]
```

Flag indicating if the BLE application is connected.

Flag indicating if the BLE application is connected.

Definition at line 52 of file ble.c.

4.9.3.2 monitor_data

```
char* monitor_data  [extern]
```

Pointer to the monitor data buffer.

4.9.3.3 monitor_data_len

```
size_t monitor_data_len  [extern]
```

Length of the monitor data buffer.

4.9.3.4 monitor_offset

```
size_t monitor_offset  [extern]
```

Current offset in the monitor data buffer.

4.9.3.5 monitor_reading

```
bool monitor_reading  [extern]
```

Flag indicating if monitor data is being read.

4.10 ble.h

Go to the documentation of this file.

```
00001 #ifndef BLE_H
00002 #define BLE_H
00003
00004 #include <stdint.h>
00005 #include "nimble/nimble_port.h"
00006 #include "host/ble_hs.h"
00007
00011 #define SERVICE_UUID                    0x1234
00012
00016 #define READ_CONFIGURATION_CHAR_UUID    0xFFF1
00017
00021 #define WRITE_CONFIGURATION_CHAR_UUID   0xFFF2
00022
00026 #define READ_MONITOR_CHAR_UUID        0xFFF3
00027
00031 #define READ_ONE_WIRE_CHAR_UUID       0xFFF4
00032
00036 extern char *monitor_data;
00037
00041 extern size_t monitor_data_len;
00042
00046 extern size_t monitor_offset;
00047
00051 extern bool monitor_reading;
00052
00056 extern bool app_connected_ble;
00057
00061 void ble_init(void);
00062
00063 #endif // BLE_H
```

4.11 main/conf_task_manager.c File Reference

```
#include "conf_task_manager.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/timers.h"
#include "esp_log.h"
#include "string.h"
#include <cJSON.h>
#include "nvs_utils.h"
#include "device_config.h"
#include "variables.h"
#include "ladder_elements.h"
```

Data Structures

- struct TaskInfo

  Structure to store task information.

Functions

- static void config_timeout_callback (TimerHandle_t xTimer)

  Callback function for configuration timeout.
- static bool process_node (cJSON ∗node, bool ∗condition)

  Processes a single ladder node (excluding Coil nodes).
- static bool process_nodes (cJSON ∗nodes, bool ∗condition, cJSON ∗∗last_coil)

  Processes an array of ladder nodes, identifying the last coil if present.
- static void process_coil (cJSON ∗node, bool condition)

  Processes a Coil node.
- static void process_block_task (void ∗pvParameters)

  Main task function to process a wire (ladder logic block).
- void delete_all_tasks (void)

  Deletes all tasks and cleans up associated resources.
- void configure (const char ∗data, int data_len, bool loaded_from_nvs)

  Configures tasks based on provided data.

Variables

- static const char ∗ TAG = "conf_task_manager"

  Tag for logging messages from the configuration task manager module.
- static const int CONFIG_TIMEOUT_MS = 10000

  Timeout duration for configuration data reception (10 seconds).
- static TimerHandle_t config_timeout_timer = NULL

  Timer handle for configuration timeout.
- static char ∗ large_buffer = NULL

  Dynamic buffer for storing configuration data parts.
- static size_t total_received = 0

  Total number of bytes received in the buffer.
- static TaskInfo ∗ tasks = NULL

  Array of task information structures.
- static int num_tasks = 0

  Number of tasks currently managed.

4.11.1 Function Documentation

4.11.1.1 config_timeout_callback()

```
static void config_timeout_callback (
            TimerHandle_t xTimer)  [static]
```

Callback function for configuration timeout.

Parameters

xTimer	Handle of the timer that triggered the callback.

Definition at line 61 of file conf_task_manager.c.

4.11.1.2 configure()

```
void configure (
            const char * data,
            int data_len,
            bool loaded_from_nvs)
```

Configures tasks based on provided data.

Parameters

data	Pointer to the configuration data.
data_len	Length of the configuration data.
loaded_from_nvs	Indicates if the data was loaded from non-volatile storage.

Definition at line 427 of file conf_task_manager.c.

4.11.1.3 delete_all_tasks()

```
void delete_all_tasks (
            void )
```

Deletes all tasks and cleans up associated resources.

Deletes all configured tasks.

Definition at line 390 of file conf_task_manager.c.

4.11.1.4 process_block_task()

```
static void process_block_task (
            void * pvParameters)  [static]
```

Main task function to process a wire (ladder logic block).

Parameters

pvParameters	Pointer to the JSON wire configuration.

Definition at line 353 of file conf_task_manager.c.

4.11.1.5 process_coil()

```
static void process_coil (
            cJSON * node,
            bool condition)  [static]
```

Processes a Coil node.

Parameters

node	JSON object representing the coil node.
condition	Current condition state.

Definition at line 296 of file conf_task_manager.c.

4.11.1.6 process_node()

```
static bool process_node (
            cJSON * node,
            bool * condition)  [static]
```

Processes a single ladder node (excluding Coil nodes).

Parameters

node	JSON object representing the node.
condition	Pointer to the current condition state.

Returns

bool Updated condition state or false on error.

Definition at line 82 of file conf_task_manager.c.

4.11.1.7 process_nodes()

```
static bool process_nodes (
            cJSON * nodes,
            bool * condition,
            cJSON ** last_coil)  [static]
```

Processes an array of ladder nodes, identifying the last coil if present.

Parameters

nodes	JSON array of nodes.
condition	Pointer to the current condition state.
last_coil	Pointer to store the last coil node, if any.

Returns

bool Updated condition state or false if the node list is empty or invalid.

Definition at line 247 of file conf_task_manager.c.

4.11.2 Variable Documentation

4.11.2.1 CONFIG_TIMEOUT_MS

```
const int CONFIG_TIMEOUT_MS = 10000  [static]
```

Timeout duration for configuration data reception (10 seconds).

Definition at line 22 of file conf_task_manager.c.

4.11.2.2 config_timeout_timer

```
TimerHandle_t config_timeout_timer = NULL  [static]
```

Timer handle for configuration timeout.

Definition at line 27 of file conf_task_manager.c.

4.11.2.3 large_buffer

```
char* large_buffer = NULL  [static]
```

Dynamic buffer for storing configuration data parts.

Definition at line 32 of file conf_task_manager.c.

4.11.2.4 num_tasks

```
int num_tasks = 0  [static]
```

Number of tasks currently managed.

Definition at line 55 of file conf_task_manager.c.

4.11.2.5 TAG

```
const char* TAG = "conf_task_manager"  [static]
```

Tag for logging messages from the configuration task manager module.

Definition at line 17 of file conf_task_manager.c.

4.11.2.6 tasks

```
TaskInfo* tasks = NULL  [static]
```

Array of task information structures.

Definition at line 50 of file conf_task_manager.c.

4.11.2.7 total_received

```
size_t total_received = 0  [static]
```

Total number of bytes received in the buffer.

Definition at line 37 of file conf_task_manager.c.

4.12 conf_task_manager.c

Go to the documentation of this file.

```
00001 #include "conf_task_manager.h"
00002 #include "freertos/FreeRTOS.h"
00003 #include "freertos/task.h"
00004 #include "freertos/timers.h"
00005 #include "esp_log.h"
00006 #include "string.h"
00007 #include <cJSON.h>
00008
00009 #include "nvs_utils.h"
00010 #include "device_config.h"
00011 #include "variables.h"
00012 #include "ladder_elements.h"
00013
00017 static const char *TAG = "conf_task_manager";
00018
00022 static const int CONFIG_TIMEOUT_MS = 10000; // 10 seconds timeout
00023
00027 static TimerHandle_t config_timeout_timer = NULL;
00028
00032 static char *large_buffer = NULL;
00033
00037 static size_t total_received = 0;
00038
00042 typedef struct {
00043     TaskHandle_t handle;
00044     cJSON *wire_copy;
00045 } TaskInfo;
00046
00050 static TaskInfo *tasks = NULL;
00051
00055 static int num_tasks = 0;
00056
00061 static void config_timeout_callback(TimerHandle_t xTimer) {
00062     // Log warning and clear buffer on timeout
00063     ESP_LOGW(TAG, "Configuration timeout - clearing buffer");
00064     if (large_buffer) {
00065         free(large_buffer);
00066         large_buffer = NULL;
00067         total_received = 0;
00068     }
00069 }
00070
00071 // Forward declarations
00072 static bool process_node(cJSON *node, bool *condition);
00073 static bool process_nodes(cJSON *nodes, bool *condition, cJSON **last_coil);
00074 static void process_coil(cJSON *node, bool condition);
00075
00082 static bool process_node(cJSON *node, bool *condition) {
00083     if (!node || !condition || !cJSON_IsObject(node)) {
00084         // Log error for invalid node or condition
00085         ESP_LOGE(TAG, "Invalid node or condition");
00086         return false;
00087     }
00088
00089     cJSON *type = cJSON_GetObjectItem(node, "Type");
00090     if (!cJSON_IsString(type)) {
00091         // Log error if node type is missing or invalid
00092         ESP_LOGE(TAG, "Node missing Type or Type is not a string");
00093         return false;
00094     }
00095
00096     if (strcmp(type->valuestring, "LadderElement") == 0) {
00097         cJSON *element_type = cJSON_GetObjectItem(node, "ElementType");
00098         cJSON *combo_values = cJSON_GetObjectItem(node, "ComboBoxValues");
00099
00100         if (!cJSON_IsString(element_type) || !cJSON_IsArray(combo_values)) {
```

```
00101             // Log error if LadderElement is missing required fields
00102             ESP_LOGE(TAG, "LadderElement missing ElementType or ComboBoxValues");
00103             return false;
00104         }
00105 
00106         // Get arguments from ComboBoxValues
00107         cJSON *arg1 = cJSON_GetArrayItem(combo_values, 0);
00108         cJSON *arg2 = cJSON_GetArrayItem(combo_values, 1);
00109         cJSON *arg3 = cJSON_GetArrayItem(combo_values, 2);
00110 
00111         const char *var1 = (arg1 && cJSON_IsString(arg1)) ? arg1->valuestring : NULL;
00112         const char *var2 = (arg2 && cJSON_IsString(arg2)) ? arg2->valuestring : NULL;
00113         const char *var3 = (arg3 && cJSON_IsString(arg3)) ? arg3->valuestring : NULL;
00114 
00115         // Handle Contacts and Comparisons
00116         if (strcmp(element_type->valuestring, "NOContact") == 0 && var1) {
00117             bool result = no_contact(var1);
00118             ESP_LOGD(TAG, "NOContact(%s) = %d", var1, result);
00119             *condition &= result;
00120             return *condition;
00121         } else if (strcmp(element_type->valuestring, "NCContact") == 0 && var1) {
00122             bool result = nc_contact(var1);
00123             ESP_LOGD(TAG, "NCContact(%s) = %d", var1, result);
00124             *condition &= result;
00125             return *condition;
00126         } else if (strcmp(element_type->valuestring, "GreaterCompare") == 0 && var1 && var2) {
00127             bool result = greater(var1, var2);
00128             ESP_LOGD(TAG, "GreaterCompare(%s, %s) = %d", var1, var2, result);
00129             *condition &= result;
00130             return *condition;
00131         } else if (strcmp(element_type->valuestring, "LessCompare") == 0 && var1 && var2) {
00132             bool result = less(var1, var2);
00133             ESP_LOGD(TAG, "LessCompare(%s, %s) = %d", var1, var2, result);
00134             *condition &= result;
00135             return *condition;
00136         } else if (strcmp(element_type->valuestring, "GreaterOrEqualCompare") == 0 && var1 && var2) {
00137             bool result = greater_or_equal(var1, var2);
00138             ESP_LOGD(TAG, "GreaterOrEqualCompare(%s, %s) = %d", var1, var2, result);
00139             *condition &= result;
00140             return *condition;
00141         } else if (strcmp(element_type->valuestring, "LessOrEqualCompare") == 0 && var1 && var2) {
00142             bool result = less_or_equal(var1, var2);
00143             ESP_LOGD(TAG, "LessOrEqualCompare(%s, %s) = %d", var1, var2, result);
00144             *condition &= result;
00145             return *condition;
00146         } else if (strcmp(element_type->valuestring, "EqualCompare") == 0 && var1 && var2) {
00147             bool result = equal(var1, var2);
00148             ESP_LOGD(TAG, "EqualCompare(%s, %s) = %d", var1, var2, result);
00149             *condition &= result;
00150             return *condition;
00151         } else if (strcmp(element_type->valuestring, "NotEqualCompare") == 0 && var1 && var2) {
00152             bool result = not_equal(var1, var2);
00153             ESP_LOGD(TAG, "NotEqualCompare(%s, %s) = %d", var1, var2, result);
00154             *condition &= result;
00155             return *condition;
00156         }
00157         // Actions executed immediately if condition is true
00158         else if (strcmp(element_type->valuestring, "AddMath") == 0 && var1 && var2 && var3 &&
      *condition) {
00159             add(var1, var2, var3, *condition);
00160             return *condition;
00161         } else if (strcmp(element_type->valuestring, "SubtractMath") == 0 && var1 && var2 && var3 &&
      *condition) {
00162             subtract(var1, var2, var3, *condition);
00163             return *condition;
00164         } else if (strcmp(element_type->valuestring, "MultiplyMath") == 0 && var1 && var2 && var3 &&
      *condition) {
00165             multiply(var1, var2, var3, *condition);
00166             return *condition;
00167         } else if (strcmp(element_type->valuestring, "DivideMath") == 0 && var1 && var2 && var3 &&
      *condition) {
00168             divide(var1, var2, var3, *condition);
00169             return *condition;
00170         } else if (strcmp(element_type->valuestring, "MoveMath") == 0 && var1 && var2 && *condition) {
00171             move(var1, var2, *condition);
00172             return *condition;
00173         } else if (strcmp(element_type->valuestring, "CountUp") == 0 && var1) {
00174             count_up(var1, *condition);
00175             return *condition;
00176         } else if (strcmp(element_type->valuestring, "CountDown") == 0 && var1) {
00177             count_down(var1, *condition);
00178             return *condition;
00179         } else if (strcmp(element_type->valuestring, "OnDelayTimer") == 0 && var1) {
00180             bool result = timer_on(var1, *condition);
00181             *condition &= result;
00182             return *condition;
00183         } else if (strcmp(element_type->valuestring, "OffDelayTimer") == 0 && var1) {
```

```
00184             bool result = timer_off(var1, *condition);
00185             // Note: Uses = instead of &= because this timer sets true regardless of prior elements
00186             *condition = result;
00187             return *condition;
00188         } else if (strcmp(element_type->valuestring, "Reset") == 0 && var1 && *condition) {
00189             reset(var1, *condition);
00190             return *condition;
00191         }
00192         return *condition; // Unknown element doesn't change condition
00193     } else if (strcmp(type->valuestring, "Branch") == 0) {
00194         cJSON *nodes1 = cJSON_GetObjectItem(node, "Nodes1");
00195         cJSON *nodes2 = cJSON_GetObjectItem(node, "Nodes2");
00196 
00197         if (!cJSON_IsArray(nodes1) || !cJSON_IsArray(nodes2)) {
00198             // Log error if Branch is missing required node arrays
00199             ESP_LOGE(TAG, "Branch missing Nodes1 or Nodes2 arrays");
00200             return false;
00201         }
00202 
00203         bool nodes1_condition = true; // Independent condition for Nodes1
00204         bool nodes2_condition = true; // Independent condition for Nodes2
00205         cJSON *nodes1_last_coil = NULL;
00206         cJSON *nodes2_last_coil = NULL;
00207 
00208         // Process both branches
00209         bool nodes1_active = process_nodes(nodes1, &nodes1_condition, &nodes1_last_coil);
00210         bool nodes2_active = process_nodes(nodes2, &nodes2_condition, &nodes2_last_coil);
00211 
00212         // Log branch conditions
00213         ESP_LOGD(TAG, "Branch: Nodes1_active=%d (cond=%d), Nodes2_active=%d (cond=%d)",
00214                  nodes1_active, nodes1_condition, nodes2_active, nodes2_condition);
00215 
00216         // OR logic: at least one branch must be active
00217         bool branch_condition = nodes1_active || nodes2_active;
00218         *condition &= branch_condition;
00219 
00220         // Process coils only if present (shouldn't be in this JSON, but handle for robustness)
00221         if (nodes1_last_coil && nodes1_condition) {
00222             // Log warning for unexpected coil in Nodes1
00223             ESP_LOGW(TAG, "Unexpected coil in Nodes1");
00224             process_coil(nodes1_last_coil, nodes1_condition);
00225         }
00226         if (nodes2_last_coil && nodes2_condition) {
00227             // Log warning for unexpected coil in Nodes2
00228             ESP_LOGW(TAG, "Unexpected coil in Nodes2");
00229             process_coil(nodes2_last_coil, nodes2_condition);
00230         }
00231 
00232         return *condition;
00233     }
00234 
00235     // Log warning for unknown node type
00236     ESP_LOGW(TAG, "Unknown node type: %s", type->valuestring);
00237     return false;
00238 }
00239 
00247 static bool process_nodes(cJSON *nodes, bool *condition, cJSON **last_coil) {
00248     if (!nodes || !cJSON_IsArray(nodes) || !condition || !last_coil) {
00249         // Log error for invalid nodes array or parameters
00250         ESP_LOGE(TAG, "Invalid nodes array or parameters");
00251         return false;
00252     }
00253 
00254     *last_coil = NULL;
00255     bool all_conditions_met = *condition;
00256     int node_count = cJSON_GetArraySize(nodes);
00257 
00258     if (node_count == 0) {
00259         // Empty node list
00260         return false;
00261     }
00262 
00263     // Check if the last node is a coil
00264     cJSON *last_node = cJSON_GetArrayItem(nodes, node_count - 1);
00265     cJSON *last_type = cJSON_GetObjectItem(last_node, "Type");
00266     cJSON *last_element_type = cJSON_GetObjectItem(last_node, "ElementType");
00267 
00268     if (last_type && cJSON_IsString(last_type) &&
00269         strcmp(last_type->valuestring, "LadderElement") == 0 &&
00270         last_element_type && cJSON_IsString(last_element_type) &&
00271         (strcmp(last_element_type->valuestring, "Coil") == 0 ||
00272          strcmp(last_element_type->valuestring, "OneShotPositiveCoil") == 0 ||
00273          strcmp(last_element_type->valuestring, "SetCoil") == 0 ||
00274          strcmp(last_element_type->valuestring, "ResetCoil") == 0)) {
00275         *last_coil = last_node;
00276         node_count--; // Exclude the coil from condition processing
00277     }
```

```
00278
00279     // Process all nodes except the last coil (if it was a coil)
00280     for (int i = 0; i < node_count; i++) {
00281         cJSON *node = cJSON_GetArrayItem(nodes, i);
00282         all_conditions_met = process_node(node, &all_conditions_met);
00283     }
00284
00285     *condition = all_conditions_met;
00286     // Log processed nodes condition
00287     ESP_LOGD(TAG, "Nodes processed, condition=%d", *condition);
00288     return all_conditions_met;
00289 }
00290
00296 static void process_coil(cJSON *node, bool condition) {
00297     if (!node || !cJSON_IsObject(node)) {
00298         // Log error for invalid coil node
00299         ESP_LOGE(TAG, "Invalid coil node");
00300         return;
00301     }
00302
00303     cJSON *type = cJSON_GetObjectItem(node, "Type");
00304     if (!cJSON_IsString(type) || strcmp(type->valuestring, "LadderElement") != 0) {
00305         // Log error if coil is not a LadderElement
00306         ESP_LOGE(TAG, "Coil node is not a LadderElement");
00307         return;
00308     }
00309
00310     cJSON *element_type = cJSON_GetObjectItem(node, "ElementType");
00311     cJSON *combo_values = cJSON_GetObjectItem(node, "ComboBoxValues");
00312
00313     if (!cJSON_IsString(element_type) || !cJSON_IsArray(combo_values)) {
00314         // Log error if coil is missing required fields
00315         ESP_LOGE(TAG, "Coil missing ElementType or ComboBoxValues");
00316         return;
00317     }
00318
00319     cJSON *arg1 = cJSON_GetArrayItem(combo_values, 0);
00320     const char *var1 = (arg1 && cJSON_IsString(arg1)) ? arg1->valuestring : NULL;
00321
00322     if (var1) {
00323         if (strcmp(element_type->valuestring, "Coil") == 0) {
00324             // Log and process standard coil
00325             ESP_LOGD(TAG, "Coil(%s, %d)", var1, condition);
00326             coil(var1, condition);
00327         } else if (strcmp(element_type->valuestring, "OneShotPositiveCoil") == 0) {
00328             // Log and process one-shot positive coil
00329             ESP_LOGD(TAG, "OneShotPositiveCoil(%s, %d)", var1, condition);
00330             one_shot_positive_coil(var1, condition);
00331         } else if (strcmp(element_type->valuestring, "SetCoil") == 0) {
00332             // Log and process set coil
00333             ESP_LOGD(TAG, "SetCoil(%s, %d)", var1, condition);
00334             set_coil(var1, condition);
00335         } else if (strcmp(element_type->valuestring, "ResetCoil") == 0) {
00336             // Log and process reset coil
00337             ESP_LOGD(TAG, "ResetCoil(%s, %d)", var1, condition);
00338             reset_coil(var1, condition);
00339         } else {
00340             // Log warning for unknown coil type
00341             ESP_LOGW(TAG, "Unknown coil type: %s", element_type->valuestring);
00342         }
00343     } else {
00344         // Log error if coil is missing variable name
00345         ESP_LOGE(TAG, "Coil missing variable name");
00346     }
00347 }
00348
00353 static void process_block_task(void *pvParameters) {
00354     cJSON *wire = (cJSON *)pvParameters;
00355
00356     // Get Nodes
00357     cJSON *nodes = cJSON_GetObjectItem(wire, "Nodes");
00358     if (!nodes || !cJSON_IsArray(nodes)) {
00359         // Log error and clean up if Nodes array is invalid
00360         ESP_LOGE(TAG, "Invalid or missing Nodes array in wire");
00361         cJSON_Delete(wire);
00362         vTaskDelete(NULL);
00363         return;
00364     }
00365
00366     while (1) {
00367         bool condition = true;
00368         cJSON *last_coil = NULL;
00369
00370         // Process nodes and identify the last coil
00371         process_nodes(nodes, &condition, &last_coil);
00372
00373         // Process the coil if present
```

```
00374         if (last_coil) {
00375             process_coil(last_coil, condition);
00376         }
00377 
00378         // Delay to prevent excessive CPU usage
00379         vTaskDelay(pdMS_TO_TICKS(10));
00380     }
00381 
00382     // Clean up (unreachable due to infinite loop)
00383     cJSON_Delete(wire);
00384     vTaskDelete(NULL);
00385 }
00386 
00390 void delete_all_tasks(void) {
00391     // Stop timeout timer if it exists
00392     if (config_timeout_timer) {
00393         xTimerStop(config_timeout_timer, portMAX_DELAY);
00394         xTimerDelete(config_timeout_timer, portMAX_DELAY);
00395         config_timeout_timer = NULL;
00396     }
00397 
00398     // Free large_buffer if it exists
00399     if (large_buffer) {
00400         free(large_buffer);
00401         large_buffer = NULL;
00402         total_received = 0;
00403     }
00404 
00405     // Delete all tasks
00406     if (tasks) {
00407         for (int i = 0; i < num_tasks; i++) {
00408             if (tasks[i].handle) {
00409                 vTaskDelete(tasks[i].handle);
00410                 // Log task deletion
00411                 ESP_LOGI(TAG, "Deleted task %d", i);
00412                 tasks[i].handle = NULL;
00413             }
00414             if (tasks[i].wire_copy) {
00415                 cJSON_Delete(tasks[i].wire_copy);
00416                 // Log wire copy cleanup
00417                 ESP_LOGI(TAG, "Freed wire_copy for task %d", i);
00418                 tasks[i].wire_copy = NULL;
00419             }
00420         }
00421         free(tasks);
00422         tasks = NULL;
00423         num_tasks = 0;
00424     }
00425 }
00426 
00427 void configure(const char *data, int data_len, bool loaded_from_nvs) {
00428     // Create/restart timeout timer
00429     if (config_timeout_timer == NULL) {
00430         config_timeout_timer = xTimerCreate(
00431             "ConfigTimeout",
00432             pdMS_TO_TICKS(CONFIG_TIMEOUT_MS),
00433             pdFALSE, // One-shot timer
00434             (void*)0,
00435             config_timeout_callback
00436         );
00437         if (config_timeout_timer == NULL) {
00438             // Log error if timer creation fails
00439             ESP_LOGE(TAG, "Failed to create config timeout timer");
00440             return;
00441         }
00442     }
00443     xTimerReset(config_timeout_timer, portMAX_DELAY);
00444 
00445     // Allocate or expand buffer for new part
00446     char *new_buffer = realloc(large_buffer, total_received + data_len + 1); // +1 for
      null-termination
00447     if (!new_buffer) {
00448         // Log error and clean up if memory allocation fails
00449         ESP_LOGE(TAG, "Memory allocation failed for buffer");
00450         free(large_buffer);
00451         large_buffer = NULL;
00452         total_received = 0;
00453         xTimerStop(config_timeout_timer, portMAX_DELAY);
00454         return;
00455     }
00456     large_buffer = new_buffer;
00457 
00458     // Copy new part into buffer
00459     memcpy(large_buffer + total_received, data, data_len);
00460     total_received += data_len;
00461     large_buffer[total_received] = '\0'; // Null-termination for safe parsing
00462 
```

```
00463     // Log received data
00464     ESP_LOGI(TAG, "Received %d bytes, total: %zu", data_len, total_received);
00465
00466     // Attempt to parse JSON
00467     cJSON *json = cJSON_Parse(large_buffer);
00468     if (json) {
00469         // JSON is valid, complete message received
00470         ESP_LOGI(TAG, "Complete JSON received, length: %zu bytes", total_received);
00471
00472         // Stop timeout timer
00473         xTimerStop(config_timeout_timer, portMAX_DELAY);
00474
00475         // Save to NVS
00476         if(!loaded_from_nvs) {
00477             delete_config_from_nvs();
00478             save_config_to_nvs(large_buffer, total_received);
00479         }
00480
00481         // Delete all previous tasks
00482         delete_all_tasks();
00483
00484         // Get Device data
00485         cJSON *device = cJSON_GetObjectItem(json, "Device");
00486         device_init(device);
00487         print_device_info();
00488
00489         // Get variables
00490         cJSON *variables = cJSON_GetObjectItem(json, "Variables");
00491         load_variables(variables);
00492
00493         cJSON *wires = cJSON_GetObjectItem(json, "Wires");
00494         if (!cJSON_IsArray(wires)) {
00495             // Log error and clean up if Wires is not an array
00496             ESP_LOGE(TAG, "Wires is not an array");
00497             cJSON_Delete(json);
00498             free(large_buffer);
00499             large_buffer = NULL;
00500             total_received = 0;
00501             return;
00502         }
00503
00504         int array_size = cJSON_GetArraySize(wires);
00505         // Log number of wires found
00506         ESP_LOGI(TAG, "Found wires: %d", array_size);
00507
00508         // Allocate memory for task array
00509         if (esp_get_free_heap_size() < array_size * sizeof(TaskInfo) + (array_size * 4096) + 1024) {
00510             // Log error and clean up if insufficient heap memory
00511             ESP_LOGE(TAG, "Insufficient heap memory for %d tasks", array_size);
00512             cJSON_Delete(json);
00513             free(large_buffer);
00514             large_buffer = NULL;
00515             total_received = 0;
00516             return;
00517         }
00518
00519         tasks = calloc(array_size, sizeof(TaskInfo));
00520         if (!tasks) {
00521             // Log error and clean up if task array allocation fails
00522             ESP_LOGE(TAG, "Memory allocation failed for %d tasks", array_size);
00523             cJSON_Delete(json);
00524             free(large_buffer);
00525             large_buffer = NULL;
00526             total_received = 0;
00527             return;
00528         }
00529         num_tasks = array_size;
00530
00531         //printf("Heap before %lu\n", esp_get_free_heap_size());
00532
00533         // Iterate through wires and create tasks
00534         for (int i = 0; i < array_size; i++) {
00535             cJSON *wire = cJSON_GetArrayItem(wires, i);
00536             if (!cJSON_IsObject(wire)) {
00537                 // Log warning and skip if wire is not an object
00538                 ESP_LOGW(TAG, "Wire %d is not an object, skipping", i);
00539                 continue;
00540             }
00541
00542             // Duplicate wire JSON object
00543             cJSON *wire_copy = cJSON_Duplicate(wire, true);
00544             if (!wire_copy) {
00545                 // Log error if wire duplication fails
00546                 ESP_LOGE(TAG, "Failed to duplicate wire %d", i);
00547                 continue;
00548             }
00549
```

```
00550             // Check available stack space
00551             if (uxTaskGetStackHighWaterMark(NULL) < 1024) {
00552                 // Log warning and skip if stack space is low
00553                 ESP_LOGW(TAG, "Low stack space, skipping task %d", i);
00554                 cJSON_Delete(wire_copy);
00555                 continue;
00556             }
00557 
00558             // Create task
00559             char task_name[16];
00560             snprintf(task_name, sizeof(task_name), "Wire%d", i);
00561             tasks[i].wire_copy = wire_copy; // Store wire_copy for cleanup
00562             if (xTaskCreate(process_block_task, task_name, 4096, wire_copy, 5, &tasks[i].handle) !=
      pdPASS) {
00563                 // Log error if task creation fails
00564                 ESP_LOGE(TAG, "Failed to create task %d", i);
00565                 cJSON_Delete(wire_copy);
00566                 tasks[i].wire_copy = NULL;
00567             } else {
00568                 // Log successful task creation
00569                 ESP_LOGI(TAG, "Created task %d for wire %d", i, i);
00570             }
00571 
00572             // Delay between task creations to avoid overloading
00573             vTaskDelay(pdMS_TO_TICKS(200));
00574         }
00575         //printf("Heap after %lu\n", esp_get_free_heap_size());
00576 
00577         // Free memory and reset state
00578         cJSON_Delete(json);
00579         free(large_buffer);
00580         large_buffer = NULL;
00581         total_received = 0;
00582     } else {
00583         // Log info if JSON is incomplete
00584         ESP_LOGI(TAG, "JSON incomplete, waiting for next part...");
00585         cJSON_Delete(json);
00586     }
00587 }
```

4.13 main/conf_task_manager.h File Reference

```
#include <stdbool.h>
```

Functions

- void configure (const char *data, int data_len, bool loaded_from_nvs)

  Configures tasks based on provided data.
- void delete_all_tasks (void)

  Deletes all configured tasks.

4.13.1 Function Documentation

4.13.1.1 configure()

```
void configure (
            const char * data,
            int data_len,
            bool loaded_from_nvs)
```

Configures tasks based on provided data.

Parameters

data	Pointer to the configuration data.
data_len	Length of the configuration data.
loaded_from_nvs	Indicates if the data was loaded from non-volatile storage.

Definition at line 427 of file conf_task_manager.c.

4.13.1.2 delete_all_tasks()

```
void delete_all_tasks (
            void )
```

Deletes all configured tasks.

Deletes all configured tasks.

Definition at line 390 of file conf_task_manager.c.

4.14 conf_task_manager.h

Go to the documentation of this file.

```
00001 #ifndef TASK_MANAGER_H
00002 #define TASK_MANAGER_H
00003
00004 #include <stdbool.h>
00005
00012 void configure(const char *data, int data_len, bool loaded_from_nvs);
00013
00017 void delete_all_tasks(void);
00018
00019 #endif // TASK_MANAGER_H
```

4.15 main/device_config.c File Reference

```
#include "device_config.h"
#include "driver/gpio.h"
#include "esp_log.h"
#include "cJSON.h"
#include <string.h>
#include <stdlib.h>
#include "sensor.h"
```

Functions

- void free_device (Device ∗dev)

  Frees the memory allocated for the Device structure.
- void print_device_info (void)

  Prints information about the device configuration.
- void load_device_configuration (cJSON ∗device)

  Loads the device configuration from a JSON object.
- bool find_pin_by_name (const char ∗pin_name, gpio_num_t ∗pin)

  Finds a GPIO pin number based on its name.
- void init_digital_inputs (void)

  Initializes digital input pins.
- void init_digital_outputs (void)

  Initializes digital output pins.
- void init_analog_inputs (void)

  Initializes analog input pins (not implemented).
- void init_analog_outputs (void)

  Initializes analog output pins (not implemented).
- void init_one_wire_inputs (void)

  Initializes one-wire input pins.
- void device_init (cJSON ∗device)

  Initializes the device configuration from a JSON object.
- bool get_digital_input_value (const char ∗pin_name)

  Gets the value of a digital input pin by its name.
- esp_err_t set_digital_output_value (const char ∗pin_name, int value)

  Sets the value of a digital output pin by its name.
- bool get_digital_output_value (const char ∗pin_name)

  Gets the value of a digital output pin by its name.
- int get_analog_input_value (const char ∗pin_name)

  Gets the value of an analog input pin by its name.
- esp_err_t set_analog_output_value (const char ∗pin_name, uint8_t value)

  Sets the value of an analog output (DAC) pin by its name.
- int get_analog_output_value (const char ∗pin_name)

  Gets the value of an analog output (DAC) pin by its name.
- float get_one_wire_value (const char ∗pin_name)

  Gets the value from a one-wire device by its pin name.

Variables

- static const char ∗ TAG = "DEVICE CONFIGURATION"

  Tag for logging messages from the device configuration module.
- Device _device = {0}

  Global device configuration structure, initialized to zero.

4.15.1 Function Documentation

4.15.1.1 device_init()

```
void device_init (
            cJSON * device)
```

Initializes the device configuration from a JSON object.

Parameters

device	JSON object containing the device configuration.

Definition at line 826 of file device_config.c.

4.15.1.2 find_pin_by_name()

```
bool find_pin_by_name (
            const char * pin_name,
            gpio_num_t * pin)
```

Finds a GPIO pin number based on its name.

Parameters

pin_name	Name of the pin to find.
pin	Pointer to store the found GPIO pin number.

Returns

bool True if the pin is found, false otherwise.

Definition at line 640 of file device_config.c.

4.15.1.3 free_device()

```
void free_device (
            Device * dev)
```

Frees the memory allocated for the Device structure.

Parameters

dev	Pointer to the Device structure to free.

Definition at line 24 of file device_config.c.

4.15.1.4 get_analog_input_value()

```
int get_analog_input_value (
            const char * pin_name)
```

Gets the value of an analog input pin by its name.

Parameters

pin_name	Name of the analog input pin.

Returns

int The analog input value.

Definition at line 887 of file device_config.c.

4.15.1.5 get_analog_output_value()

```
int get_analog_output_value (
            const char * pin_name)
```

Gets the value of an analog output (DAC) pin by its name.

Parameters

pin_name	Name of the DAC output pin.

Returns

int The current DAC output value.

Definition at line 895 of file device_config.c.

4.15.1.6 get_digital_input_value()

```
bool get_digital_input_value (
            const char * pin_name)
```

Gets the value of a digital input pin by its name.

Parameters

pin_name	Name of the digital input pin.

Returns

bool The value of the digital input (true for high, false for low).

Definition at line 838 of file device_config.c.

4.15.1.7 get_digital_output_value()

```
bool get_digital_output_value (
            const char * pin_name)
```

Gets the value of a digital output pin by its name.

Parameters

pin_name	Name of the digital output pin.

Returns

bool The value of the digital output (true for high, false for low).

Definition at line 872 of file device_config.c.

4.15.1.8 get_one_wire_value()

```
float get_one_wire_value (
            const char * pin_name)
```

Gets the value from a one-wire device by its pin name.

Parameters

pin_name	Name of the one-wire input pin.

Returns

float The value read from the one-wire device.

Definition at line 900 of file device_config.c.

4.15.1.9 init_analog_inputs()

```
void init_analog_inputs (
            void )
```

Initializes analog input pins (not implemented).

Definition at line 769 of file device_config.c.

4.15.1.10 init_analog_outputs()

```
void init_analog_outputs (
            void )
```

Initializes analog output pins (not implemented).

Definition at line 776 of file device_config.c.

4.15.1.11 init_digital_inputs()

```
void init_digital_inputs (
            void )
```

Initializes digital input pins.

Definition at line 694 of file device_config.c.

4.15.1.12 init_digital_outputs()

```
void init_digital_outputs (
            void )
```

Initializes digital output pins.

Definition at line 731 of file device_config.c.

4.15.1.13 init_one_wire_inputs()

```
void init_one_wire_inputs (
            void )
```

Initializes one-wire input pins.

Definition at line 784 of file device_config.c.

4.15.1.14 load_device_configuration()

```
void load_device_configuration (
            cJSON * device)
```

Loads the device configuration from a JSON object.

Parameters

device	JSON object containing the device configuration.

Definition at line 200 of file device_config.c.

4.15.1.15 print_device_info()

```
void print_device_info (
            void )
```

Prints information about the device configuration.

Definition at line 113 of file device_config.c.

4.15.1.16 set_analog_output_value()

```
esp_err_t set_analog_output_value (
            const char * pin_name,
            uint8_t value)
```

Sets the value of an analog output (DAC) pin by its name.

Parameters

pin_name	Name of the DAC output pin.
value	Value to set (typically 0-255 for 8-bit DAC).

Returns

esp_err_t ESP_OK on success, or an error code on failure.

Definition at line 891 of file device_config.c.

4.15.1.17 set_digital_output_value()

```
esp_err_t set_digital_output_value (
            const char * pin_name,
            int value)
```

Sets the value of a digital output pin by its name.

Parameters

pin_name	Name of the digital output pin.
value	Value to set (0 for low, non-zero for high).

Returns

esp_err_t ESP_OK on success, or an error code on failure.

Definition at line 852 of file device_config.c.

4.15.2 Variable Documentation

4.15.2.1 _device

```
Device _device = {0}
```

Global device configuration structure, initialized to zero.

Global variable holding the device configuration.

Definition at line 18 of file device_config.c.

4.15.2.2 TAG

```
const char* TAG = "DEVICE CONFIGURATION"  [static]
```

Tag for logging messages from the device configuration module.

Definition at line 13 of file device_config.c.

4.16 device_config.c

Go to the documentation of this file.

```
00001 #include "device_config.h"
00002 #include "driver/gpio.h"
00003 #include "esp_log.h"
00004 #include "cJSON.h"
00005 #include <string.h>
00006 #include <stdlib.h>
00007
00008 #include "sensor.h"
00009
00013 static const char *TAG = "DEVICE CONFIGURATION";
00014
00018 Device _device = {0};
00019
00024 void free_device(Device *dev) {
00025     free(dev->device_name);
00026
00027     // Free Digital I/O
00028     free(dev->digital_inputs);
00029     if (dev->digital_inputs_names) {
00030         for (size_t i = 0; i < dev->digital_inputs_names_len; i++) {
00031             free(dev->digital_inputs_names[i]);
00032         }
00033         free(dev->digital_inputs_names);
00034     }
00035     free(dev->digital_outputs);
00036     if (dev->digital_outputs_names) {
00037         for (size_t i = 0; i < dev->digital_outputs_names_len; i++) {
00038             free(dev->digital_outputs_names[i]);
00039         }
00040         free(dev->digital_outputs_names);
00041     }
00042
00043     // Free Analog I/O
00044     free(dev->analog_inputs);
00045     if (dev->analog_inputs_names) {
00046         for (size_t i = 0; i < dev->analog_inputs_names_len; i++) {
00047             free(dev->analog_inputs_names[i]);
00048         }
00049         free(dev->analog_inputs_names);
00050     }
00051     free(dev->dac_outputs);
00052     if (dev->dac_outputs_names) {
00053         for (size_t i = 0; i < dev->dac_outputs_names_len; i++) {
00054             free(dev->dac_outputs_names[i]);
00055         }
```

```
00056         free(dev->dac_outputs_names);
00057     }
00058
00059     // Free One-Wire
00060     free(dev->one_wire_inputs);
00061     if (dev->one_wire_inputs_names) {
00062         for (size_t i = 0; i < dev->one_wire_inputs_len; i++) {
00063             if (dev->one_wire_inputs_names[i]) {
00064                 for (size_t j = 0; j < dev->one_wire_inputs_names_len[i]; j++) {
00065                     free(dev->one_wire_inputs_names[i][j]);
00066                 }
00067                 free(dev->one_wire_inputs_names[i]);
00068             }
00069         }
00070         free(dev->one_wire_inputs_names);
00071     }
00072     if (dev->one_wire_inputs_devices_types) {
00073         for (size_t i = 0; i < dev->one_wire_inputs_len; i++) {
00074             if (dev->one_wire_inputs_devices_types[i]) {
00075                 for (size_t j = 0; j < dev->one_wire_inputs_devices_types_len[i]; j++) {
00076                     free(dev->one_wire_inputs_devices_types[i][j]);
00077                 }
00078                 free(dev->one_wire_inputs_devices_types[i]);
00079             }
00080         }
00081         free(dev->one_wire_inputs_devices_types);
00082     }
00083     if (dev->one_wire_inputs_devices_addresses) {
00084         for (size_t i = 0; i < dev->one_wire_inputs_len; i++) {
00085             if (dev->one_wire_inputs_devices_addresses[i]) {
00086                 for (size_t j = 0; j < dev->one_wire_inputs_devices_addresses_len[i]; j++) {
00087                     free(dev->one_wire_inputs_devices_addresses[i][j]);
00088                 }
00089                 free(dev->one_wire_inputs_devices_addresses[i]);
00090             }
00091         }
00092         free(dev->one_wire_inputs_devices_addresses);
00093     }
00094     free(dev->one_wire_inputs_names_len);
00095     free(dev->one_wire_inputs_devices_types_len);
00096     free(dev->one_wire_inputs_devices_addresses_len);
00097
00098     free(dev->uart);
00099     free(dev->i2c);
00100     free(dev->spi);
00101
00102     if (dev->parent_devices) {
00103         for (size_t i = 0; i < dev->parent_devices_len; i++) {
00104             free(dev->parent_devices[i]);
00105         }
00106         free(dev->parent_devices);
00107     }
00108
00109     // Reset the structure to zeros
00110     memset(dev, 0, sizeof(Device));
00111 }
00112
00113 void print_device_info(void){
00114     ESP_LOGI(TAG, "Device Info:");
00115     ESP_LOGI(TAG, "  device_name: %s", _device.device_name ? _device.device_name : "(null)");
00116     ESP_LOGI(TAG, "  logic_voltage: %.1f", _device.logic_voltage);
00117     ESP_LOGI(TAG, "  digital_inputs: [%zu elements]", _device.digital_inputs_len);
00118     for (size_t i = 0; i < _device.digital_inputs_len; i++) {
00119         ESP_LOGI(TAG, "    - %d", _device.digital_inputs[i]);
00120     }
00121     ESP_LOGI(TAG, "  digital_inputs_names: [%zu elements]", _device.digital_inputs_names_len);
00122     for (size_t i = 0; i < _device.digital_inputs_names_len; i++) {
00123         ESP_LOGI(TAG, "    - %s", _device.digital_inputs_names[i] ? _device.digital_inputs_names[i] :
      "(null)");
00124     }
00125     ESP_LOGI(TAG, "  digital_outputs: [%zu elements]", _device.digital_outputs_len);
00126     for (size_t i = 0; i < _device.digital_outputs_len; i++) {
00127         ESP_LOGI(TAG, "    - %d", _device.digital_outputs[i]);
00128     }
00129     ESP_LOGI(TAG, "  digital_outputs_names: [%zu elements]", _device.digital_outputs_names_len);
00130     for (size_t i = 0; i < _device.digital_outputs_names_len; i++) {
00131         ESP_LOGI(TAG, "    - %s", _device.digital_outputs_names[i] ? _device.digital_outputs_names[i]
      : "(null)");
00132     }
00133     ESP_LOGI(TAG, "  analog_inputs: [%zu elements]", _device.analog_inputs_len);
00134     for (size_t i = 0; i < _device.analog_inputs_len; i++) {
00135         ESP_LOGI(TAG, "    - %d", _device.analog_inputs[i]);
00136     }
00137     ESP_LOGI(TAG, "  analog_inputs_names: [%zu elements]", _device.analog_inputs_names_len);
00138     for (size_t i = 0; i < _device.analog_inputs_names_len; i++) {
00139         ESP_LOGI(TAG, "    - %s", _device.analog_inputs_names[i] ? _device.analog_inputs_names[i] :
      "(null)");
```

```
00140     }
00141     ESP_LOGI(TAG, "  dac_outputs: [%zu elements]", _device.dac_outputs_len);
00142     for (size_t i = 0; i < _device.dac_outputs_len; i++) {
00143         ESP_LOGI(TAG, "    - %d", _device.dac_outputs[i]);
00144     }
00145     ESP_LOGI(TAG, "  dac_outputs_names: [%zu elements]", _device.dac_outputs_names_len);
00146     for (size_t i = 0; i < _device.dac_outputs_names_len; i++) {
00147         ESP_LOGI(TAG, "    - %s", _device.dac_outputs_names[i] ? _device.dac_outputs_names[i] :
    "(null)");
00148     }
00149     ESP_LOGI(TAG, "  one_wire_inputs: [%zu elements]", _device.one_wire_inputs_len);
00150     for (size_t i = 0; i < _device.one_wire_inputs_len; i++) {
00151         ESP_LOGI(TAG, "    - %d", _device.one_wire_inputs[i]);
00152     }
00153     ESP_LOGI(TAG, "  one_wire_inputs_names: [%zu elements]", _device.one_wire_inputs_len);
00154     for (size_t i = 0; i < _device.one_wire_inputs_len; i++) {
00155         ESP_LOGI(TAG, "    Pin %zu: [%zu elements]", i, _device.one_wire_inputs_names_len[i]);
00156         for (size_t j = 0; j < _device.one_wire_inputs_names_len[i]; j++) {
00157             ESP_LOGI(TAG, "      - %s", _device.one_wire_inputs_names[i][j] ?
    _device.one_wire_inputs_names[i][j] : "(null)");
00158         }
00159     }
00160     ESP_LOGI(TAG, "  one_wire_inputs_devices_types: [%zu elements]", _device.one_wire_inputs_len);
00161     for (size_t i = 0; i < _device.one_wire_inputs_len; i++) {
00162         ESP_LOGI(TAG, "    Pin %zu: [%zu elements]", i, _device.one_wire_inputs_devices_types_len[i]);
00163         for (size_t j = 0; j < _device.one_wire_inputs_devices_types_len[i]; j++) {
00164             ESP_LOGI(TAG, "      - %s", _device.one_wire_inputs_devices_types[i][j] ?
    _device.one_wire_inputs_devices_types[i][j] : "(null)");
00165         }
00166     }
00167     ESP_LOGI(TAG, "  one_wire_inputs_devices_addresses: [%zu elements]", _device.one_wire_inputs_len);
00168     for (size_t i = 0; i < _device.one_wire_inputs_len; i++) {
00169         ESP_LOGI(TAG, "    Pin %zu: [%zu elements]", i,
    _device.one_wire_inputs_devices_addresses_len[i]);
00170         for (size_t j = 0; j < _device.one_wire_inputs_devices_addresses_len[i]; j++) {
00171             ESP_LOGI(TAG, "      - %s", _device.one_wire_inputs_devices_addresses[i][j] ?
    _device.one_wire_inputs_devices_addresses[i][j] : "(null)");
00172         }
00173     }
00174     ESP_LOGI(TAG, "  pwm_channels: %d", _device.pwm_channels);
00175     ESP_LOGI(TAG, "  max_hardware_timers: %d", _device.max_hardware_timers);
00176     ESP_LOGI(TAG, "  has_rtos: %s", _device.has_rtos ? "true" : "false");
00177     ESP_LOGI(TAG, "  uart: [%zu elements]", _device.uart_len);
00178     for (size_t i = 0; i < _device.uart_len; i++) {
00179         ESP_LOGI(TAG, "    - %d", _device.uart[i]);
00180     }
00181     ESP_LOGI(TAG, "  i2c: [%zu elements]", _device.i2c_len);
00182     for (size_t i = 0; i < _device.i2c_len; i++) {
00183         ESP_LOGI(TAG, "    - %d", _device.i2c[i]);
00184     }
00185     ESP_LOGI(TAG, "  spi: [%zu elements]", _device.spi_len);
00186     for (size_t i = 0; i < _device.spi_len; i++) {
00187         ESP_LOGI(TAG, "    - %d", _device.spi[i]);
00188     }
00189     ESP_LOGI(TAG, "  usb: %s", _device.usb ? "true" : "false");
00190     ESP_LOGI(TAG, "  parent_devices: [%zu elements]", _device.parent_devices_len);
00191     for (size_t i = 0; i < _device.parent_devices_len; i++) {
00192         ESP_LOGI(TAG, "    - %s", _device.parent_devices[i]);
00193     }
00194 }
00195
00200 void load_device_configuration(cJSON *device)
00201 {
00202     // Free existing device memory before loading new configuration
00203     free_device(&_device);
00204
00205     // device_name
00206     cJSON *device_name = cJSON_GetObjectItem(device, "device_name");
00207     if (device_name && cJSON_IsString(device_name) && device_name->valuestring) {
00208         _device.device_name = strdup(device_name->valuestring);
00209         if (!_device.device_name) {
00210             // Log memory allocation error for device_name
00211             ESP_LOGE(TAG, "Error allocating memory for device_name");
00212         }
00213     }
00214
00215     // logic_voltage
00216     cJSON *logic_voltage = cJSON_GetObjectItem(device, "logic_voltage");
00217     if (logic_voltage && cJSON_IsNumber(logic_voltage)) {
00218         _device.logic_voltage = logic_voltage->valuedouble;
00219     }
00220
00221     // digital_inputs
00222     cJSON *digital_inputs = cJSON_GetObjectItem(device, "digital_inputs");
00223     if (digital_inputs && cJSON_IsArray(digital_inputs)) {
00224         _device.digital_inputs_len = cJSON_GetArraySize(digital_inputs);
00225         _device.digital_inputs = malloc(_device.digital_inputs_len * sizeof(int));
```

```
00226         if (_device.digital_inputs) {
00227             for (size_t i = 0; i < _device.digital_inputs_len; i++) {
00228                 cJSON *item = cJSON_GetArrayItem(digital_inputs, i);
00229                 if (item && cJSON_IsNumber(item)) {
00230                     _device.digital_inputs[i] = item->valueint;
00231                 }
00232             }
00233         } else {
00234             if(_device.digital_inputs_len != 0)
00235                 // Log memory allocation error for digital_inputs
00236                 ESP_LOGE(TAG, "Error allocating memory for digital_inputs");
00237         }
00238     }
00239
00240     // digital_inputs_names
00241     cJSON *digital_inputs_names = cJSON_GetObjectItem(device, "digital_inputs_names");
00242     if (digital_inputs_names && cJSON_IsArray(digital_inputs_names)) {
00243         _device.digital_inputs_names_len = cJSON_GetArraySize(digital_inputs_names);
00244         _device.digital_inputs_names = malloc(_device.digital_inputs_names_len * sizeof(char *));
00245         if (_device.digital_inputs_names) {
00246             for (size_t i = 0; i < _device.digital_inputs_names_len; i++) {
00247                 cJSON *item = cJSON_GetArrayItem(digital_inputs_names, i);
00248                 if (item && cJSON_IsString(item) && item->valuestring) {
00249                     _device.digital_inputs_names[i] = strdup(item->valuestring);
00250                     if (!_device.digital_inputs_names[i]) {
00251                         // Log memory allocation error for digital_inputs_names
00252                         ESP_LOGE(TAG, "Error allocating memory for digital_inputs_names[%zu]", i);
00253                     }
00254                 } else {
00255                     _device.digital_inputs_names[i] = NULL;
00256                 }
00257             }
00258         } else {
00259             if(_device.digital_inputs_names_len != 0)
00260                 // Log memory allocation error for digital_inputs_names array
00261                 ESP_LOGE(TAG, "Error allocating memory for digital_inputs_names array");
00262         }
00263     }
00264
00265     // digital_outputs
00266     cJSON *digital_outputs = cJSON_GetObjectItem(device, "digital_outputs");
00267     if (digital_outputs && cJSON_IsArray(digital_outputs)) {
00268         _device.digital_outputs_len = cJSON_GetArraySize(digital_outputs);
00269         _device.digital_outputs = malloc(_device.digital_outputs_len * sizeof(int));
00270         if (_device.digital_outputs) {
00271             for (size_t i = 0; i < _device.digital_outputs_len; i++) {
00272                 cJSON *item = cJSON_GetArrayItem(digital_outputs, i);
00273                 if (item && cJSON_IsNumber(item)) {
00274                     _device.digital_outputs[i] = item->valueint;
00275                 }
00276             }
00277         } else {
00278             if(_device.digital_outputs_len != 0)
00279                 // Log memory allocation error for digital_outputs
00280                 ESP_LOGE(TAG, "Error allocating memory for digital_outputs");
00281         }
00282     }
00283
00284     // digital_outputs_names
00285     cJSON *digital_outputs_names = cJSON_GetObjectItem(device, "digital_outputs_names");
00286     if (digital_outputs_names && cJSON_IsArray(digital_outputs_names)) {
00287         _device.digital_outputs_names_len = cJSON_GetArraySize(digital_outputs_names);
00288         _device.digital_outputs_names = malloc(_device.digital_outputs_names_len * sizeof(char *));
00289         if (_device.digital_outputs_names) {
00290             for (size_t i = 0; i < _device.digital_outputs_names_len; i++) {
00291                 cJSON *item = cJSON_GetArrayItem(digital_outputs_names, i);
00292                 if (item && cJSON_IsString(item) && item->valuestring) {
00293                     _device.digital_outputs_names[i] = strdup(item->valuestring);
00294                     if (!_device.digital_outputs_names[i]) {
00295                         // Log memory allocation error for digital_outputs_names
00296                         ESP_LOGE(TAG, "Error allocating memory for digital_outputs_names[%zu]", i);
00297                     }
00298                 } else {
00299                     _device.digital_outputs_names[i] = NULL;
00300                 }
00301             }
00302         } else {
00303             if(_device.digital_outputs_names_len != 0)
00304                 // Log memory allocation error for digital_outputs_names array
00305                 ESP_LOGE(TAG, "Error allocating memory for digital_outputs_names array");
00306         }
00307     }
00308
00309     // analog_inputs
00310     cJSON *analog_inputs = cJSON_GetObjectItem(device, "analog_inputs");
00311     if (analog_inputs && cJSON_IsArray(analog_inputs)) {
00312         _device.analog_inputs_len = cJSON_GetArraySize(analog_inputs);
```

```
00313         _device.analog_inputs = malloc(_device.analog_inputs_len * sizeof(int));
00314         if (_device.analog_inputs) {
00315             for (size_t i = 0; i < _device.analog_inputs_len; i++) {
00316                 cJSON *item = cJSON_GetArrayItem(analog_inputs, i);
00317                 if (item && cJSON_IsNumber(item)) {
00318                     _device.analog_inputs[i] = item->valueint;
00319                 }
00320             }
00321         } else {
00322             if(_device.analog_inputs_len != 0)
00323                 // Log memory allocation error for analog_inputs
00324                 ESP_LOGE(TAG, "Error allocating memory for analog_inputs");
00325         }
00326     }
00327
00328     // analog_inputs_names
00329     cJSON *analog_inputs_names = cJSON_GetObjectItem(device, "analog_inputs_names");
00330     if (analog_inputs_names && cJSON_IsArray(analog_inputs_names)) {
00331         _device.analog_inputs_names_len = cJSON_GetArraySize(analog_inputs_names);
00332         _device.analog_inputs_names = malloc(_device.analog_inputs_names_len * sizeof(char *));
00333         if (_device.analog_inputs_names) {
00334             for (size_t i = 0; i < _device.analog_inputs_names_len; i++) {
00335                 cJSON *item = cJSON_GetArrayItem(analog_inputs_names, i);
00336                 if (item && cJSON_IsString(item) && item->valuestring) {
00337                     _device.analog_inputs_names[i] = strdup(item->valuestring);
00338                     if (!_device.analog_inputs_names[i]) {
00339                         // Log memory allocation error for analog_inputs_names
00340                         ESP_LOGE(TAG, "Error allocating memory for analog_inputs_names[%zu]", i);
00341                     }
00342                 } else {
00343                     _device.analog_inputs_names[i] = NULL;
00344                 }
00345             }
00346         } else {
00347             if(_device.analog_inputs_names_len != 0)
00348                 // Log memory allocation error for analog_inputs_names array
00349                 ESP_LOGE(TAG, "Error allocating memory for analog_inputs_names array");
00350         }
00351     }
00352
00353     // dac_outputs
00354     cJSON *dac_outputs = cJSON_GetObjectItem(device, "dac_outputs");
00355     if (dac_outputs && cJSON_IsArray(dac_outputs)) {
00356         _device.dac_outputs_len = cJSON_GetArraySize(dac_outputs);
00357         _device.dac_outputs = malloc(_device.dac_outputs_len * sizeof(int));
00358         if (_device.dac_outputs) {
00359             for (size_t i = 0; i < _device.dac_outputs_len; i++) {
00360                 cJSON *item = cJSON_GetArrayItem(dac_outputs, i);
00361                 if (item && cJSON_IsNumber(item)) {
00362                     _device.dac_outputs[i] = item->valueint;
00363                 }
00364             }
00365         } else {
00366             if(_device.dac_outputs_len != 0)
00367                 // Log memory allocation error for dac_outputs
00368                 ESP_LOGE(TAG, "Error allocating memory for dac_outputs");
00369         }
00370     }
00371
00372     // dac_outputs_names
00373     cJSON *dac_outputs_names = cJSON_GetObjectItem(device, "dac_outputs_names");
00374     if (dac_outputs_names && cJSON_IsArray(dac_outputs_names)) {
00375         _device.dac_outputs_names_len = cJSON_GetArraySize(dac_outputs_names);
00376         _device.dac_outputs_names = malloc(_device.dac_outputs_names_len * sizeof(char *));
00377         if (_device.dac_outputs_names) {
00378             for (size_t i = 0; i < _device.dac_outputs_names_len; i++) {
00379                 cJSON *item = cJSON_GetArrayItem(dac_outputs_names, i);
00380                 if (item && cJSON_IsString(item) && item->valuestring) {
00381                     _device.dac_outputs_names[i] = strdup(item->valuestring);
00382                     if (!_device.dac_outputs_names[i]) {
00383                         // Log memory allocation error for dac_outputs_names
00384                         ESP_LOGE(TAG, "Error allocating memory for dac_outputs_names[%zu]", i);
00385                     }
00386                 } else {
00387                     _device.dac_outputs_names[i] = NULL;
00388                 }
00389             }
00390         } else {
00391             if(_device.dac_outputs_names_len != 0)
00392                 // Log memory allocation error for dac_outputs_names array
00393                 ESP_LOGE(TAG, "Error allocating memory for dac_outputs_names array");
00394         }
00395     }
00396
00397     // one_wire_inputs
00398     cJSON *one_wire_inputs = cJSON_GetObjectItem(device, "one_wire_inputs");
00399     if (one_wire_inputs && cJSON_IsArray(one_wire_inputs)) {
```

```
00400         _device.one_wire_inputs_len = cJSON_GetArraySize(one_wire_inputs);
00401         _device.one_wire_inputs = malloc(_device.one_wire_inputs_len * sizeof(int));
00402         if (_device.one_wire_inputs) {
00403             for (size_t i = 0; i < _device.one_wire_inputs_len; i++) {
00404                 cJSON *item = cJSON_GetArrayItem(one_wire_inputs, i);
00405                 if (item && cJSON_IsNumber(item)) {
00406                     _device.one_wire_inputs[i] = item->valueint;
00407                 }
00408             }
00409         } else {
00410             if(_device.one_wire_inputs_len != 0)
00411                 // Log memory allocation error for one_wire_inputs
00412                 ESP_LOGE(TAG, "Error allocating memory for one_wire_inputs");
00413         }
00414     }
00415
00416     // one_wire_inputs_names
00417     cJSON *one_wire_inputs_names = cJSON_GetObjectItem(device, "one_wire_inputs_names");
00418     if (one_wire_inputs_names && cJSON_IsArray(one_wire_inputs_names)) {
00419         _device.one_wire_inputs_names = malloc(_device.one_wire_inputs_len * sizeof(char **));
00420         _device.one_wire_inputs_names_len = malloc(_device.one_wire_inputs_len * sizeof(size_t));
00421         if (_device.one_wire_inputs_names && _device.one_wire_inputs_names_len) {
00422             for (size_t i = 0; i < _device.one_wire_inputs_len; i++) {
00423                 cJSON *sub_array = cJSON_GetArrayItem(one_wire_inputs_names, i);
00424                 if (sub_array && cJSON_IsArray(sub_array)) {
00425                     _device.one_wire_inputs_names_len[i] = cJSON_GetArraySize(sub_array);
00426                     _device.one_wire_inputs_names[i] = malloc(_device.one_wire_inputs_names_len[i] *
    sizeof(char *));
00427                     if (_device.one_wire_inputs_names[i]) {
00428                         for (size_t j = 0; j < _device.one_wire_inputs_names_len[i]; j++) {
00429                             cJSON *item = cJSON_GetArrayItem(sub_array, j);
00430                             if (item && cJSON_IsString(item) && item->valuestring) {
00431                                 _device.one_wire_inputs_names[i][j] = strdup(item->valuestring);
00432                                 if (!_device.one_wire_inputs_names[i][j]) {
00433                                     // Log memory allocation error for one_wire_inputs_names
00434                                     ESP_LOGE(TAG, "Error allocating memory for
    one_wire_inputs_names[%zu][%zu]", i, j);
00435                                 }
00436                             } else {
00437                                 _device.one_wire_inputs_names[i][j] = NULL;
00438                             }
00439                         }
00440                     } else {
00441                         if (_device.one_wire_inputs_names_len[i] != 0)
00442                             // Log memory allocation error for one_wire_inputs_names subarray
00443                             ESP_LOGE(TAG, "Error allocating memory for one_wire_inputs_names[%zu]",
    i);
00444                     }
00445                 } else {
00446                     _device.one_wire_inputs_names_len[i] = 0;
00447                     _device.one_wire_inputs_names[i] = NULL;
00448                 }
00449             }
00450         } else {
00451             // Log memory allocation error for one_wire_inputs_names or its length array
00452             ESP_LOGE(TAG, "Error allocating memory for one_wire_inputs_names or
    one_wire_inputs_names_len");
00453         }
00454     }
00455
00456     // one_wire_inputs_devices_types
00457     cJSON *one_wire_inputs_devices_types = cJSON_GetObjectItem(device,
    "one_wire_inputs_devices_types");
00458     if (one_wire_inputs_devices_types && cJSON_IsArray(one_wire_inputs_devices_types)) {
00459         _device.one_wire_inputs_devices_types = malloc(_device.one_wire_inputs_len * sizeof(char **));
00460         _device.one_wire_inputs_devices_types_len = malloc(_device.one_wire_inputs_len *
    sizeof(size_t));
00461         if (_device.one_wire_inputs_devices_types && _device.one_wire_inputs_devices_types_len) {
00462             for (size_t i = 0; i < _device.one_wire_inputs_len; i++) {
00463                 cJSON *sub_array = cJSON_GetArrayItem(one_wire_inputs_devices_types, i);
00464                 if (sub_array && cJSON_IsArray(sub_array)) {
00465                     _device.one_wire_inputs_devices_types_len[i] = cJSON_GetArraySize(sub_array);
00466                     _device.one_wire_inputs_devices_types[i] =
    malloc(_device.one_wire_inputs_devices_types_len[i] * sizeof(char *));
00467                     if (_device.one_wire_inputs_devices_types[i]) {
00468                         for (size_t j = 0; j < _device.one_wire_inputs_devices_types_len[i]; j++) {
00469                             cJSON *item = cJSON_GetArrayItem(sub_array, j);
00470                             if (item && cJSON_IsString(item) && item->valuestring) {
00471                                 _device.one_wire_inputs_devices_types[i][j] =
    strdup(item->valuestring);
00472                                 if (!_device.one_wire_inputs_devices_types[i][j]) {
00473                                     // Log memory allocation error for one_wire_inputs_devices_types
00474                                     ESP_LOGE(TAG, "Error allocating memory for
    one_wire_inputs_devices_types[%zu][%zu]", i, j);
00475                                 }
00476                             } else {
00477                                 _device.one_wire_inputs_devices_types[i][j] = NULL;
```

```
00478                             }
00479                         }
00480                     } else {
00481                         if (_device.one_wire_inputs_devices_types_len[i] != 0)
00482                             // Log memory allocation error for one_wire_inputs_devices_types subarray
00483                             ESP_LOGE(TAG, "Error allocating memory for
    one_wire_inputs_devices_types[%zu]", i);
00484                     }
00485                 } else {
00486                     _device.one_wire_inputs_devices_types_len[i] = 0;
00487                     _device.one_wire_inputs_devices_types[i] = NULL;
00488                 }
00489             }
00490         } else {
00491             // Log memory allocation error for one_wire_inputs_devices_types or its length array
00492             ESP_LOGE(TAG, "Error allocating memory for one_wire_inputs_devices_types or
    one_wire_inputs_devices_types_len");
00493         }
00494     }
00495
00496     // one_wire_inputs_devices_addresses
00497     cJSON *one_wire_inputs_devices_addresses = cJSON_GetObjectItem(device,
    "one_wire_inputs_devices_addresses");
00498     if (one_wire_inputs_devices_addresses && cJSON_IsArray(one_wire_inputs_devices_addresses)) {
00499         _device.one_wire_inputs_devices_addresses = malloc(_device.one_wire_inputs_len * sizeof(char
    **));
00500         _device.one_wire_inputs_devices_addresses_len = malloc(_device.one_wire_inputs_len *
    sizeof(size_t));
00501         if (_device.one_wire_inputs_devices_addresses &&
    _device.one_wire_inputs_devices_addresses_len) {
00502             for (size_t i = 0; i < _device.one_wire_inputs_len; i++) {
00503                 cJSON *sub_array = cJSON_GetArrayItem(one_wire_inputs_devices_addresses, i);
00504                 if (sub_array && cJSON_IsArray(sub_array)) {
00505                     _device.one_wire_inputs_devices_addresses_len[i] = cJSON_GetArraySize(sub_array);
00506                     _device.one_wire_inputs_devices_addresses[i] =
    malloc(_device.one_wire_inputs_devices_addresses_len[i] * sizeof(char *));
00507                     if (_device.one_wire_inputs_devices_addresses[i]) {
00508                         for (size_t j = 0; j < _device.one_wire_inputs_devices_addresses_len[i]; j++)
    {
00509                             cJSON *item = cJSON_GetArrayItem(sub_array, j);
00510                             if (item && cJSON_IsString(item) && item->valuestring) {
00511                                 _device.one_wire_inputs_devices_addresses[i][j] =
    strdup(item->valuestring);
00512                                 if (!_device.one_wire_inputs_devices_addresses[i][j]) {
00513                                     // Log memory allocation error for
    one_wire_inputs_devices_addresses
00514                                     ESP_LOGE(TAG, "Error allocating memory for
    one_wire_inputs_devices_addresses[%zu][%zu]", i, j);
00515                                 }
00516                             } else {
00517                                 _device.one_wire_inputs_devices_addresses[i][j] = NULL;
00518                             }
00519                         }
00520                     } else {
00521                         if (_device.one_wire_inputs_devices_addresses_len[i] != 0)
00522                             // Log memory allocation error for one_wire_inputs_devices_addresses
    subarray
00523                             ESP_LOGE(TAG, "Error allocating memory for
    one_wire_inputs_devices_addresses[%zu]", i);
00524                     }
00525                 } else {
00526                     _device.one_wire_inputs_devices_addresses_len[i] = 0;
00527                     _device.one_wire_inputs_devices_addresses[i] = NULL;
00528                 }
00529             }
00530         } else {
00531             // Log memory allocation error for one_wire_inputs_devices_addresses or its length array
00532             ESP_LOGE(TAG, "Error allocating memory for one_wire_inputs_devices_addresses or
    one_wire_inputs_devices_addresses_len");
00533         }
00534     }
00535
00536     // pwm_channels
00537     cJSON *pwm_channels = cJSON_GetObjectItem(device, "pwm_channels");
00538     if (pwm_channels && cJSON_IsNumber(pwm_channels)) {
00539         _device.pwm_channels = pwm_channels->valueint;
00540     }
00541
00542     // max_hardware_timers
00543     cJSON *max_hardware_timers = cJSON_GetObjectItem(device, "max_hardware_timers");
00544     if (max_hardware_timers && cJSON_IsNumber(max_hardware_timers)) {
00545         _device.max_hardware_timers = max_hardware_timers->valueint;
00546     }
00547
00548     // has_rtos
00549     cJSON *has_rtos = cJSON_GetObjectItem(device, "has_rtos");
00550     if (has_rtos && cJSON_IsBool(has_rtos)) {
```

```
00551         _device.has_rtos = cJSON_IsTrue(has_rtos);
00552     }
00553
00554     // UART
00555     cJSON *uart = cJSON_GetObjectItem(device, "UART");
00556     if (uart && cJSON_IsArray(uart)) {
00557         _device.uart_len = cJSON_GetArraySize(uart);
00558         _device.uart = malloc(_device.uart_len * sizeof(int));
00559         if (_device.uart) {
00560             for (size_t i = 0; i < _device.uart_len; i++) {
00561                 cJSON *item = cJSON_GetArrayItem(uart, i);
00562                 if (item && cJSON_IsNumber(item)) {
00563                     _device.uart[i] = item->valueint;
00564                 }
00565             }
00566         } else {
00567             // Log memory allocation error for UART
00568             ESP_LOGE(TAG, "Error allocating memory for UART");
00569         }
00570     }
00571
00572     // I2C
00573     cJSON *i2c = cJSON_GetObjectItem(device, "I2C");
00574     if (i2c && cJSON_IsArray(i2c)) {
00575         _device.i2c_len = cJSON_GetArraySize(i2c);
00576         _device.i2c = malloc(_device.i2c_len * sizeof(int));
00577         if (_device.i2c) {
00578             for (size_t i = 0; i < _device.i2c_len; i++) {
00579                 cJSON *item = cJSON_GetArrayItem(i2c, i);
00580                 if (item && cJSON_IsNumber(item)) {
00581                     _device.i2c[i] = item->valueint;
00582                 }
00583             }
00584         } else {
00585             // Log memory allocation error for I2C
00586             ESP_LOGE(TAG, "Error allocating memory for I2C");
00587         }
00588     }
00589
00590     // SPI
00591     cJSON *spi = cJSON_GetObjectItem(device, "SPI");
00592     if (spi && cJSON_IsArray(spi)) {
00593         _device.spi_len = cJSON_GetArraySize(spi);
00594         _device.spi = malloc(_device.spi_len * sizeof(int));
00595         if (_device.spi) {
00596             for (size_t i = 0; i < _device.spi_len; i++) {
00597                 cJSON *item = cJSON_GetArrayItem(spi, i);
00598                 if (item && cJSON_IsNumber(item)) {
00599                     _device.spi[i] = item->valueint;
00600                 }
00601             }
00602         } else {
00603             // Log memory allocation error for SPI
00604             ESP_LOGE(TAG, "Error allocating memory for SPI");
00605         }
00606     }
00607
00608     // USB
00609     cJSON *usb = cJSON_GetObjectItem(device, "USB");
00610     if (usb && cJSON_IsBool(usb)) {
00611         _device.usb = cJSON_IsTrue(usb);
00612     }
00613
00614     // parent_devices
00615     cJSON *parent_devices = cJSON_GetObjectItem(device, "parent_devices");
00616     if (parent_devices && cJSON_IsArray(parent_devices)) {
00617         _device.parent_devices_len = cJSON_GetArraySize(parent_devices);
00618         _device.parent_devices = malloc(_device.parent_devices_len * sizeof(char *));
00619         if (_device.parent_devices) {
00620             for (size_t i = 0; i < _device.parent_devices_len; i++) {
00621                 cJSON *item = cJSON_GetArrayItem(parent_devices, i);
00622                 if (item && cJSON_IsString(item) && item->valuestring) {
00623                     _device.parent_devices[i] = strdup(item->valuestring);
00624                     if (!_device.parent_devices[i]) {
00625                         // Log memory allocation error for parent_devices
00626                         ESP_LOGE(TAG, "Error allocating memory for parent_devices[%zu]", i);
00627                     }
00628                 } else {
00629                     _device.parent_devices[i] = NULL;
00630                 }
00631             }
00632         } else {
00633             if(_device.parent_devices_len != 0)
00634                 // Log memory allocation error for parent_devices array
00635                 ESP_LOGE(TAG, "Error allocating memory for parent_devices array");
00636         }
00637     }
```

```
00638 }
00639
00640 bool find_pin_by_name(const char *pin_name, gpio_num_t *pin) {
00641     if (!pin_name || !pin) {
00642         return false;
00643     }
00644
00645     // Check digital inputs
00646     for (size_t i = 0; i < _device.digital_inputs_names_len && i < _device.digital_inputs_len; i++) {
00647         if (_device.digital_inputs_names[i] && strcmp(pin_name, _device.digital_inputs_names[i]) == 0)
      {
00648             *pin = (gpio_num_t)_device.digital_inputs[i];
00649             return true;
00650         }
00651     }
00652
00653     // Check digital outputs
00654     for (size_t i = 0; i < _device.digital_outputs_names_len && i < _device.digital_outputs_len; i++)
      {
00655         if (_device.digital_outputs_names[i] && strcmp(pin_name, _device.digital_outputs_names[i]) ==
      0) {
00656             *pin = (gpio_num_t)_device.digital_outputs[i];
00657             return true;
00658         }
00659     }
00660
00661     // Check analog inputs
00662     for (size_t i = 0; i < _device.analog_inputs_names_len && i < _device.analog_inputs_len; i++) {
00663         if (_device.analog_inputs_names[i] && strcmp(pin_name, _device.analog_inputs_names[i]) == 0) {
00664             *pin = (gpio_num_t)_device.analog_inputs[i];
00665             return true;
00666         }
00667     }
00668
00669     // Check DAC outputs
00670     for (size_t i = 0; i < _device.dac_outputs_names_len && i < _device.dac_outputs_len; i++) {
00671         if (_device.dac_outputs_names[i] && strcmp(pin_name, _device.dac_outputs_names[i]) == 0) {
00672             *pin = (gpio_num_t)_device.dac_outputs[i];
00673             return true;
00674         }
00675     }
00676
00677     // Check OneWire inputs
00678     for (size_t i = 0; i < _device.one_wire_inputs_len; i++) {
00679         for (size_t j = 0; j < _device.one_wire_inputs_names_len[i]; j++) {
00680             if (_device.one_wire_inputs_names[i][j] && strcmp(pin_name,
      _device.one_wire_inputs_names[i][j]) == 0) {
00681                 *pin = (gpio_num_t)_device.one_wire_inputs[i];
00682                 return true;
00683             }
00684         }
00685     }
00686
00687     return false;
00688 }
00689
00690 // ================== INITIALIZATION DIGITAL I/O ==================
00694 void init_digital_inputs(void) {
00695     if (_device.digital_inputs == NULL || _device.digital_inputs_len == 0) {
00696         // Log warning if no digital inputs are available
00697         ESP_LOGW(TAG, "No digital inputs to initialize");
00698         return;
00699     }
00700
00701     for (size_t i = 0; i < _device.digital_inputs_len; i++) {
00702         gpio_num_t pin = (gpio_num_t)_device.digital_inputs[i];
00703         const char *name = _device.digital_inputs_names && i < _device.digital_inputs_names_len &&
      _device.digital_inputs_names[i] ? _device.digital_inputs_names[i] : "unnamed";
00704
00705         // Log initialization of digital input
00706         ESP_LOGI(TAG, "Initializing digital input %s on GPIO %d", name, pin);
00707
00708         gpio_config_t io_conf = {
00709             .intr_type = GPIO_INTR_DISABLE,
00710             .mode = GPIO_MODE_INPUT,
00711             .pin_bit_mask = (1ULL << pin),
00712             .pull_down_en = GPIO_PULLDOWN_DISABLE,
00713             .pull_up_en = GPIO_PULLUP_DISABLE
00714         };
00715
00716         esp_err_t err = gpio_config(&io_conf);
00717         if (err != ESP_OK) {
00718             // Log error during GPIO configuration
00719             ESP_LOGE(TAG, "Error configuring GPIO %d: %s", pin, esp_err_to_name(err));
00720             continue;
00721         }
00722
```

```
00723         // Log successful initialization
00724         ESP_LOGI(TAG, "Digital input %s (GPIO %d) successfully initialized", name, pin);
00725     }
00726 }
00727
00731 void init_digital_outputs(void) {
00732     if (_device.digital_outputs == NULL || _device.digital_outputs_len == 0) {
00733         // Log warning if no digital outputs are available
00734         ESP_LOGW(TAG, "No digital outputs to initialize");
00735         return;
00736     }
00737
00738     for (size_t i = 0; i < _device.digital_outputs_len; i++) {
00739         gpio_num_t pin = (gpio_num_t)_device.digital_outputs[i];
00740         const char *name = _device.digital_outputs_names && i < _device.digital_outputs_names_len &&
      _device.digital_outputs_names[i] ? _device.digital_outputs_names[i] : "unnamed";
00741
00742         // Log initialization of digital output
00743         ESP_LOGI(TAG, "Initializing digital output %s on GPIO %d", name, pin);
00744
00745         gpio_config_t io_conf = {
00746             .intr_type = GPIO_INTR_DISABLE,                 // Disable interrupts
00747             .mode = GPIO_MODE_INPUT_OUTPUT,                 // Set as output (with input for reading)
00748             .pin_bit_mask = (1ULL << pin),                 // Pin mask (64-bit)
00749             .pull_down_en = GPIO_PULLDOWN_DISABLE,          // Disable pull-down
00750             .pull_up_en = GPIO_PULLUP_DISABLE               // Disable pull-up
00751         };
00752
00753         esp_err_t err = gpio_config(&io_conf);
00754         if (err != ESP_OK) {
00755             // Log error during GPIO configuration
00756             ESP_LOGE(TAG, "Error configuring GPIO %d: %s", pin, esp_err_to_name(err));
00757             continue;
00758         }
00759
00760         // Log successful initialization
00761         ESP_LOGI(TAG, "Digital output %s (GPIO %d) successfully initialized", name, pin);
00762     }
00763 }
00764
00765 // ========== INITIALIZATION ANALOG I/O (not implemented) ==================
00769 void init_analog_inputs(void) {
00770
00771 }
00772
00776 void init_analog_outputs(void) {
00777
00778 }
00779
00780 // ================ INITIALIZATION ONEWIRE INPUTS ==================
00784 void init_one_wire_inputs(void) {
00785     if (_device.one_wire_inputs == NULL || _device.one_wire_inputs_len == 0) {
00786         // Log warning if no one-wire inputs are available
00787         ESP_LOGW(TAG, "No OneWire inputs to initialize");
00788         return;
00789     }
00790
00791     for (size_t i = 0; i < _device.one_wire_inputs_len; i++) {
00792         gpio_num_t pin = (gpio_num_t)_device.one_wire_inputs[i];
00793
00794         // Log initialization of one-wire input
00795         ESP_LOGI(TAG, "Initializing OneWire input on GPIO %d", pin);
00796
00797         // Reset pin to default state (commented out)
00798         // esp_err_t err; // = gpio_reset_pin(pin);
00799         // if (err != ESP_OK) {
00800         //     ESP_LOGE(TAG, "Error resetting GPIO %d: %s", pin, esp_err_to_name(err));
00801         //     continue;
00802         // }
00803
00804         // Set direction to input
00805         esp_err_t err = gpio_set_direction(pin, GPIO_MODE_INPUT);
00806         if (err != ESP_OK) {
00807             // Log error setting GPIO direction
00808             ESP_LOGE(TAG, "Error setting direction for GPIO %d: %s", pin, esp_err_to_name(err));
00809             continue;
00810         }
00811
00812         // Set pull-up mode (required for OneWire)
00813         err = gpio_set_pull_mode(pin, GPIO_PULLUP_ONLY);
00814         if (err != ESP_OK) {
00815             // Log error setting pull mode
00816             ESP_LOGE(TAG, "Error setting pull mode for GPIO %d: %s", pin, esp_err_to_name(err));
00817             continue;
00818         }
00819
00820         // Log successful initialization
```

```
00821         ESP_LOGI(TAG, "OneWire input (GPIO %d) successfully initialized", pin);
00822     }
00823 }
00824
00825 // ================== DEVICE INITIALIZATION ==================
00826 void device_init(cJSON *device)
00827 {
00828     load_device_configuration(device);
00829
00830     init_digital_inputs();
00831     init_digital_outputs();
00832     init_analog_inputs();
00833     init_analog_outputs();
00834     init_one_wire_inputs();
00835 }
00836
00837 // ========================= DIGITAL I/O ==========================
00838 bool get_digital_input_value(const char *pin_name) {
00839     gpio_num_t pin;
00840     if (!find_pin_by_name(pin_name, &pin)) {
00841         // Log error if digital input is not found
00842         ESP_LOGE(TAG, "Digital input %s not found", pin_name);
00843         return -1;
00844     }
00845
00846     bool value = gpio_get_level(pin);
00847     // Log digital input value (commented out)
00848     // ESP_LOGI(TAG, "Digital input %s (GPIO %d): %d", pin_name, pin, value);
00849     return value;
00850 }
00851
00852 esp_err_t set_digital_output_value(const char *pin_name, int value) {
00853     gpio_num_t pin;
00854     if (!find_pin_by_name(pin_name, &pin)) {
00855         // Log error if digital output is not found
00856         ESP_LOGE(TAG, "Digital output %s not found", pin_name);
00857         return ESP_ERR_NOT_FOUND;
00858     }
00859
00860     esp_err_t err = gpio_set_level(pin, value);
00861     if (err != ESP_OK) {
00862         // Log error setting digital output
00863         ESP_LOGE(TAG, "Error setting digital output %s (GPIO %d): %s", pin_name, pin,
      esp_err_to_name(err));
00864         return err;
00865     }
00866
00867     // Log digital output value set (commented out)
00868     // ESP_LOGI(TAG, "Digital output %s (GPIO %d) set to %d", pin_name, pin, value);
00869     return ESP_OK;
00870 }
00871
00872 bool get_digital_output_value(const char *pin_name) {
00873     gpio_num_t pin;
00874     if (!find_pin_by_name(pin_name, &pin)) {
00875         // Log error if digital output is not found
00876         ESP_LOGE(TAG, "Digital output %s not found", pin_name);
00877         return -1;
00878     }
00879
00880     bool value = gpio_get_level(pin);
00881     // Log digital output value (commented out)
00882     // ESP_LOGI(TAG, "Digital output %s (GPIO %d): %d", pin_name, pin, value);
00883     return value;
00884 }
00885
00886 // ================== ANALOG I/O (not implemented) ===========================
00887 int get_analog_input_value(const char *pin_name) {
00888     return -1;
00889 }
00890
00891 esp_err_t set_analog_output_value(const char *pin_name, uint8_t value) {
00892     return ESP_OK;
00893 }
00894
00895 int get_analog_output_value(const char *pin_name) {
00896     return -1;
00897 }
00898
00899 // ======================== ONE WIRE ==========================
00900 float get_one_wire_value(const char *pin_name) {
00901     if (!pin_name) {
00902         // Log error if pin name is not provided
00903         ESP_LOGE(TAG, "No OneWire input name provided");
00904         return -1.0f;
00905     }
00906
```

```
00907     // Find the corresponding OneWire pin and device index
00908     for (size_t i = 0; i < _device.one_wire_inputs_len; i++) {
00909         if (_device.one_wire_inputs_names[i]) {
00910             for (size_t j = 0; j < _device.one_wire_inputs_names_len[i]; j++) {
00911                 if (_device.one_wire_inputs_names[i][j] && strcmp(pin_name,
      _device.one_wire_inputs_names[i][j]) == 0) {
00912                     // Found the name, check type and address
00913                     if (j < _device.one_wire_inputs_devices_types_len[i] && j <
      _device.one_wire_inputs_devices_addresses_len[i]  &&  i < _device.one_wire_inputs_len) {
00914                         const char *sensor_type = _device.one_wire_inputs_devices_types[i][j];
00915                         const char *sensor_address = _device.one_wire_inputs_devices_addresses[i][j];
00916                         gpio_num_t pin = (gpio_num_t)_device.one_wire_inputs[i];
00917                         if (sensor_type && sensor_address) {
00918                             // Call external function to read the sensor
00919                             float value = read_one_wire_sensor(sensor_type, sensor_address, pin);
00920                             return value;
00921                             return 0; // Note: This line is unreachable due to the previous return
00922                         } else {
00923                             // Log error if sensor type or address is missing
00924                             ESP_LOGE(TAG, "Missing type or address for OneWire sensor %s", pin_name);
00925                             return -1.0f;
00926                         }
00927                     } else {
00928                         // Log error if array lengths do not match
00929                         ESP_LOGE(TAG, "Array length mismatch for OneWire sensor %s", pin_name);
00930                         return -1.0f;
00931                     }
00932                 }
00933             }
00934         }
00935     }
00936
00937     // Log error if OneWire input is not found
00938     ESP_LOGE(TAG, "OneWire input %s not found", pin_name);
00939     return -1.0f;
00940 }
```

4.17 main/device_config.h File Reference

```
#include "driver/gpio.h"
#include "cJSON.h"
#include "esp_log.h"
```

Data Structures

- struct Device

  Structure defining the device configuration.

Functions

- void print_device_info (void)

  Prints information about the device configuration.
- bool find_pin_by_name (const char *pin_name, gpio_num_t *pin)

  Finds a GPIO pin number based on its name.
- void device_init (cJSON *device)

  Initializes the device configuration from a JSON object.
- bool get_digital_input_value (const char *pin_name)

  Gets the value of a digital input pin by its name.
- esp_err_t set_digital_output_value (const char *pin_name, int value)

  Sets the value of a digital output pin by its name.
- bool get_digital_output_value (const char *pin_name)

  Gets the value of a digital output pin by its name.

- int get_analog_input_value (const char *pin_name)
  Gets the value of an analog input pin by its name.
- esp_err_t set_analog_output_value (const char *pin_name, uint8_t value)
  Sets the value of an analog output (DAC) pin by its name.
- int get_analog_output_value (const char *pin_name)
  Gets the value of an analog output (DAC) pin by its name.
- float get_one_wire_value (const char *pin_name)
  Gets the value from a one-wire device by its pin name.

Variables

- Device _device
  Global variable holding the device configuration.

4.17.1 Function Documentation

4.17.1.1 device_init()

```
void device_init (
            cJSON * device)
```

Initializes the device configuration from a JSON object.

Parameters

device	JSON object containing the device configuration.

Definition at line 826 of file device_config.c.

4.17.1.2 find_pin_by_name()

```
bool find_pin_by_name (
            const char * pin_name,
            gpio_num_t * pin)
```

Finds a GPIO pin number based on its name.

Parameters

pin_name	Name of the pin to find.
pin	Pointer to store the found GPIO pin number.

Returns

bool True if the pin is found, false otherwise.

Definition at line 640 of file device_config.c.

4.17.1.3 get_analog_input_value()

```
int get_analog_input_value (
            const char * pin_name)
```

Gets the value of an analog input pin by its name.

Parameters

pin_name	Name of the analog input pin.

Returns

int The analog input value.

Definition at line 887 of file device_config.c.

4.17.1.4 get_analog_output_value()

```
int get_analog_output_value (
            const char * pin_name)
```

Gets the value of an analog output (DAC) pin by its name.

Parameters

pin_name	Name of the DAC output pin.

Returns

int The current DAC output value.

Definition at line 895 of file device_config.c.

4.17.1.5 get_digital_input_value()

```
bool get_digital_input_value (
            const char * pin_name)
```

Gets the value of a digital input pin by its name.

Parameters

pin_name	Name of the digital input pin.

Returns

bool The value of the digital input (true for high, false for low).

Definition at line 838 of file device_config.c.

4.17.1.6 get_digital_output_value()

```
bool get_digital_output_value (
            const char * pin_name)
```

Gets the value of a digital output pin by its name.

Parameters

pin_name	Name of the digital output pin.

Returns

bool The value of the digital output (true for high, false for low).

Definition at line 872 of file device_config.c.

4.17.1.7 get_one_wire_value()

```
float get_one_wire_value (
            const char * pin_name)
```

Gets the value from a one-wire device by its pin name.

Parameters

pin_name	Name of the one-wire input pin.

Returns

float The value read from the one-wire device.

Definition at line 900 of file device_config.c.

4.17.1.8 print_device_info()

```
void print_device_info (
            void )
```

Prints information about the device configuration.

Definition at line 113 of file device_config.c.

4.17.1.9 set_analog_output_value()

```
esp_err_t set_analog_output_value (
            const char * pin_name,
            uint8_t value)
```

Sets the value of an analog output (DAC) pin by its name.

Parameters

pin_name	Name of the DAC output pin.
value	Value to set (typically 0-255 for 8-bit DAC).

Returns

esp_err_t ESP_OK on success, or an error code on failure.

Definition at line 891 of file device_config.c.

4.17.1.10 set_digital_output_value()

```
esp_err_t set_digital_output_value (
            const char * pin_name,
            int value)
```

Sets the value of a digital output pin by its name.

Parameters

pin_name	Name of the digital output pin.
value	Value to set (0 for low, non-zero for high).

Returns

esp_err_t ESP_OK on success, or an error code on failure.

Definition at line 852 of file device_config.c.

4.17.2 Variable Documentation

4.17.2.1 _device

```
Device _device  [extern]
```

Global variable holding the device configuration.

Global variable holding the device configuration.

Definition at line 18 of file device_config.c.

4.18 device_config.h

Go to the documentation of this file.

```
00001 #ifndef DEVICE_CONFIG_H
00002 #define DEVICE_CONFIG_H
00003
00004 #include "driver/gpio.h"
00005 #include "cJSON.h"
00006 #include "esp_log.h"
00007
00011 typedef struct {
00012     char *device_name;
00013     double logic_voltage;
00014
00015     // Digital I/O
00016     int *digital_inputs;
00017     size_t digital_inputs_len;
00018     char **digital_inputs_names;
00019     size_t digital_inputs_names_len;
00020     int *digital_outputs;
00021     size_t digital_outputs_len;
00022     char **digital_outputs_names;
00023     size_t digital_outputs_names_len;
00024
00025     // Analog I/O
00026     int *analog_inputs;
00027     size_t analog_inputs_len;
00028     char **analog_inputs_names;
```

```
00029     size_t analog_inputs_names_len;
00030     int *dac_outputs;
00031     size_t dac_outputs_len;
00032     char **dac_outputs_names;
00033     size_t dac_outputs_names_len;
00034
00035     // One-Wire
00036     int *one_wire_inputs;
00037     size_t one_wire_inputs_len;
00038     char ***one_wire_inputs_names;
00039     size_t *one_wire_inputs_names_len;
00040     char ***one_wire_inputs_devices_types;
00041     size_t *one_wire_inputs_devices_types_len;
00042     char ***one_wire_inputs_devices_addresses;
00043     size_t *one_wire_inputs_devices_addresses_len;
00044
00045     // Other
00046     int pwm_channels;
00047     int max_hardware_timers;
00048     bool has_rtos;
00049     int *uart;
00050     size_t uart_len;
00051     int *i2c;
00052     size_t i2c_len;
00053     int *spi;
00054     size_t spi_len;
00055     bool usb;
00056
00057     char **parent_devices;
00058     size_t parent_devices_len;
00059 } Device;
00060
00064 extern Device _device;
00065
00069 void print_device_info(void);
00070
00077 bool find_pin_by_name(const char *pin_name, gpio_num_t *pin);
00078
00083 void device_init(cJSON *device);
00084
00090 bool get_digital_input_value(const char *pin_name);
00091
00098 esp_err_t set_digital_output_value(const char *pin_name, int value);
00099
00105 bool get_digital_output_value(const char *pin_name);
00106
00112 int get_analog_input_value(const char *pin_name);
00113
00120 esp_err_t set_analog_output_value(const char *pin_name, uint8_t value);
00121
00127 int get_analog_output_value(const char *pin_name);
00128
00134 float get_one_wire_value(const char *pin_name);
00135
00136 #endif // DEVICE_CONFIG_H
```

4.19 main/ladder_elements.c File Reference

```
#include "ladder_elements.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include <string.h>
#include <math.h>
#include <stdbool.h>
#include "esp_timer.h"
#include "device_config.h"
#include "variables.h"
```

Data Structures

- struct OneShotState

Structure to track the previous state for one-shot positive coils.

- struct TimerState

  Structure to track the state of timers.

Functions

- static bool * get_one_shot_state (const char *var_name)

  Helper function to find or add a one-shot state for a variable.
- static TimerState * get_timer_state (const char *var_name)

  Helper function to find or add a timer state for a variable.
- bool r_trig (const char *var_name, bool condition)

  Detects a rising edge (transition from false to true) for a variable.
- bool f_trig (const char *var_name, bool condition)

  Detects a falling edge (transition from true to false) for a variable.
- bool no_contact (const char *var_name)

  Normally Open (NO) Contact: Returns true (active) when the associated signal is true, otherwise false (inactive).
- bool nc_contact (const char *var_name)

  Normally Closed (NC) Contact: Returns true (active) when the associated signal is false (inactive), otherwise false (inactive).
- void coil (const char *var_name, bool condition)

  Coil: Writes the current value (true/false) to the target variable.
- void one_shot_positive_coil (const char *var_name, bool condition)

  One Shot Positive Coil: Writes true only on the rising edge (first time the signal becomes true), otherwise false.
- void set_coil (const char *var_name, bool condition)

  Set Coil: Writes true when the signal is true and retains the value until reset.
- void reset_coil (const char *var_name, bool condition)

  Reset Coil: Writes false when the signal is true and retains the value until set.
- void add (const char *var_name_a, const char *var_name_b, const char *var_name_c, bool condition)

  Add: Performs addition (A + B = C).
- void subtract (const char *var_name_a, const char *var_name_b, const char *var_name_c, bool condition)

  Subtract: Performs subtraction (A - B = C).
- void multiply (const char *var_name_a, const char *var_name_b, const char *var_name_c, bool condition)

  *Multiply: Performs multiplication (A * B = C).*
- void divide (const char *var_name_a, const char *var_name_b, const char *var_name_c, bool condition)

  Divide: Performs division (A / B = C).
- void move (const char *var_name_a, const char *var_name_b, bool condition)

  Move: Copies the value of A to B.
- bool greater (const char *var_name_a, const char *var_name_b)

  Greater: Checks if A is greater than B (A > B).
- bool less (const char *var_name_a, const char *var_name_b)

  Less: Checks if A is less than B (A < B).
- bool greater_or_equal (const char *var_name_a, const char *var_name_b)

  Greater or Equal: Checks if A is greater than or equal to B (A >= B).
- bool less_or_equal (const char *var_name_a, const char *var_name_b)

  Less or Equal: Checks if A is less than or equal to B (A <= B).
- bool equal (const char *var_name_a, const char *var_name_b)

  Equal: Checks if A is equal to B (A == B).
- bool not_equal (const char *var_name_a, const char *var_name_b)

  Not Equal: Checks if A is not equal to B (A != B).
- void count_up (const char *var_name, bool condition)

Count Up: Increments the counter variable.

- void count_down (const char *var_name, bool condition)

  Count Down: Decrements the counter variable.
- bool timer_on (const char *var_name, bool condition)

  Timer On-Delay: Activates the timer with an on-delay mechanism.
- bool timer_off (const char *var_name, bool condition)

  Timer Off-Delay: Activates the timer with an off-delay mechanism.
- void reset (const char *var_name, bool condition)

  Reset: Resets the specified variable (e.g., counter or timer).

Variables

- static const char * TAG = "LADDER"

  Tag for logging messages from the ladder logic module.
- static OneShotState one_shot_states [MAX_ONE_SHOT_STATES] = {0}

  Array to store one-shot states.
- static size_t one_shot_count = 0

  Current number of one-shot states.
- static TimerState timer_states [MAX_TIMER_STATES] = {0}

  Array to store timer states.
- static size_t timer_state_count = 0

  Current number of timer states.

4.19.1 Function Documentation

4.19.1.1 add()

```
void add (
            const char * var_name_a,
            const char * var_name_b,
            const char * var_name_c,
            bool condition)
```

Add: Performs addition (A + B = C).

Parameters

var_name_a	Name of the first input variable.
var_name_b	Name of the second input variable.
var_name_c	Name of the output variable.
condition	Condition to enable the operation.

Definition at line 197 of file ladder_elements.c.

4.19.1.2 coil()

```
void coil (
            const char * var_name,
            bool condition)
```

Coil: Writes the current value (true/false) to the target variable.

Parameters

var_name	Name of the target variable.
condition	The condition value to write.

Definition at line 160 of file ladder_elements.c.

4.19.1.3 count_down()

```
void count_down (
            const char * var_name,
            bool condition)
```

Count Down: Decrements the counter variable.

Parameters

var_name	Name of the counter variable.
condition	Condition to enable counting.

Definition at line 317 of file ladder_elements.c.

4.19.1.4 count_up()

```
void count_up (
            const char * var_name,
            bool condition)
```

Count Up: Increments the counter variable.

Parameters

var_name	Name of the counter variable.
condition	Condition to enable counting.

Definition at line 305 of file ladder_elements.c.

4.19.1.5 divide()

```
void divide (
            const char * var_name_a,
            const char * var_name_b,
            const char * var_name_c,
            bool condition)
```

Divide: Performs division (A / B = C).

Parameters

var_name↩ _a	Name of the first input variable.
var_name↩ _b	Name of the second input variable.
var_name↩ _c	Name of the output variable.
condition	Condition to enable the operation.

Definition at line 228 of file ladder_elements.c.

4.19.1.6 equal()

```
bool equal (
            const char * var_name_a,
            const char * var_name_b)
```

Equal: Checks if A is equal to B (A == B).

Parameters

var_name↩ _a	Name of the first variable.
var_name↩ _b	Name of the second variable.

Returns

bool True if A == B, false otherwise.

Definition at line 288 of file ladder_elements.c.

4.19.1.7 f_trig()

```
bool f_trig (
            const char * var_name,
            bool condition)
```

Detects a falling edge (transition from true to false) for a variable.

Parameters

var_name	Name of the variable.
condition	Current condition to evaluate.

Returns

bool True if a falling edge is detected, false otherwise.

Definition at line 130 of file ladder_elements.c.

4.19.1.8 get_one_shot_state()

```
static bool * get_one_shot_state (
            const char * var_name)  [static]
```

Helper function to find or add a one-shot state for a variable.

Parameters

var_name	Name of the variable.

Returns

bool* Pointer to the previous state, or NULL if the limit is exceeded.

Definition at line 60 of file ladder_elements.c.

4.19.1.9 get_timer_state()

```
static TimerState * get_timer_state (
            const char * var_name)  [static]
```

Helper function to find or add a timer state for a variable.

Parameters

var_name	Name of the timer variable.

Returns

TimerState* Pointer to the timer state, or NULL if the limit is exceeded.

Definition at line 84 of file ladder_elements.c.

4.19.1.10 greater()

```
bool greater (
            const char * var_name_a,
            const char * var_name_b)
```

Greater: Checks if A is greater than B (A > B).

Parameters

var_name_a	Name of the first variable.
var_name_b	Name of the second variable.

Returns

bool True if A > B, false otherwise.

Definition at line 256 of file ladder_elements.c.

4.19.1.11 greater_or_equal()

```
bool greater_or_equal (
            const char * var_name_a,
            const char * var_name_b)
```

Greater or Equal: Checks if A is greater than or equal to B (A >= B).

Parameters

var_name↩ _a	Name of the first variable.
var_name↩ _b	Name of the second variable.

Returns

bool True if A >= B, false otherwise.

Definition at line 272 of file ladder_elements.c.

4.19.1.12 less()

```
bool less (
            const char * var_name_a,
            const char * var_name_b)
```

Less: Checks if A is less than B (A < B).

Parameters

var_name↩ _a	Name of the first variable.
var_name↩ _b	Name of the second variable.

Returns

bool True if A < B, false otherwise.

Definition at line 264 of file ladder_elements.c.

4.19.1.13 less_or_equal()

```
bool less_or_equal (
            const char * var_name_a,
            const char * var_name_b)
```

Less or Equal: Checks if A is less than or equal to B (A <= B).

Parameters

var_name↩ _a	Name of the first variable.
var_name↩ _b	Name of the second variable.

Returns

bool True if A <= B, false otherwise.

Definition at line 280 of file ladder_elements.c.

4.19.1.14 move()

```
void move (
            const char * var_name_a,
            const char * var_name_b,
            bool condition)
```

Move: Copies the value of A to B.

Parameters

var_name_a	Name of the source variable.
var_name_b	Name of the destination variable.
condition	Condition to enable the operation.

Definition at line 246 of file ladder_elements.c.

4.19.1.15 multiply()

```
void multiply (
            const char * var_name_a,
            const char * var_name_b,
            const char * var_name_c,
            bool condition)
```

Multiply: Performs multiplication (A * B = C).

Parameters

var_name_a	Name of the first input variable.
var_name_b	Name of the second input variable.
var_name_c	Name of the output variable.
condition	Condition to enable the operation.

Definition at line 218 of file ladder_elements.c.

4.19.1.16 nc_contact()

```
bool nc_contact (
            const char * var_name)
```

Normally Closed (NC) Contact: Returns true (active) when the associated signal is false (inactive), otherwise false (inactive).

Parameters

var_name	Name of the variable to check.

Returns

bool True if the signal is inactive, false otherwise.

Definition at line 152 of file ladder_elements.c.

4.19.1.17 no_contact()

```
bool no_contact (
            const char * var_name)
```

Normally Open (NO) Contact: Returns true (active) when the associated signal is true, otherwise false (inactive).

Parameters

var_name	Name of the variable to check.

Returns

bool True if the signal is active, false otherwise.

Definition at line 145 of file ladder_elements.c.

4.19.1.18 not_equal()

```
bool not_equal (
            const char * var_name_a,
            const char * var_name_b)
```

Not Equal: Checks if A is not equal to B (A != B).

Parameters

var_name_a	Name of the first variable.
var_name_b	Name of the second variable.

Returns

bool True if A != B, false otherwise.

Definition at line 296 of file ladder_elements.c.

4.19.1.19 one_shot_positive_coil()

```
void one_shot_positive_coil (
            const char * var_name,
            bool condition)
```

One Shot Positive Coil: Writes true only on the rising edge (first time the signal becomes true), otherwise false.

Parameters

var_name	Name of the target variable.
condition	The condition to evaluate for the rising edge.

Definition at line 166 of file ladder_elements.c.

4.19.1.20 r_trig()

```
bool r_trig (
            const char * var_name,
            bool condition)
```

Detects a rising edge (transition from false to true) for a variable.

Parameters

var_name	Name of the variable.
condition	Current condition to evaluate.

Returns

bool True if a rising edge is detected, false otherwise.

Definition at line 110 of file ladder_elements.c.

4.19.1.21 reset()

```
void reset (
            const char * var_name,
            bool condition)
```

Reset: Resets the specified variable (e.g., counter or timer).

Parameters

var_name	Name of the variable to reset.
condition	Condition to trigger the reset.

Definition at line 454 of file ladder_elements.c.

4.19.1.22 reset_coil()

```
void reset_coil (
            const char * var_name,
            bool condition)
```

Reset Coil: Writes false when the signal is true and retains the value until set.

Parameters

var_name	Name of the target variable.
condition	The condition to reset the coil.

Definition at line 188 of file ladder_elements.c.

4.19.1.23 set_coil()

```
void set_coil (
            const char * var_name,
            bool condition)
```

Set Coil: Writes true when the signal is true and retains the value until reset.

Parameters

var_name	Name of the target variable.
condition	The condition to set the coil.

Definition at line 180 of file ladder_elements.c.

4.19.1.24 subtract()

```
void subtract (
            const char * var_name_a,
            const char * var_name_b,
            const char * var_name_c,
            bool condition)
```

Subtract: Performs subtraction (A - B = C).

Parameters

var_name_a	Name of the first input variable.
var_name_b	Name of the second input variable.
var_name_c	Name of the output variable.
condition	Condition to enable the operation.

Definition at line 207 of file ladder_elements.c.

4.19.1.25 timer_off()

```
bool timer_off (
            const char * var_name,
            bool condition)
```

Timer Off-Delay: Activates the timer with an off-delay mechanism.

Parameters

var_name	Name of the timer variable.
condition	Condition to start the timer.

Returns

bool True when the timer is active, false after the delay expires.

Definition at line 392 of file ladder_elements.c.

4.19.1.26 timer_on()

```
bool timer_on (
            const char * var_name,
            bool condition)
```

Timer On-Delay: Activates the timer with an on-delay mechanism.

Parameters

var_name	Name of the timer variable.
condition	Condition to start the timer.

Returns

bool True when the timer reaches its setpoint, false otherwise.

Definition at line 329 of file ladder_elements.c.

4.19.2 Variable Documentation

4.19.2.1 one_shot_count

```
size_t one_shot_count = 0  [static]
```

Current number of one-shot states.

Definition at line 43 of file ladder_elements.c.

4.19.2.2 one_shot_states

```
OneShotState one_shot_states[MAX_ONE_SHOT_STATES] = {0}  [static]
```

Array to store one-shot states.

Definition at line 38 of file ladder_elements.c.

4.19.2.3 TAG

```
const char* TAG = "LADDER"  [static]
```

Tag for logging messages from the ladder logic module.

Definition at line 16 of file ladder_elements.c.

4.19.2.4 timer_state_count

```
size_t timer_state_count = 0  [static]
```

Current number of timer states.

Definition at line 53 of file ladder_elements.c.

4.19.2.5 timer_states

```
TimerState timer_states[MAX_TIMER_STATES] = {0}  [static]
```

Array to store timer states.

Definition at line 48 of file ladder_elements.c.

4.20 ladder_elements.c

Go to the documentation of this file.

```
00001 #include "ladder_elements.h"
00002 #include "freertos/FreeRTOS.h"
00003 #include "freertos/task.h"
00004 #include "esp_log.h"
00005 #include <string.h>
00006 #include <math.h>
00007 #include <stdbool.h>
00008 #include "esp_timer.h"
00009
00010 #include "device_config.h"
00011 #include "variables.h"
00012
00016 static const char *TAG = "LADDER";
00017
00021 typedef struct {
00022     char var_name[MAX_VAR_NAME_LENGTH];
00023     bool prev_state;
00024 } OneShotState;
00025
00029 typedef struct {
00030     char var_name[MAX_VAR_NAME_LENGTH];
00031     int64_t start_time;
00032     bool running;
00033 } TimerState;
00034
00038 static OneShotState one_shot_states[MAX_ONE_SHOT_STATES] = {0};
00039
00043 static size_t one_shot_count = 0;
00044
00048 static TimerState timer_states[MAX_TIMER_STATES] = {0};
00049
00053 static size_t timer_state_count = 0;
00054
00060 static bool *get_one_shot_state(const char *var_name) {
00061     // Check if the state already exists
00062     for (size_t i = 0; i < one_shot_count; i++) {
00063         if (strcmp(one_shot_states[i].var_name, var_name) == 0) {
```

```
00064             return &one_shot_states[i].prev_state;
00065         }
00066     }
00067     // Add a new state if within limits
00068     if (one_shot_count < MAX_ONE_SHOT_STATES) {
00069         strncpy(one_shot_states[one_shot_count].var_name, var_name, MAX_VAR_NAME_LENGTH - 1);
00070         one_shot_states[one_shot_count].var_name[MAX_VAR_NAME_LENGTH - 1] = '\0';
00071         one_shot_states[one_shot_count].prev_state = false;
00072         one_shot_count++;
00073         return &one_shot_states[one_shot_count - 1].prev_state;
00074     }
00075     ESP_LOGE(TAG, "Too many one-shot states for %s", var_name);
00076     return NULL;
00077 }
00078
00084 static TimerState *get_timer_state(const char *var_name) {
00085     // Check if the state already exists
00086     for (size_t i = 0; i < timer_state_count; i++) {
00087         if (strcmp(timer_states[i].var_name, var_name) == 0) {
00088             return &timer_states[i];
00089         }
00090     }
00091     // Add a new state if within limits
00092     if (timer_state_count < MAX_TIMER_STATES) {
00093         strncpy(timer_states[timer_state_count].var_name, var_name, MAX_VAR_NAME_LENGTH - 1);
00094         timer_states[timer_state_count].var_name[MAX_VAR_NAME_LENGTH - 1] = '\0';
00095         timer_states[timer_state_count].start_time = 0;
00096         timer_states[timer_state_count].running = false;
00097         timer_state_count++;
00098         return &timer_states[timer_state_count - 1];
00099     }
00100     ESP_LOGE(TAG, "Too many timer states for %s", var_name);
00101     return NULL;
00102 }
00103
00110 bool r_trig(const char *var_name, bool condition) {
00111     bool *prev_state = get_one_shot_state(var_name);
00112     if (!prev_state) {
00113         return false;
00114     }
00115
00116     bool result = condition && !(*prev_state);
00117     *prev_state = condition;
00118
00119     // Log the rising edge detection (commented out)
00120     // ESP_LOGI(TAG, "R_TRIG: %s condition=%d, result=%d", var_name, condition, result);
00121     return result;
00122 }
00123
00130 bool f_trig(const char *var_name, bool condition) {
00131     bool *prev_state = get_one_shot_state(var_name);
00132     if (!prev_state) {
00133         return false;
00134     }
00135
00136     bool result = !condition && (*prev_state);
00137     *prev_state = condition;
00138
00139     // Log the falling edge detection (commented out)
00140     // ESP_LOGI(TAG, "F_TRIG: %s condition=%d, result=%d", var_name, condition, result);
00141     return result;
00142 }
00143
00144 // ============== CONTACTS ===============
00145 bool no_contact(const char *var_name) {
00146     bool result = read_variable(var_name);
00147     // Log the NO contact state (commented out)
00148     // ESP_LOGI(TAG, "NO Contact: %s value=%d", var_name, result);
00149     return !result;
00150 }
00151
00152 bool nc_contact(const char *var_name) {
00153     bool result = read_variable(var_name);
00154     // Log the NC contact state (commented out)
00155     // ESP_LOGI(TAG, "NC Contact: %s value=%d", var_name, result);
00156     return result;
00157 }
00158
00159 // ============== COILS ================
00160 void coil(const char *var_name, bool condition) {
00161     // Log the coil operation (commented out)
00162     // ESP_LOGI(TAG, "Coil: %s value=%d", var_name, condition);
00163     write_variable(var_name, condition);
00164 }
00165
00166 void one_shot_positive_coil(const char *var_name, bool condition) {
00167     bool *prev_state = get_one_shot_state(var_name);
```

```
00168     if (!prev_state) {
00169         return;
00170     }
00171
00172     bool output = condition && !(*prev_state);
00173     *prev_state = condition;
00174
00175     // Log the one-shot positive coil operation (commented out)
00176     // ESP_LOGI(TAG, "One Shot Positive Coil: %s value=%d", var_name, output);
00177     write_variable(var_name, output);
00178 }
00179
00180 void set_coil(const char *var_name, bool condition) {
00181     if (condition) {
00182         // Log the set coil operation (commented out)
00183         // ESP_LOGI(TAG, "Set Coil: %s value=%d", var_name, condition);
00184         write_variable(var_name, true);
00185     }
00186 }
00187
00188 void reset_coil(const char *var_name, bool condition) {
00189     if (condition) {
00190         // Log the reset coil operation (commented out)
00191         // ESP_LOGI(TAG, "Reset Coil: %s value=%d", var_name, condition);
00192         write_variable(var_name, false);
00193     }
00194 }
00195
00196 // ============== MATH ===============
00197 void add(const char *var_name_a, const char *var_name_b, const char *var_name_c, bool condition) {
00198     if (r_trig(var_name_c, condition)) {
00199         double a = read_numeric_variable(var_name_a);
00200         double b = read_numeric_variable(var_name_b);
00201         // Log the add operation (commented out)
00202         // ESP_LOGI(TAG, "Add: %s (%f) + %s (%f) => %s (%f)", var_name_a, a, var_name_b, b,
      var_name_c, a + b);
00203         write_numeric_variable(var_name_c, a + b);
00204     }
00205 }
00206
00207 void subtract(const char *var_name_a, const char *var_name_b, const char *var_name_c, bool condition)
      {
00208     if (r_trig(var_name_c, condition))
00209     {
00210         double a = read_numeric_variable(var_name_a);
00211         double b = read_numeric_variable(var_name_b);
00212         // Log the subtract operation (commented out)
00213         // ESP_LOGI(TAG, "Subtract: %s (%f) - %s (%f) => %s (%f)", var_name_a, a, var_name_b, b,
      var_name_c, a - b);
00214         write_numeric_variable(var_name_c, a - b);
00215     }
00216 }
00217
00218 void multiply(const char *var_name_a, const char *var_name_b, const char *var_name_c, bool condition)
      {
00219     if (r_trig(var_name_c, condition)) {
00220         double a = read_numeric_variable(var_name_a);
00221         double b = read_numeric_variable(var_name_b);
00222         // Log the multiply operation (commented out)
00223         // ESP_LOGI(TAG, "Multiply: %s (%f) * %s (%f) => %s (%f)", var_name_a, a, var_name_b, b,
      var_name_c, a * b);
00224         write_numeric_variable(var_name_c, a * b);
00225     }
00226 }
00227
00228 void divide(const char *var_name_a, const char *var_name_b, const char *var_name_c, bool condition) {
00229     if (r_trig(var_name_c, condition)) {
00230         double a = read_numeric_variable(var_name_a);
00231         double b = read_numeric_variable(var_name_b);
00232
00233         // Check for division by zero
00234         if (fabs(b) < 1e-6) {
00235             // Log division by zero error
00236             ESP_LOGE(TAG, "Division by zero for %s", var_name_b);
00237             return;
00238         }
00239
00240         // Log the divide operation (commented out)
00241         // ESP_LOGI(TAG, "Divide: %s (%f) / %s (%f) => %s (%f)", var_name_a, a, var_name_b, b,
      var_name_c, a / b);
00242         write_numeric_variable(var_name_c, a / b);
00243     }
00244 }
00245
00246 void move(const char *var_name_a, const char *var_name_b, bool condition) {
00247     if(r_trig(var_name_b, condition)) {
00248         double a = read_numeric_variable(var_name_a);
```

```
00249         // Log the move operation (commented out)
00250         // ESP_LOGI(TAG, "Move: %s (%f) => %s (%f)", var_name_a, a, var_name_b, a);
00251         write_numeric_variable(var_name_b, a);
00252     }
00253 }
00254
00255 // ============== COMPARE ===============
00256 bool greater(const char *var_name_a, const char *var_name_b) {
00257     double a = read_numeric_variable(var_name_a);
00258     double b = read_numeric_variable(var_name_b);
00259     // Log the greater comparison (commented out)
00260     // ESP_LOGI(TAG, "Greater: %s (%f) > %s (%f): %d", var_name_a, a, var_name_b, b, a > b);
00261     return a > b;
00262 }
00263
00264 bool less(const char *var_name_a, const char *var_name_b) {
00265     double a = read_numeric_variable(var_name_a);
00266     double b = read_numeric_variable(var_name_b);
00267     // Log the less comparison (commented out)
00268     // ESP_LOGI(TAG, "Less: %s (%f) < %s (%f): %d", var_name_a, a, var_name_b, b, a < b);
00269     return a < b;
00270 }
00271
00272 bool greater_or_equal(const char *var_name_a, const char *var_name_b) {
00273     double a = read_numeric_variable(var_name_a);
00274     double b = read_numeric_variable(var_name_b);
00275     // Log the greater or equal comparison (commented out)
00276     // ESP_LOGI(TAG, "Greater Or Equal: %s (%f) >= %s (%f): %d", var_name_a, a, var_name_b, b, a >=
      b);
00277     return a >= b;
00278 }
00279
00280 bool less_or_equal(const char *var_name_a, const char *var_name_b) {
00281     double a = read_numeric_variable(var_name_a);
00282     double b = read_numeric_variable(var_name_b);
00283     // Log the less or equal comparison (commented out)
00284     // ESP_LOGI(TAG, "Less Or Equal: %s (%f) <= %s (%f): %d", var_name_a, a, var_name_b, b, a <= b);
00285     return a <= b;
00286 }
00287
00288 bool equal(const char *var_name_a, const char *var_name_b) {
00289     double a = read_numeric_variable(var_name_a);
00290     double b = read_numeric_variable(var_name_b);
00291     // Log the equal comparison (commented out)
00292     // ESP_LOGI(TAG, "Equal: %s (%f) == %s (%f): %d", var_name_a, a, var_name_b, b, a == b);
00293     return a == b;
00294 }
00295
00296 bool not_equal(const char *var_name_a, const char *var_name_b) {
00297     double a = read_numeric_variable(var_name_a);
00298     double b = read_numeric_variable(var_name_b);
00299     // Log the not equal comparison (commented out)
00300     // ESP_LOGI(TAG, "Not Equal: %s (%f) != %s (%f): %d", var_name_a, a, var_name_b, b, a != b);
00301     return a != b;
00302 }
00303
00304 // ======= COUNTERS / TIMERS ============
00305 void count_up(const char *var_name, bool condition) {
00306     if (r_trig(var_name, condition)) {
00307         VariableNode *node = find_variable(var_name);
00308         Counter *c = (Counter *)node->data;
00309         c->cv += 1.0; // Increment CV by 1.0
00310         c->qu = (c->cv >= c->pv); // Update QU
00311         c->qd = (c->cv <= 0.0);  // Update QD
00312         // Log the counter increment (commented out)
00313         // ESP_LOGI(TAG, "Counter: %s (cv: %f) incremented", var_name, c->cv);
00314     }
00315 }
00316
00317 void count_down(const char *var_name, bool condition) {
00318     if (r_trig(var_name, condition)) {
00319         VariableNode *node = find_variable(var_name);
00320         Counter *c = (Counter *)node->data;
00321         c->cv -= 1.0; // Decrement CV by 1.0
00322         c->qu = (c->cv >= c->pv); // Update QU
00323         c->qd = (c->cv <= 0.0);  // Update QD
00324         // Log the counter decrement (commented out)
00325         // ESP_LOGI(TAG, "Counter: %s (cv: %f) decremented", var_name, c->cv);
00326     }
00327 }
00328
00329 bool timer_on(const char *var_name, bool condition) {
00330     VariableNode *node = find_variable(var_name);
00331
00332     Timer *t = (Timer *)node->data;
00333     TimerState *state = get_timer_state(var_name);
00334     if (!state) {
```

```
00335         // Log failure to get timer state
00336         ESP_LOGE(TAG, "Failed to get state for timer %s", var_name);
00337         return false;
00338     }
00339
00340     // Update input
00341     t->in = condition;
00342
00343     // If PT <= 0, timer does not run
00344     if (t->pt <= 0) {
00345         t->et = 0;
00346         t->q = false;
00347         state->running = false;
00348         // Log timer stopped due to invalid PT (commented out)
00349         // ESP_LOGI(TAG, "TON: %s PT<=0, Q=false, ET=0", var_name);
00350         return false;
00351     }
00352
00353     if (condition) {
00354         if (!state->running && !t->q) { // Start timer only if not active and Q is false
00355             state->start_time = esp_timer_get_time();
00356             state->running = true;
00357             // Log timer start (commented out)
00358             // ESP_LOGI(TAG, "TON: %s started", var_name);
00359         }
00360
00361         if (state->running) {
00362             // Calculate elapsed time
00363             int64_t current_time = esp_timer_get_time();
00364             t->et = (double)(current_time - state->start_time) / 1000.0; // Microseconds to
      milliseconds
00365             if (t->et > t->pt) {
00366                 t->et = t->pt; // Limit ET to PT
00367                 state->running = false; // Stop further counting
00368             }
00369
00370             // Check if PT is reached
00371             t->q = (t->et >= t->pt);
00372         } else {
00373             // If timer is stopped (ET >= PT), maintain Q=true and ET=PT
00374             t->et = t->pt;
00375             t->q = true;
00376         }
00377
00378         // Log timer state (commented out)
00379         // ESP_LOGI(TAG, "TON: %s IN=%d, ET=%f, Q=%d", var_name, t->in, t->et, t->q);
00380     } else {
00381         // Reset timer
00382         t->et = 0;
00383         t->q = false;
00384         state->running = false;
00385         // Log timer stop (commented out)
00386         // ESP_LOGI(TAG, "TON: %s stopped, Q=false, ET=0", var_name);
00387     }
00388
00389     return t->q;
00390 }
00391
00392 bool timer_off(const char *var_name, bool condition) {
00393     VariableNode *node = find_variable(var_name);
00394
00395     Timer *t = (Timer *)node->data;
00396     TimerState *state = get_timer_state(var_name);
00397     if (!state) {
00398         // Log failure to get timer state
00399         ESP_LOGE(TAG, "Failed to get state for timer %s", var_name);
00400         return false;
00401     }
00402
00403     // Update input
00404     t->in = condition;
00405
00406     // If PT <= 0, timer does not run
00407     if (t->pt <= 0) {
00408         t->et = 0;
00409         t->q = condition;
00410         state->running = false;
00411         // Log timer stopped due to invalid PT (commented out)
00412         // ESP_LOGI(TAG, "TOF: %s PT<=0, Q=%d, ET=0", var_name, t->q);
00413         return t->q;
00414     }
00415
00416     if (condition) {
00417         // When IN=true, Q=true and timer does not run
00418         t->q = true;
00419         t->et = 0;
00420         state->running = false;
```

```
00421         // Log timer state when input is true (commented out)
00422         // ESP_LOGI(TAG, "TOF: %s IN=true, Q=true, ET=0", var_name);
00423     } else {
00424         if (!state->running && t->q) { // Start timer only if Q is true
00425             state->start_time = esp_timer_get_time();
00426             state->running = true;
00427             // Log timer start (commented out)
00428             // ESP_LOGI(TAG, "TOF: %s started", var_name);
00429         }
00430 
00431         if (state->running) {
00432             // Calculate elapsed time
00433             int64_t current_time = esp_timer_get_time();
00434             t->et = (double)(current_time - state->start_time) / 1000.0; // Microseconds to
      milliseconds
00435             if (t->et > t->pt) {
00436                 t->et = t->pt; // Limit ET to PT
00437                 state->running = false; // Stop further counting
00438             }
00439 
00440             // Q remains true while ET < PT
00441             t->q = (t->et < t->pt);
00442         } else if (!t->q) {
00443             // If timer is not running and Q=false, maintain ET=0
00444             t->et = 0;
00445         }
00446 
00447         // Log timer state (commented out)
00448         // ESP_LOGI(TAG, "TOF: %s IN=%d, ET=%f, Q=%d", var_name, t->in, t->et, t->q);
00449     }
00450 
00451     return t->q;
00452 }
00453 
00454 void reset(const char *var_name, bool condition) {
00455     if (r_trig(var_name, condition)) {
00456         VariableNode *node = find_variable(var_name);
00457 
00458         if (node->type == VAR_TYPE_COUNTER) {
00459             Counter *c = (Counter *)node->data;
00460             bool action_taken = false;
00461 
00462             if (c->cu) {
00463                 c->cv = 0.0;
00464                 action_taken = true;
00465             }
00466             if (c->cd) {
00467                 c->cv = c->pv;
00468                 action_taken = true;
00469             }
00470 
00471             if (action_taken) {
00472                 c->qu = (c->cv >= c->pv);
00473                 c->qd = (c->cv <= 0.0);
00474             }
00475             // Log counter reset
00476             ESP_LOGI(TAG, "Counter: %s reset (cv: %f)", var_name, c->cv);
00477         } else if (node->type == VAR_TYPE_TIMER) {
00478             Timer *t = (Timer *)node->data;
00479             TimerState *state = get_timer_state(var_name);
00480             if (state) {
00481                 t->et = 0;
00482                 t->q = false;
00483                 t->in = false;
00484                 state->running = false;
00485                 // Log timer reset
00486                 ESP_LOGI(TAG, "Timer: %s reset (ET=0, Q=false, IN=false)", var_name);
00487             }
00488         }
00489     }
00490 }
```

4.21 main/ladder_elements.h File Reference

```
#include <stdbool.h>
```

Macros

- #define MAX_ONE_SHOT_STATES 64

Maximum number of variables that can detect a rising edge (for mathematical operations, coils, one-shot coils, counters, timers, reset).

- #define MAX_TIMER_STATES 32

  Maximum number of timer states.

Functions

- bool no_contact (const char *var_name)

  Normally Open (NO) Contact: Returns true (active) when the associated signal is true, otherwise false (inactive).
- bool nc_contact (const char *var_name)

  Normally Closed (NC) Contact: Returns true (active) when the associated signal is false (inactive), otherwise false (inactive).
- void coil (const char *var_name, bool condition)

  Coil: Writes the current value (true/false) to the target variable.
- void one_shot_positive_coil (const char *var_name, bool condition)

  One Shot Positive Coil: Writes true only on the rising edge (first time the signal becomes true), otherwise false.
- void set_coil (const char *var_name, bool condition)

  Set Coil: Writes true when the signal is true and retains the value until reset.
- void reset_coil (const char *var_name, bool condition)

  Reset Coil: Writes false when the signal is true and retains the value until set.
- void add (const char *var_name_a, const char *var_name_b, const char *var_name_c, bool condition)

  Add: Performs addition (A + B = C).
- void subtract (const char *var_name_a, const char *var_name_b, const char *var_name_c, bool condition)

  Subtract: Performs subtraction (A - B = C).
- void multiply (const char *var_name_a, const char *var_name_b, const char *var_name_c, bool condition)

  *Multiply: Performs multiplication (A * B = C).*
- void divide (const char *var_name_a, const char *var_name_b, const char *var_name_c, bool condition)

  Divide: Performs division (A / B = C).
- void move (const char *var_name_a, const char *var_name_b, bool condition)

  Move: Copies the value of A to B.
- bool greater (const char *var_name_a, const char *var_name_b)

  Greater: Checks if A is greater than B (A > B).
- bool less (const char *var_name_a, const char *var_name_b)

  Less: Checks if A is less than B (A < B).
- bool greater_or_equal (const char *var_name_a, const char *var_name_b)

  Greater or Equal: Checks if A is greater than or equal to B (A >= B).
- bool less_or_equal (const char *var_name_a, const char *var_name_b)

  Less or Equal: Checks if A is less than or equal to B (A <= B).
- bool equal (const char *var_name_a, const char *var_name_b)

  Equal: Checks if A is equal to B (A == B).
- bool not_equal (const char *var_name_a, const char *var_name_b)

  Not Equal: Checks if A is not equal to B (A != B).
- void count_up (const char *var_name, bool condition)

  Count Up: Increments the counter variable.
- void count_down (const char *var_name, bool condition)

  Count Down: Decrements the counter variable.
- bool timer_on (const char *var_name, bool condition)

  Timer On-Delay: Activates the timer with an on-delay mechanism.
- bool timer_off (const char *var_name, bool condition)

  Timer Off-Delay: Activates the timer with an off-delay mechanism.
- void reset (const char *var_name, bool condition)

  Reset: Resets the specified variable (e.g., counter or timer).

4.21.1 Macro Definition Documentation

4.21.1.1 MAX_ONE_SHOT_STATES

```
#define MAX_ONE_SHOT_STATES 64
```

Maximum number of variables that can detect a rising edge (for mathematical operations, coils, one-shot coils, counters, timers, reset).

Definition at line 9 of file ladder_elements.h.

4.21.1.2 MAX_TIMER_STATES

```
#define MAX_TIMER_STATES 32
```

Maximum number of timer states.

Definition at line 14 of file ladder_elements.h.

4.21.2 Function Documentation

4.21.2.1 add()

```
void add (
            const char * var_name_a,
            const char * var_name_b,
            const char * var_name_c,
            bool condition)
```

Add: Performs addition (A + B = C).

Parameters

var_name_a	Name of the first input variable.
var_name_b	Name of the second input variable.
var_name_c	Name of the output variable.
condition	Condition to enable the operation.

Definition at line 197 of file ladder_elements.c.

4.21.2.2 coil()

```
void coil (
            const char * var_name,
            bool condition)
```

Coil: Writes the current value (true/false) to the target variable.

Parameters

var_name	Name of the target variable.
condition	The condition value to write.

Definition at line 160 of file ladder_elements.c.

4.21.2.3 count_down()

```
void count_down (
            const char * var_name,
            bool condition)
```

Count Down: Decrements the counter variable.

Parameters

var_name	Name of the counter variable.
condition	Condition to enable counting.

Definition at line 317 of file ladder_elements.c.

4.21.2.4 count_up()

```
void count_up (
            const char * var_name,
            bool condition)
```

Count Up: Increments the counter variable.

Parameters

var_name	Name of the counter variable.
condition	Condition to enable counting.

Definition at line 305 of file ladder_elements.c.

4.21.2.5 divide()

```
void divide (
            const char * var_name_a,
            const char * var_name_b,
            const char * var_name_c,
            bool condition)
```

Divide: Performs division (A / B = C).

Parameters

var_name↩ _a	Name of the first input variable.
var_name↩ _b	Name of the second input variable.
var_name↩ _c	Name of the output variable.
condition	Condition to enable the operation.

Definition at line 228 of file ladder_elements.c.

4.21.2.6 equal()

```
bool equal (
            const char * var_name_a,
            const char * var_name_b)
```

Equal: Checks if A is equal to B (A == B).

Parameters

var_name↩ _a	Name of the first variable.
var_name↩ _b	Name of the second variable.

Returns

bool True if A == B, false otherwise.

Definition at line 288 of file ladder_elements.c.

4.21.2.7 greater()

```
bool greater (
            const char * var_name_a,
            const char * var_name_b)
```

Greater: Checks if A is greater than B (A $>$ B).

Parameters

var_name↩ _a	Name of the first variable.
var_name↩ _b	Name of the second variable.

Returns

bool True if A $>$ B, false otherwise.

Definition at line 256 of file ladder_elements.c.

4.21.2.8 greater_or_equal()

```
bool greater_or_equal (
            const char * var_name_a,
            const char * var_name_b)
```

Greater or Equal: Checks if A is greater than or equal to B (A >= B).

Parameters

var_name_a	Name of the first variable.
var_name_b	Name of the second variable.

Returns

bool True if A >= B, false otherwise.

Definition at line 272 of file ladder_elements.c.

4.21.2.9 less()

```
bool less (
            const char * var_name_a,
            const char * var_name_b)
```

Less: Checks if A is less than B (A < B).

Parameters

var_name_a	Name of the first variable.
var_name_b	Name of the second variable.

Returns

bool True if A < B, false otherwise.

Definition at line 264 of file ladder_elements.c.

4.21.2.10 less_or_equal()

```
bool less_or_equal (
            const char * var_name_a,
            const char * var_name_b)
```

Less or Equal: Checks if A is less than or equal to B (A <= B).

Parameters

var_name↩ _a	Name of the first variable.
var_name↩ _b	Name of the second variable.

Returns

bool True if A <= B, false otherwise.

Definition at line 280 of file ladder_elements.c.

4.21.2.11 move()

```
void move (
            const char * var_name_a,
            const char * var_name_b,
            bool condition)
```

Move: Copies the value of A to B.

Parameters

var_name↩ _a	Name of the source variable.
var_name↩ _b	Name of the destination variable.
condition	Condition to enable the operation.

Definition at line 246 of file ladder_elements.c.

4.21.2.12 multiply()

```
void multiply (
            const char * var_name_a,
            const char * var_name_b,
            const char * var_name_c,
            bool condition)
```

Multiply: Performs multiplication (A * B = C).

Parameters

var_name↩ _a	Name of the first input variable.
var_name↩ _b	Name of the second input variable.
var_name↩ _c	Name of the output variable.
condition	Condition to enable the operation.

Definition at line 218 of file ladder_elements.c.

4.21.2.13 nc_contact()

```
bool nc_contact (
            const char * var_name)
```

Normally Closed (NC) Contact: Returns true (active) when the associated signal is false (inactive), otherwise false (inactive).

Parameters

var_name	Name of the variable to check.

Returns

bool True if the signal is inactive, false otherwise.

Definition at line 152 of file ladder_elements.c.

4.21.2.14 no_contact()

```
bool no_contact (
            const char * var_name)
```

Normally Open (NO) Contact: Returns true (active) when the associated signal is true, otherwise false (inactive).

Parameters

var_name	Name of the variable to check.

Returns

bool True if the signal is active, false otherwise.

Definition at line 145 of file ladder_elements.c.

4.21.2.15 not_equal()

```
bool not_equal (
            const char * var_name_a,
            const char * var_name_b)
```

Not Equal: Checks if A is not equal to B (A != B).

Parameters

var_name_a	Name of the first variable.
var_name_b	Name of the second variable.

Returns

bool True if A != B, false otherwise.

Definition at line 296 of file ladder_elements.c.

4.21.2.16 one_shot_positive_coil()

```
void one_shot_positive_coil (
            const char * var_name,
            bool condition)
```

One Shot Positive Coil: Writes true only on the rising edge (first time the signal becomes true), otherwise false.

Parameters

var_name	Name of the target variable.
condition	The condition to evaluate for the rising edge.

Definition at line 166 of file ladder_elements.c.

4.21.2.17 reset()

```
void reset (
            const char * var_name,
            bool condition)
```

Reset: Resets the specified variable (e.g., counter or timer).

Parameters

var_name	Name of the variable to reset.
condition	Condition to trigger the reset.

Definition at line 454 of file ladder_elements.c.

4.21.2.18 reset_coil()

```
void reset_coil (
            const char * var_name,
            bool condition)
```

Reset Coil: Writes false when the signal is true and retains the value until set.

Parameters

var_name	Name of the target variable.
condition	The condition to reset the coil.

Definition at line 188 of file ladder_elements.c.

4.21.2.19 set_coil()

```
void set_coil (
            const char * var_name,
            bool condition)
```

Set Coil: Writes true when the signal is true and retains the value until reset.

Parameters

var_name	Name of the target variable.
condition	The condition to set the coil.

Definition at line 180 of file ladder_elements.c.

4.21.2.20 subtract()

```
void subtract (
            const char * var_name_a,
            const char * var_name_b,
            const char * var_name_c,
            bool condition)
```

Subtract: Performs subtraction (A - B = C).

Parameters

var_name_a	Name of the first input variable.
var_name_b	Name of the second input variable.
var_name_c	Name of the output variable.
condition	Condition to enable the operation.

Definition at line 207 of file ladder_elements.c.

4.21.2.21 timer_off()

```
bool timer_off (
            const char * var_name,
            bool condition)
```

Timer Off-Delay: Activates the timer with an off-delay mechanism.

Parameters

var_name	Name of the timer variable.
condition	Condition to start the timer.

Returns

bool True when the timer is active, false after the delay expires.

Definition at line 392 of file ladder_elements.c.

4.21.2.22 timer_on()

```
bool timer_on (
            const char * var_name,
            bool condition)
```

Timer On-Delay: Activates the timer with an on-delay mechanism.

Parameters

var_name	Name of the timer variable.
condition	Condition to start the timer.

Returns

bool True when the timer reaches its setpoint, false otherwise.

Definition at line 329 of file ladder_elements.c.

4.22 ladder_elements.h

Go to the documentation of this file.

```
00001 #ifndef LADDER_ELEMENTS_H
00002 #define LADDER_ELEMENTS_H
00003
00004 #include <stdbool.h>
00005
00009 #define MAX_ONE_SHOT_STATES 64
00010
00014 #define MAX_TIMER_STATES 32
00015
00021 bool no_contact(const char *var_name);
00022
00028 bool nc_contact(const char *var_name);
00029
00035 void coil(const char *var_name, bool condition);
00036
00042 void one_shot_positive_coil(const char *var_name, bool condition);
00043
00049 void set_coil(const char *var_name, bool condition);
00050
00056 void reset_coil(const char *var_name, bool condition);
00057
00065 void add(const char *var_name_a, const char *var_name_b, const char *var_name_c, bool condition);
00066
00074 void subtract(const char *var_name_a, const char *var_name_b, const char *var_name_c, bool condition);
00075
00083 void multiply(const char *var_name_a, const char *var_name_b, const char *var_name_c, bool condition);
00084
00092 void divide(const char *var_name_a, const char *var_name_b, const char *var_name_c, bool condition);
00093
00100 void move(const char *var_name_a, const char *var_name_b, bool condition);
00101
00108 bool greater(const char *var_name_a, const char *var_name_b);
00109
00116 bool less(const char *var_name_a, const char *var_name_b);
00117
00124 bool greater_or_equal(const char *var_name_a, const char *var_name_b);
00125
00132 bool less_or_equal(const char *var_name_a, const char *var_name_b);
00133
00140 bool equal(const char *var_name_a, const char *var_name_b);
00141
00148 bool not_equal(const char *var_name_a, const char *var_name_b);
00149
00155 void count_up(const char *var_name, bool condition);
00156
00162 void count_down(const char *var_name, bool condition);
00163
00170 bool timer_on(const char *var_name, bool condition);
00171
00178 bool timer_off(const char *var_name, bool condition);
00179
00185 void reset(const char *var_name, bool condition);
00186
00187 #endif // LADDER_ELEMENTS_H
```

4.23 main/main.c File Reference

```
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_system.h"
#include "driver/gpio.h"
#include <esp_log.h>
#include <esp_err.h>
#include "wifi.h"
#include "mqtt.h"
#include "nvs_utils.h"
#include "variables.h"
#include "one_wire_detect.h"
#include "conf_task_manager.h"
#include "ble.h"
```

Macros

- #define GPIO18_OUTPUT_PIN 18

  GPIO pin used for output on this specific device. This pin (GPIO18) is configured for output and is specific to this device; it is not required for all devices.

Functions

- void app_main (void)

  Main application entry point. Initializes hardware, network, and communication modules, then enters an infinite loop to handle periodic tasks such as sending variables and publishing sensor data.

Variables

- static const char * TAG = "filip_device"

  Tag for logging messages from the main application module.

4.23.1 Macro Definition Documentation

4.23.1.1 GPIO18_OUTPUT_PIN

```
#define GPIO18_OUTPUT_PIN 18
```

GPIO pin used for output on this specific device. This pin (GPIO18) is configured for output and is specific to this device; it is not required for all devices.

Definition at line 23 of file main.c.

4.23.2 Function Documentation

4.23.2.1 app_main()

```
void app_main (
            void )
```

Main application entry point. Initializes hardware, network, and communication modules, then enters an infinite loop to handle periodic tasks such as sending variables and publishing sensor data.

Definition at line 35 of file main.c.

4.23.3 Variable Documentation

4.23.3.1 TAG

```
const char* TAG = "filip_device"  [static]
```

Tag for logging messages from the main application module.

Definition at line 28 of file main.c.

4.24 main.c

Go to the documentation of this file.

```
00001 #include "freertos/FreeRTOS.h"
00002 #include "freertos/task.h"
00003 #include "esp_system.h"
00004 #include "driver/gpio.h"
00005 #include <esp_log.h>
00006 #include <esp_err.h>
00007
00008 #include "wifi.h"
00009 #include "mqtt.h"
00010 #include "nvs_utils.h"
00011
00012 #include "variables.h"
00013
00014 #include "one_wire_detect.h"
00015 #include "conf_task_manager.h"
00016
00017 #include "ble.h"
00018
00023 #define GPIO18_OUTPUT_PIN 18
00024
00028 static const char *TAG = "filip_device";
00029
00035 void app_main(void)
00036 {
00037     // Initialize GPIO18 for output (specific to this device, not required for all devices)
00038     gpio_reset_pin(GPIO18_OUTPUT_PIN);
00039     gpio_set_direction(GPIO18_OUTPUT_PIN, GPIO_MODE_OUTPUT);
00040     gpio_set_level(GPIO18_OUTPUT_PIN, 1);
00041
00042     // Initialize Non-Volatile Storage (NVS) for Wi-Fi and configuration
00043     esp_err_t ret = nvs_init();
00044     if (ret != ESP_OK) {
00045         ESP_LOGE(TAG, "Failed to initialize NVS, halting...");
00046         return;
00047     }
00048
00049     // Load configuration from NVS
00050     char *nvs_data = NULL;
00051     size_t nvs_data_len = 0;
00052     ret = load_config_from_nvs(&nvs_data, &nvs_data_len);
00053     if (ret == ESP_OK && nvs_data != NULL) {
```

```
00054         configure(nvs_data, nvs_data_len, true); // Apply loaded configuration
00055         free(nvs_data);                          // Free allocated memory
00056     }
00057
00058     // Initialize Wi-Fi (includes NTP and MQTT initialization within wifi.c)
00059     wifi_init();
00060
00061     // Initialize Bluetooth Low Energy (BLE)
00062     ble_init();
00063
00064     // Counter for periodic tasks
00065     // int cnt = 0;
00066     while(1){
00067         // cnt++;
00068         // if (cnt == 50){
00069         //     // Log free heap size every 50 iterations (commented out)
00070         //     printf("Heap %lu bytes\n", esp_get_free_heap_size());
00071         //     cnt = 0; // Reset counter
00072         // }
00073
00074         // Send variables to parent devices if MQTT is connected
00075         if (mqtt_is_connected())
00076             send_variables_to_parents();
00077
00078         // Publish sensor data if the application is connected via MQTT
00079         if(app_connected_mqtt) {
00080             char *monitor_json = read_variables_json(); // Read variables as JSON
00081             if (monitor_json) {
00082                 mqtt_publish(monitor_json, topics[TOPIC_IDX_MONITOR], MQTT_QOS); // Publish variables
00083                 free(monitor_json); // Free allocated memory
00084             }
00085             char *one_wire_json = search_for_one_wire_sensors(); // Read one-wire sensor data
00086             if (one_wire_json) {
00087                 mqtt_publish(one_wire_json, topics[TOPIC_IDX_ONE_WIRE], MQTT_QOS); // Publish sensor
    data
00088                 free(one_wire_json); // Free allocated memory
00089             }
00090         } else if (app_connected_ble) {
00091             // Placeholder for BLE-specific functionality (currently empty)
00092         }
00093
00094         // Delay for 100ms before the next iteration
00095         vTaskDelay(pdMS_TO_TICKS(100));
00096     }
00097 }
```

4.25 main/mqtt.c File Reference

```
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "mqtt_client.h"
#include "mqtt.h"
#include "nvs_utils.h"
#include "one_wire_detect.h"
#include "esp_mac.h"
#include <stdio.h>
#include "esp_system.h"
#include "esp_wifi.h"
#include "conf_task_manager.h"
#include "variables.h"
```

Functions

- static void connection_timeout_task (void *pvParameters)

  Task to monitor the timeout for "Present" messages.
- static void mqtt_event_handler (void *arg, esp_event_base_t event_base, int32_t event_id, void *event_data)

Handles MQTT events such as connection, disconnection, and data reception.

- void mqtt_init ()

  Initializes the MQTT client and sets up communication with the broker.
- void mqtt_publish (const char *message, const char *topic, int qos)

  Publishes a message to the specified MQTT topic.
- bool mqtt_is_connected (void)

  Checks if the MQTT client is connected to the broker.

Variables

- static const char * TAG = "MQTT_MODULE"

  Tag for logging messages from the MQTT module.
- static bool mqtt_connected = false

  Flag indicating whether the MQTT client is connected to the broker.
- esp_mqtt_client_handle_t mqtt_client

  Handle for the MQTT client.
- bool app_connected_mqtt = false

  Flag indicating whether the application is connected via MQTT.
- static TickType_t last_present_time = 0

  Timestamp of the last received "Present" message.
- static TaskHandle_t connection_timeout_task_handle = NULL

  Handle for the connection timeout task.
- static char mac_str [13]

  String to store the device's MAC address as a 12-character hexadecimal string.
- char topics [8][MAX_TOPIC_LEN]

  Array to store MQTT topic strings, each with a maximum length of MAX_TOPIC_LEN.

4.25.1 Function Documentation

4.25.1.1 connection_timeout_task()

```
static void connection_timeout_task (
            void * pvParameters)  [static]
```

Task to monitor the timeout for "Present" messages.

Parameters

pvParameters	Unused task parameter.

Definition at line 63 of file mqtt.c.

4.25.1.2 mqtt_event_handler()

```
static void mqtt_event_handler (
            void * arg,
            esp_event_base_t event_base,
            int32_t event_id,
            void * event_data)  [static]
```

Handles MQTT events such as connection, disconnection, and data reception.

Parameters

arg	Unused argument.
event_base	Event base identifier.
event_id	Specific event identifier.
event_data	Pointer to event data.

Definition at line 85 of file mqtt.c.

4.25.1.3 mqtt_init()

```
void mqtt_init (
            void )
```

Initializes the MQTT client and sets up communication with the broker.

Definition at line 192 of file mqtt.c.

4.25.1.4 mqtt_is_connected()

```
bool mqtt_is_connected (
            void )
```

Checks if the MQTT client is connected to the broker.

Returns

bool True if connected, false otherwise.

Definition at line 236 of file mqtt.c.

4.25.1.5 mqtt_publish()

```
void mqtt_publish (
            const char * topic,
            const char * message,
            int qos)
```

Publishes a message to the specified MQTT topic.

Parameters

topic	The MQTT topic to publish to.
message	The message to publish.
qos	Quality of Service level for the message.

Definition at line 230 of file mqtt.c.

4.25.2 Variable Documentation

4.25.2.1 app_connected_mqtt

```
bool app_connected_mqtt = false
```

Flag indicating whether the application is connected via MQTT.

Flag indicating whether the application is connected to the MQTT broker.

Definition at line 37 of file mqtt.c.

4.25.2.2 connection_timeout_task_handle

```
TaskHandle_t connection_timeout_task_handle = NULL  [static]
```

Handle for the connection timeout task.

Definition at line 47 of file mqtt.c.

4.25.2.3 last_present_time

```
TickType_t last_present_time = 0  [static]
```

Timestamp of the last received "Present" message.

Definition at line 42 of file mqtt.c.

4.25.2.4 mac_str

```
char mac_str[13]  [static]
```

String to store the device's MAC address as a 12-character hexadecimal string.

Definition at line 52 of file mqtt.c.

4.25.2.5 mqtt_client

```
esp_mqtt_client_handle_t mqtt_client
```

Handle for the MQTT client.

Definition at line 32 of file mqtt.c.

4.25.2.6 mqtt_connected

```
bool mqtt_connected = false  [static]
```

Flag indicating whether the MQTT client is connected to the broker.

Definition at line 27 of file mqtt.c.

4.25.2.7 TAG

```
const char* TAG = "MQTT_MODULE"  [static]
```

Tag for logging messages from the MQTT module.

Definition at line 22 of file mqtt.c.

4.25.2.8 topics

```
char topics[8][MAX_TOPIC_LEN]
```

Array to store MQTT topic strings, each with a maximum length of MAX_TOPIC_LEN.

External array to store MQTT topic strings.

Definition at line 57 of file mqtt.c.

4.26 mqtt.c

Go to the documentation of this file.

```
00001 #include <string.h>
00002 #include "freertos/FreeRTOS.h"
00003 #include "freertos/task.h"
00004 #include "esp_log.h"
00005 #include "mqtt_client.h"
00006 #include "mqtt.h"
00007
00008 #include "nvs_utils.h"
00009 #include "one_wire_detect.h"
00010
00011 #include "esp_mac.h"
00012
00013 #include <stdio.h>
00014 #include "esp_system.h"
00015 #include "esp_wifi.h"
00016 #include "conf_task_manager.h"
00017 #include "variables.h"
00018
00022 static const char *TAG = "MQTT_MODULE";
00023
00027 static bool mqtt_connected = false;
00028
00032 esp_mqtt_client_handle_t mqtt_client;
00033
00037 bool app_connected_mqtt = false;
00038
00042 static TickType_t last_present_time = 0; // Time of the last "Present" message
00043
00047 static TaskHandle_t connection_timeout_task_handle = NULL; // Handle for the task
00048
00052 static char mac_str[13];
00053
00057 char topics[8][MAX_TOPIC_LEN]; // Array for all topics
00058
00063 static void connection_timeout_task(void *pvParameters) {
00064     while (1) {
00065         if (app_connected_mqtt && (xTaskGetTickCount() - last_present_time > pdMS_TO_TICKS(10000))) {
00066             ESP_LOGI(TAG, "No 'Present' message received for 10 seconds, disconnecting app");
00067             app_connected_mqtt = false;
00068             mqtt_publish("Disconnected", topics[TOPIC_IDX_CONNECTION_RESPONSE], MQTT_QOS);
00069             // Delete the task as the application is no longer connected
00070             connection_timeout_task_handle = NULL;
00071             vTaskDelete(NULL);
00072         }
00073         // Check every second
00074         vTaskDelay(pdMS_TO_TICKS(1000)); // Check every second
00075     }
00076 }
00077
```

```
00085 static void mqtt_event_handler(void *arg, esp_event_base_t event_base, int32_t event_id, void
      *event_data) {
00086     esp_mqtt_event_handle_t event = event_data;
00087     switch (event_id) {
00088         case MQTT_EVENT_CONNECTED:
00089             // Handle successful connection to the MQTT broker
00090             ESP_LOGI(TAG, "MQTT Connected to broker");
00091             mqtt_connected = true;
00092             // Subscribe to relevant topics
00093             esp_mqtt_client_subscribe(mqtt_client, topics[TOPIC_IDX_CONNECTION_REQUEST], MQTT_QOS); //
      Application requests connection with the device
00094             esp_mqtt_client_subscribe(mqtt_client, topics[TOPIC_IDX_CONFIG_REQUEST], MQTT_QOS);     //
      Application requests configuration from the device
00095             esp_mqtt_client_subscribe(mqtt_client, topics[TOPIC_IDX_CONFIG_RECEIVE], MQTT_QOS);     //
      Application sends configuration to the device
00096             esp_mqtt_client_subscribe(mqtt_client, topics[TOPIC_IDX_CHILDREN_LISTENER], MQTT_QOS);  //
      Application sends configuration to the device
00097             break;
00098         case MQTT_EVENT_DISCONNECTED:
00099             // Handle disconnection from the MQTT broker
00100             ESP_LOGI(TAG, "MQTT Disconnected");
00101             mqtt_connected = false;
00102             app_connected_mqtt = false;
00103             // Delete the task if it exists
00104             if (connection_timeout_task_handle != NULL) {
00105                 vTaskDelete(connection_timeout_task_handle);
00106                 connection_timeout_task_handle = NULL;
00107             }
00108             break;
00109         case MQTT_EVENT_SUBSCRIBED:
00110             // Log successful subscription to a topic
00111             ESP_LOGI(TAG, "Subscribed to topic");
00112             break;
00113         case MQTT_EVENT_UNSUBSCRIBED:
00114             // Log successful unsubscription from a topic
00115             ESP_LOGI(TAG, "Unsubscribed from topic");
00116             break;
00117         case MQTT_EVENT_DATA:
00118             // Handle incoming MQTT data
00119             // Validate topic before processing
00120             if (event->topic == NULL || event->topic_len == 0) {
00121                 ESP_LOGE(TAG, "Received MQTT message with invalid topic (NULL or empty)");
00122                 break;
00123             }
00124             // Application requests connection with the device
00125             else if (strncmp(event->topic, topics[TOPIC_IDX_CONNECTION_REQUEST], event->topic_len) ==
      0)
00126             {
00127                 if (strncmp(event->data, "Present", event->data_len) == 0) {
00128                     // Update last presence timestamp for "Present" message
00129                     last_present_time = xTaskGetTickCount(); // Update presence time
00130                 }
00131                 else if(app_connected_mqtt && strncmp(event->data, "Disconnect", event->data_len) ==
      0){
00132                     // Handle app disconnection
00133                     ESP_LOGI(TAG, "App disconnected");
00134                     app_connected_mqtt = false;
00135                     // Delete the task if it exists
00136                     if (connection_timeout_task_handle != NULL) {
00137                         vTaskDelete(connection_timeout_task_handle);
00138                         connection_timeout_task_handle = NULL;
00139                     }
00140                 }
00141                 else if (!app_connected_mqtt && strncmp(event->data, "Connect", event->data_len) ==
      0){
00142                     // Handle app connection
00143                     ESP_LOGI(TAG, "App connected");
00144                     app_connected_mqtt = true;
00145                     last_present_time = xTaskGetTickCount(); // Update presence time
00146                     mqtt_publish("Connected", topics[TOPIC_IDX_CONNECTION_RESPONSE], MQTT_QOS);
00147                     // Create task to monitor connection timeout
00148                     if (xTaskCreate(connection_timeout_task, "connection_timeout_task", 2048, NULL, 5,
      &connection_timeout_task_handle) != pdPASS) {
00149                         ESP_LOGE(TAG, "Failed to create connection timeout task");
00150                     }
00151                 }
00152             }
00153             // Application requests configuration from the device
00154             else if (strncmp(event->topic, topics[TOPIC_IDX_CONFIG_REQUEST], event->topic_len) == 0 &&
      app_connected_mqtt)
00155             {
00156                 ESP_LOGI(TAG, "Configuration requested");
00157                 char *nvs_data = NULL;
00158                 size_t nvs_data_len = 0;
00159                 // Load configuration from NVS
00160                 esp_err_t ret = load_config_from_nvs(&nvs_data, &nvs_data_len);
00161                 if (ret == ESP_OK && nvs_data != NULL) {
```

```
00162                     // Publish configuration to response topic
00163                     mqtt_publish(nvs_data, topics[TOPIC_IDX_CONFIG_RESPONSE], MQTT_QOS);
00164                     free(nvs_data);
00165                     ESP_LOGI(TAG, "Configuration sent successfully");
00166                 } else {
00167                     ESP_LOGE(TAG, "Configuration sent unsuccessfully");
00168                 }
00169             }
00170             // Application sends configuration to the device
00171             else if (strncmp(event->topic, topics[TOPIC_IDX_CONFIG_RECEIVE], event->topic_len) == 0)
00172             {
00173                 // Process received configuration
00174                 configure(event->data, event->data_len, false);
00175             }
00176             // Update variables based on information received from remote devices
00177             else if (strncmp(event->topic, topics[TOPIC_IDX_CHILDREN_LISTENER], event->topic_len) ==
      0)
00178             {
00179                 // Update variables based on data from remote devices
00180                 update_variables_from_children(event->data);
00181             }
00182             break;
00183         case MQTT_EVENT_ERROR:
00184             // Log MQTT error
00185             ESP_LOGE(TAG, "MQTT Error");
00186             break;
00187         default:
00188             break;
00189     }
00190 }
00191
00192 void mqtt_init() {
00193     // Configure MQTT client with broker URI
00194     esp_mqtt_client_config_t mqtt_cfg = {
00195         .broker.address.uri = MQTT_BROKER_URI,
00196     };
00197     mqtt_client = esp_mqtt_client_init(&mqtt_cfg);
00198     // Register event handler for MQTT events
00199     esp_mqtt_client_register_event(mqtt_client, ESP_EVENT_ANY_ID, mqtt_event_handler, NULL);
00200     // Start the MQTT client
00201     esp_mqtt_client_start(mqtt_client);
00202
00203     // Initialize app connection state
00204     app_connected_mqtt = false;
00205
00206     // Get MAC address
00207     uint8_t mac[6];
00208     esp_read_mac(mac, ESP_MAC_WIFI_STA);
00209     snprintf(mac_str, sizeof(mac_str), "%02X%02X%02X%02X%02X%02X", mac[0], mac[1], mac[2], mac[3],
      mac[4], mac[5]);
00210
00211     // Initialize topics with MAC prefix
00212     const char *suffixes[] = {
00213         TOPIC_CONNECTION_REQUEST,
00214         TOPIC_CONNECTION_RESPONSE,
00215         TOPIC_MONITOR,
00216         TOPIC_ONE_WIRE,
00217         TOPIC_CONFIG_REQUEST,
00218         TOPIC_CONFIG_RESPONSE,
00219         TOPIC_CONFIG_RECEIVE,
00220         TOPIC_CHILDREN_LISTENER,
00221     };
00222     for (int i = 0; i < 8; i++) {
00223         snprintf(topics[i], MAX_TOPIC_LEN, "%s%s", mac_str, suffixes[i]);
00224     }
00225
00226     // Log the device's MAC address
00227     ESP_LOGI(TAG, "MAC Address: %s", mac_str);
00228 }
00229
00230 void mqtt_publish(const char *message, const char* topic, int qos) {
00231     // Publish message if connected to the broker
00232     if (mqtt_connected)
00233         esp_mqtt_client_publish(mqtt_client, topic, message, 0, qos, 0);
00234 }
00235
00236 bool mqtt_is_connected(void) {
00237     return mqtt_connected;
00238 }
```

4.27 main/mqtt.h File Reference

```
#include "esp_system.h"
#include "config.h"
```

Macros

- #define TOPIC_CONNECTION_REQUEST "/connection_request"

  MQTT broker URI, defined in config.h.
- #define TOPIC_CONNECTION_RESPONSE "/connection_response"

  Suffix for connection response topic.
- #define TOPIC_MONITOR "/monitor"

  Suffix for monitoring topic.
- #define TOPIC_ONE_WIRE "/one_wire"

  Suffix for one-wire sensor data topic.
- #define TOPIC_CONFIG_REQUEST "/config_request"

  Suffix for configuration request topic.
- #define TOPIC_CONFIG_RESPONSE "/config_response"

  Suffix for configuration response topic.
- #define TOPIC_CONFIG_RECEIVE "/config_device"

  Suffix for receiving configuration topic.
- #define TOPIC_CHILDREN_LISTENER "/children_listener"

  Suffix for children listener topic.
- #define MAX_TOPIC_LEN 35

  Maximum length of an MQTT topic string, including null terminator.
- #define MQTT_QOS 1

  Quality of Service level for MQTT messages.

Enumerations

- enum {
  TOPIC_IDX_CONNECTION_REQUEST , TOPIC_IDX_CONNECTION_RESPONSE , TOPIC_IDX_MONITOR , TOPIC_IDX_ONE_WIRE ,
  TOPIC_IDX_CONFIG_REQUEST , TOPIC_IDX_CONFIG_RESPONSE , TOPIC_IDX_CONFIG_RECEIVE , TOPIC_IDX_CHILDREN_LISTENER }

  Enumeration of topic indices for accessing the topics array.

Functions

- void mqtt_init (void)

  Initializes the MQTT client and sets up communication with the broker.
- void mqtt_publish (const char *topic, const char *message, int qos)

  Publishes a message to the specified MQTT topic.
- bool mqtt_is_connected (void)

  Checks if the MQTT client is connected to the broker.

Variables

- char topics [8][MAX_TOPIC_LEN]

  External array to store MQTT topic strings.
- bool app_connected_mqtt

  Flag indicating whether the application is connected to the MQTT broker.

4.27.1 Macro Definition Documentation

4.27.1.1 MAX_TOPIC_LEN

```
#define MAX_TOPIC_LEN 35
```

Maximum length of an MQTT topic string, including null terminator.

Definition at line 32 of file mqtt.h.

4.27.1.2 MQTT_QOS

```
#define MQTT_QOS 1
```

Quality of Service level for MQTT messages.

Definition at line 37 of file mqtt.h.

4.27.1.3 TOPIC_CHILDREN_LISTENER

```
#define TOPIC_CHILDREN_LISTENER "/children_listener"
```

Suffix for children listener topic.

Definition at line 27 of file mqtt.h.

4.27.1.4 TOPIC_CONFIG_RECEIVE

```
#define TOPIC_CONFIG_RECEIVE "/config_device"
```

Suffix for receiving configuration topic.

Definition at line 25 of file mqtt.h.

4.27.1.5 TOPIC_CONFIG_REQUEST

```
#define TOPIC_CONFIG_REQUEST "/config_request"
```

Suffix for configuration request topic.

Definition at line 23 of file mqtt.h.

4.27.1.6 TOPIC_CONFIG_RESPONSE

```
#define TOPIC_CONFIG_RESPONSE "/config_response"
```

Suffix for configuration response topic.

Definition at line 24 of file mqtt.h.

4.27.1.7 TOPIC_CONNECTION_REQUEST

```
#define TOPIC_CONNECTION_REQUEST "/connection_request"
```

MQTT broker URI, defined in config.h.

Note

Expected to be defined as MQTT_BROKER_URI in config.h.

Topic suffix definitions for MQTT communication. Suffix for connection request topic.

Definition at line 19 of file mqtt.h.

4.27.1.8 TOPIC_CONNECTION_RESPONSE

```
#define TOPIC_CONNECTION_RESPONSE "/connection_response"
```

Suffix for connection response topic.

Definition at line 20 of file mqtt.h.

4.27.1.9 TOPIC_MONITOR

```
#define TOPIC_MONITOR "/monitor"
```

Suffix for monitoring topic.

Definition at line 21 of file mqtt.h.

4.27.1.10 TOPIC_ONE_WIRE

```
#define TOPIC_ONE_WIRE "/one_wire"
```

Suffix for one-wire sensor data topic.

Definition at line 22 of file mqtt.h.

4.27.2 Enumeration Type Documentation

4.27.2.1 anonymous enum

```
anonymous enum
```

Enumeration of topic indices for accessing the topics array.

Enumerator

TOPIC_IDX_CONNECTION_REQUEST	Index for connection request topic.
TOPIC_IDX_CONNECTION_RESPONSE	Index for connection response topic.
TOPIC_IDX_MONITOR	Index for monitoring topic.
TOPIC_IDX_ONE_WIRE	Index for one-wire sensor data topic.
TOPIC_IDX_CONFIG_REQUEST	Index for configuration request topic.
TOPIC_IDX_CONFIG_RESPONSE	Index for configuration response topic.
TOPIC_IDX_CONFIG_RECEIVE	Index for configuration receive topic.
TOPIC_IDX_CHILDREN_LISTENER	Index for children listener topic.

Definition at line 42 of file mqtt.h.

4.27.3 Function Documentation

4.27.3.1 mqtt_init()

```
void mqtt_init (
            void )
```

Initializes the MQTT client and sets up communication with the broker.

Definition at line 192 of file mqtt.c.

4.27.3.2 mqtt_is_connected()

```
bool mqtt_is_connected (
            void )
```

Checks if the MQTT client is connected to the broker.

Returns

bool True if connected, false otherwise.

Definition at line 236 of file mqtt.c.

4.27.3.3 mqtt_publish()

```
void mqtt_publish (
            const char * topic,
            const char * message,
            int qos)
```

Publishes a message to the specified MQTT topic.

Parameters

topic	The MQTT topic to publish to.
message	The message to publish.
qos	Quality of Service level for the message.

Definition at line 230 of file mqtt.c.

4.27.4 Variable Documentation

4.27.4.1 app_connected_mqtt

```
bool app_connected_mqtt  [extern]
```

Flag indicating whether the application is connected to the MQTT broker.

Flag indicating whether the application is connected to the MQTT broker.

Definition at line 37 of file mqtt.c.

4.27.4.2 topics

```
char topics[8][MAX_TOPIC_LEN]  [extern]
```

External array to store MQTT topic strings.

Note

Array of 8 topics, each with a maximum length of MAX_TOPIC_LEN.

External array to store MQTT topic strings.

Definition at line 57 of file mqtt.c.

4.28 mqtt.h

Go to the documentation of this file.

```
00001 #ifndef MQTT_H
00002 #define MQTT_H
00003
00004 #include "esp_system.h"
00005
00006 #include "config.h"
00007
00012 // MQTT BROKER
00013 // Defined in config.h
00014 // #define MQTT_BROKER_URI
00015
00019 #define TOPIC_CONNECTION_REQUEST "/connection_request"
00020 #define TOPIC_CONNECTION_RESPONSE "/connection_response"
00021 #define TOPIC_MONITOR "/monitor"
00022 #define TOPIC_ONE_WIRE "/one_wire"
00023 #define TOPIC_CONFIG_REQUEST "/config_request"
00024 #define TOPIC_CONFIG_RESPONSE "/config_response"
00025 #define TOPIC_CONFIG_RECEIVE "/config_device"
```

```
00026
00027 #define TOPIC_CHILDREN_LISTENER "/children_listener"
00028
00032 #define MAX_TOPIC_LEN 35
00033
00037 #define MQTT_QOS 1
00038
00042 enum {
00043     TOPIC_IDX_CONNECTION_REQUEST,
00044     TOPIC_IDX_CONNECTION_RESPONSE,
00045     TOPIC_IDX_MONITOR,
00046     TOPIC_IDX_ONE_WIRE,
00047     TOPIC_IDX_CONFIG_REQUEST,
00048     TOPIC_IDX_CONFIG_RESPONSE,
00049     TOPIC_IDX_CONFIG_RECEIVE,
00050     TOPIC_IDX_CHILDREN_LISTENER,
00051 };
00052
00057 extern char topics[8][MAX_TOPIC_LEN];
00058
00062 extern bool app_connected_mqtt;
00063
00067 void mqtt_init(void);
00068
00075 void mqtt_publish(const char *topic, const char *message, int qos);
00076
00081 bool mqtt_is_connected(void);
00082
00083 #endif // MQTT_H
```

4.29 main/ntp.c File Reference

```
#include "ntp.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_log.h"
#include "variables.h"
```

Functions

- bool is_ntp_sync (void)

  Checks if the system time is synchronized with an NTP server.
- void time_sync_notification_cb (struct timeval *tv)

  Callback function triggered on successful NTP time synchronization.
- static void clock_task (void *arg)

  Task to continuously update the current time and date.
- void obtain_time (void)

  Initializes NTP client, synchronizes time, and starts the clock task.

Variables

- static const char * TAG = "NTP"

  Tag for logging messages from the NTP module.
- int hour

  Global variables to store the current time and date.
- int minute

  Current minute (0-59).
- int second

  Current second (0-59).

- int day
  Current day of the month (1-31).
- int month
  Current month (1-12).
- int year
  Current year (e.g., 2025).
- int day_in_year
  Current day of the year (1-366).
- time_t now
  Current time in seconds since epoch.
- struct tm timeinfo
  Structure to hold broken-down time information.
- bool ntp_sync = false
  Flag indicating whether NTP synchronization is complete.

4.29.1 Function Documentation

4.29.1.1 clock_task()

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

Task to continuously update the current time and date.

Parameters

arg	Unused task parameter.

Definition at line 67 of file ntp.c.

4.29.1.2 is_ntp_sync()

```
bool is_ntp_sync (
            void )
```

Checks if the system time is synchronized with an NTP server.

Returns

bool True if NTP synchronization is complete, false otherwise.

Definition at line 37 of file ntp.c.

4.29.1.3 obtain_time()

```
void obtain_time (
            void )
```

Initializes NTP client, synchronizes time, and starts the clock task.

Initializes NTP client and synchronizes system time with an NTP server.

Definition at line 99 of file ntp.c.

4.29.1.4 time_sync_notification_cb()

```
void time_sync_notification_cb (
             struct timeval * tv)
```

Callback function triggered on successful NTP time synchronization.

Parameters

tv	Pointer to the synchronized time value.

Definition at line 46 of file ntp.c.

4.29.2 Variable Documentation

4.29.2.1 day

```
int day
```

Current day of the month (1-31).

Definition at line 16 of file ntp.c.

4.29.2.2 day_in_year

```
int day_in_year
```

Current day of the year (1-366).

Definition at line 16 of file ntp.c.

4.29.2.3 hour

```
int hour
```

Global variables to store the current time and date.

Current hour (0-23).

Definition at line 16 of file ntp.c.

4.29.2.4 minute

```
int minute
```

Current minute (0-59).

Definition at line 16 of file ntp.c.

4.29.2.5 month

```
int month
```

Current month (1-12).

Definition at line 16 of file ntp.c.

4.29.2.6 now

```
time_t now
```

Current time in seconds since epoch.

Definition at line 21 of file ntp.c.

4.29.2.7 ntp_sync

```
bool ntp_sync = false
```

Flag indicating whether NTP synchronization is complete.

Definition at line 31 of file ntp.c.

4.29.2.8 second

```
int second
```

Current second (0-59).

Definition at line 16 of file ntp.c.

4.29.2.9 TAG

```
const char* TAG = "NTP"  [static]
```

Tag for logging messages from the NTP module.

Definition at line 11 of file ntp.c.

4.29.2.10 timeinfo

```
struct tm timeinfo
```

Structure to hold broken-down time information.

Definition at line 26 of file ntp.c.

4.29.2.11 year

```
int year
```

Current year (e.g., 2025).

Definition at line 16 of file ntp.c.

4.30 ntp.c

Go to the documentation of this file.

```
00001 #include "ntp.h"
00002 #include "freertos/FreeRTOS.h"
00003 #include "freertos/task.h"
00004 #include "esp_log.h"
00005
00006 #include "variables.h"
00007
00011 static const char *TAG = "NTP";
00012
00016 int hour, minute, second, day, month, year, day_in_year;
00017
00021 time_t now;
00022
00026 struct tm timeinfo;
00027
00031 bool ntp_sync = false;
00032
00037 bool is_ntp_sync(void)
00038 {
00039     return ntp_sync;
00040 }
00041
00046 void time_sync_notification_cb(struct timeval *tv)
00047 {
00048     ESP_LOGI(TAG, "Notification of a time synchronization event");
00049     ntp_sync = true;
00050
00051     // Set timezone to Central European Time with daylight saving rules
00052     setenv("TZ", "CET-1CEST,M3.5.0/2,M10.5.0/3", 1);
00053     tzset();
00054
00055     // Log the current local time
00056     char strftime_buf[64];
00057     time(&now);
00058     localtime_r(&now, &timeinfo);
00059     strftime(strftime_buf, sizeof(strftime_buf), "%H:%M:%S %d.%m.%Y.", &timeinfo);
00060     ESP_LOGI(TAG, "Current Time: %s", strftime_buf);
00061 }
00062
00067 static void clock_task(void *arg)
00068 {
00069     while (1)
00070     {
00071         // Update current time
00072         time(&now);
00073         localtime_r(&now, &timeinfo);
00074
00075         // Update global time and date variables
00076         hour = timeinfo.tm_hour;
00077         minute = timeinfo.tm_min;
00078         second = timeinfo.tm_sec;
00079         day = timeinfo.tm_mday;
00080         month = timeinfo.tm_mon + 1;
00081         year = timeinfo.tm_year + 1900;
00082         day_in_year = timeinfo.tm_yday + 1;
00083
00084         // Update the Current Time variable if it exists
00085         VariableNode *node = find_current_time_variable();
00086         if(node){
00087             Time *t = (Time *)node->data;
00088             t->value = hour * 10000 + minute * 100 + second;
00089         }
00090
00091         // Delay for 1 second
00092         vTaskDelay(pdMS_TO_TICKS(1000));
00093     }
00094 }
00095
00099 void obtain_time(void)
00100 {
00101     ESP_LOGI(TAG, "Initializing and starting SNTP");
00102
00103     // Configure SNTP with default settings and specify NTP server
00104     esp_sntp_config_t config = ESP_NETIF_SNTP_DEFAULT_CONFIG("pool.ntp.org");
00105
00106     // Set callback for time synchronization
00107     config.sync_cb = time_sync_notification_cb;
00108     esp_netif_sntp_init(&config);
00109
00110     // Wait for time synchronization
00111     time_t now = 0;
00112     struct tm timeinfo = {0};
```

```
00113     int retry = 0;
00114     const int retry_count = 100;
00115     while (esp_netif_sntp_sync_wait(2000 / portTICK_PERIOD_MS) == ESP_ERR_TIMEOUT && ++retry <
      retry_count)
00116     {
00117         ESP_LOGI(TAG, "Waiting for system time to be set... (%d/%d)", retry, retry_count);
00118     }
00119     time(&now);
00120     localtime_r(&now, &timeinfo);
00121     esp_netif_sntp_deinit();
00122
00123     // Create clock task to continuously update time
00124     xTaskCreate(clock_task, "clock", 1024 * 2, NULL, 10, NULL);
00125 }
```

4.31 main/ntp.h File Reference

```
#include <sys/time.h>
#include "esp_netif_sntp.h"
#include "esp_sntp.h"
```

Functions

- void obtain_time (void)

  Initializes NTP client and synchronizes system time with an NTP server.
- bool is_ntp_sync (void)

  Checks if the system time is synchronized with an NTP server.

Variables

- int hour

  Global variables to store the current time and date.
- int minute

  Current minute (0-59).
- int second

  Current second (0-59).
- int day

  Current day of the month (1-31).
- int month

  Current month (1-12).
- int year

  Current year (e.g., 2025).
- int day_in_year

  Current day of the year (1-366).

4.31.1 Function Documentation

4.31.1.1 is_ntp_sync()

```
bool is_ntp_sync (
            void )
```

Checks if the system time is synchronized with an NTP server.

Returns

bool True if NTP synchronization is complete, false otherwise.

Definition at line 37 of file ntp.c.

4.31.1.2 obtain_time()

```
void obtain_time (
            void )
```

Initializes NTP client and synchronizes system time with an NTP server.

Initializes NTP client and synchronizes system time with an NTP server.

Definition at line 99 of file ntp.c.

4.31.2 Variable Documentation

4.31.2.1 day

```
int day  [extern]
```

Current day of the month (1-31).

Definition at line 16 of file ntp.c.

4.31.2.2 day_in_year

```
int day_in_year  [extern]
```

Current day of the year (1-366).

Definition at line 16 of file ntp.c.

4.31.2.3 hour

```
int hour  [extern]
```

Global variables to store the current time and date.

Current hour (0-23).

Definition at line 16 of file ntp.c.

4.31.2.4 minute

```
int minute  [extern]
```

Current minute (0-59).

Definition at line 16 of file ntp.c.

4.31.2.5 month

```
int month  [extern]
```

Current month (1-12).

Definition at line 16 of file ntp.c.

4.31.2.6 second

```
int second  [extern]
```

Current second (0-59).

Definition at line 16 of file ntp.c.

4.31.2.7 year

```
int year  [extern]
```

Current year (e.g., 2025).

Definition at line 16 of file ntp.c.

4.32 ntp.h

Go to the documentation of this file.

```
00001 #ifndef NTP_H
00002 #define NTP_H
00003
00004 #include <sys/time.h>
00005 #include "esp_netif_sntp.h"
00006 #include "esp_sntp.h"
00007
00011 extern int hour;
00012 extern int minute;
00013 extern int second;
00014 extern int day;
00015 extern int month;
00016 extern int year;
00017 extern int day_in_year;
00018
00022 void obtain_time(void);
00023
00028 bool is_ntp_sync(void);
00029
00030 #endif
```

4.33 main/nvs_utils.c File Reference

```
#include "nvs_utils.h"
#include "nvs_flash.h"
#include "nvs.h"
#include "esp_log.h"
```

Macros

- #define NVS_NAMESPACE "storage"

  Namespace used for storing data in NVS.
- #define NVS_KEY "json_config"

  Key used to store JSON configuration data in NVS.

Functions

- esp_err_t nvs_init (void)

  Initializes the Non-Volatile Storage (NVS) system.
- void save_config_to_nvs (const char *data, int data_len)

  Saves configuration data to NVS.
- esp_err_t load_config_from_nvs (char **data, size_t *data_len)

  Loads configuration data from NVS.
- esp_err_t delete_config_from_nvs (void)

  Deletes configuration data from NVS.

Variables

- static const char * TAG = "nvs_module"

  Tag for logging messages from the NVS utility module.

4.33.1 Macro Definition Documentation

4.33.1.1 NVS_KEY

```
#define NVS_KEY "json_config"
```

Key used to store JSON configuration data in NVS.

Definition at line 14 of file nvs_utils.c.

4.33.1.2 NVS_NAMESPACE

```
#define NVS_NAMESPACE "storage"
```

Namespace used for storing data in NVS.

Definition at line 9 of file nvs_utils.c.

4.33.2 Function Documentation

4.33.2.1 delete_config_from_nvs()

```
esp_err_t delete_config_from_nvs (
            void )
```

Deletes configuration data from NVS.

Returns

esp_err_t ESP_OK on success, or an error code on failure.

Definition at line 155 of file nvs_utils.c.

4.33.2.2 load_config_from_nvs()

```
esp_err_t load_config_from_nvs (
            char ** data,
            size_t * data_len)
```

Loads configuration data from NVS.

Parameters

data	Pointer to a buffer where the loaded data will be stored.
data_len	Pointer to a variable where the length of the loaded data will be stored.

Returns

esp_err_t ESP_OK on success, or an error code on failure.

Definition at line 88 of file nvs_utils.c.

4.33.2.3 nvs_init()

```
esp_err_t nvs_init (
            void )
```

Initializes the Non-Volatile Storage (NVS) system.

Returns

esp_err_t ESP_OK on success, or an error code on failure.

Definition at line 25 of file nvs_utils.c.

4.33.2.4 save_config_to_nvs()

```
void save_config_to_nvs (
            const char * data,
            int data_len)
```

Saves configuration data to NVS.

Parameters

data	Pointer to the configuration data to be saved.
data_len	Length of the configuration data in bytes.

Definition at line 53 of file nvs_utils.c.

4.33.3 Variable Documentation

4.33.3.1 TAG

```
const char* TAG = "nvs_module"  [static]
```

Tag for logging messages from the NVS utility module.

Definition at line 19 of file nvs_utils.c.

4.34 nvs_utils.c

Go to the documentation of this file.

```
00001 #include "nvs_utils.h"
00002 #include "nvs_flash.h"
00003 #include "nvs.h"
00004 #include "esp_log.h"
00005
00009 #define NVS_NAMESPACE "storage"
00010
00014 #define NVS_KEY "json_config"
00015
00019 static const char *TAG = "nvs_module";
00020
00025 esp_err_t nvs_init(void) {
00026     esp_err_t err;
00027
00028     // Attempt to initialize NVS
00029     err = nvs_flash_init();
00030     if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
00031         // If NVS is full or version is incompatible, erase and retry
00032         ESP_LOGW(TAG, "NVS partition full or version mismatch, erasing...");
00033         err = nvs_flash_erase();
00034         if (err != ESP_OK) {
00035             ESP_LOGE(TAG, "Failed to erase NVS: %s", esp_err_to_name(err));
00036             return err;
00037         }
00038         err = nvs_flash_init();
00039     }
00040     if (err != ESP_OK) {
00041         ESP_LOGE(TAG, "Error initializing NVS: %s", esp_err_to_name(err));
00042     } else {
00043         ESP_LOGI(TAG, "NVS initialized successfully");
00044     }
00045     return err;
00046 }
00047
00053 void save_config_to_nvs(const char *data, int data_len) {
00054     nvs_handle_t nvs_handle;
00055     esp_err_t err;
00056
00057     // Open NVS namespace in read-write mode
00058     err = nvs_open(NVS_NAMESPACE, NVS_READWRITE, &nvs_handle);
00059     if (err != ESP_OK) {
00060         ESP_LOGE(TAG, "Error opening NVS: %s", esp_err_to_name(err));
00061         return;
00062     }
00063
00064     // Save data as a binary blob
00065     err = nvs_set_blob(nvs_handle, NVS_KEY, data, data_len);
```

```
00066     if (err != ESP_OK) {
00067         ESP_LOGE(TAG, "Error saving data: %s", esp_err_to_name(err));
00068     } else {
00069         ESP_LOGI(TAG, "JSON configuration successfully saved in NVS");
00070     }
00071
00072     // Commit changes to ensure they are written to flash
00073     err = nvs_commit(nvs_handle);
00074     if (err != ESP_OK) {
00075         ESP_LOGE(TAG, "Error committing NVS: %s", esp_err_to_name(err));
00076     }
00077
00078     // Close NVS handle
00079     nvs_close(nvs_handle);
00080 }
00081
00088 esp_err_t load_config_from_nvs(char **data, size_t *data_len) {
00089     nvs_handle_t nvs_handle;
00090     esp_err_t err;
00091     size_t required_size = 0;
00092
00093     // Initialize output parameters
00094     *data = NULL;
00095     *data_len = 0;
00096
00097     // Open NVS namespace in read-only mode
00098     err = nvs_open(NVS_NAMESPACE, NVS_READONLY, &nvs_handle);
00099     if (err != ESP_OK) {
00100         ESP_LOGE(TAG, "Error opening NVS %s namespace: %s", NVS_NAMESPACE, esp_err_to_name(err));
00101         return err;
00102     }
00103
00104     // Get the size of the stored blob
00105     err = nvs_get_blob(nvs_handle, NVS_KEY, NULL, &required_size);
00106     if (err == ESP_ERR_NVS_NOT_FOUND) {
00107         ESP_LOGW(TAG, "JSON configuration data not found in NVS");
00108         nvs_close(nvs_handle);
00109         return err;
00110     } else if (err != ESP_OK) {
00111         ESP_LOGE(TAG, "Error reading size: %s", esp_err_to_name(err));
00112         nvs_close(nvs_handle);
00113         return err;
00114     }
00115
00116     // Check if data exists
00117     if (required_size == 0) {
00118         ESP_LOGW(TAG, "No data found for key %s", NVS_KEY);
00119         nvs_close(nvs_handle);
00120         return ESP_ERR_NVS_NOT_FOUND;
00121     }
00122
00123     // Allocate memory for the data
00124     *data = (char *)malloc(required_size + 1); // +1 for null terminator
00125     if (*data == NULL) {
00126         ESP_LOGE(TAG, "Memory allocation error");
00127         nvs_close(nvs_handle);
00128         return ESP_ERR_NO_MEM;
00129     }
00130
00131     // Read the data
00132     err = nvs_get_blob(nvs_handle, NVS_KEY, *data, &required_size);
00133     if (err != ESP_OK) {
00134         ESP_LOGE(TAG, "Error reading data: %s", esp_err_to_name(err));
00135         free(*data);
00136         *data = NULL;
00137         nvs_close(nvs_handle);
00138         return err;
00139     }
00140
00141     // Add null terminator
00142     (*data)[required_size] = '\0';
00143     ESP_LOGI(TAG, "JSON configuration successfully read from NVS");
00144     *data_len = required_size;
00145
00146     // Close NVS handle
00147     nvs_close(nvs_handle);
00148     return ESP_OK;
00149 }
00150
00155 esp_err_t delete_config_from_nvs(void) {
00156     nvs_handle_t nvs_handle;
00157     esp_err_t err;
00158
00159     // Open NVS namespace in read-write mode
00160     err = nvs_open(NVS_NAMESPACE, NVS_READWRITE, &nvs_handle);
00161     if (err != ESP_OK) {
00162         ESP_LOGE(TAG, "Error opening NVS %s namespace: %s", NVS_NAMESPACE, esp_err_to_name(err));
```

```
00163         return err;
00164     }
00165
00166     // Erase the specified key
00167     err = nvs_erase_key(nvs_handle, NVS_KEY);
00168     if (err == ESP_ERR_NVS_NOT_FOUND) {
00169         ESP_LOGW(TAG, "JSON configuration data not found in NVS, nothing to delete");
00170         nvs_close(nvs_handle);
00171         return err;
00172     } else if (err != ESP_OK) {
00173         ESP_LOGE(TAG, "Error deleting data: %s", esp_err_to_name(err));
00174         nvs_close(nvs_handle);
00175         return err;
00176     }
00177
00178     // Commit changes to ensure deletion is written to flash
00179     err = nvs_commit(nvs_handle);
00180     if (err != ESP_OK) {
00181         ESP_LOGE(TAG, "Error committing NVS after deletion: %s", esp_err_to_name(err));
00182         nvs_close(nvs_handle);
00183         return err;
00184     }
00185
00186     ESP_LOGI(TAG, "JSON configuration successfully deleted from NVS");
00187     // Close NVS handle
00188     nvs_close(nvs_handle);
00189     return ESP_OK;
00190 }
```

4.35 main/nvs_utils.h File Reference

```
#include <esp_err.h>
```

Functions

- esp_err_t nvs_init (void)

  Initializes the Non-Volatile Storage (NVS) system.
- void save_config_to_nvs (const char *data, int data_len)

  Saves configuration data to NVS.
- esp_err_t load_config_from_nvs (char **data, size_t *data_len)

  Loads configuration data from NVS.
- esp_err_t delete_config_from_nvs (void)

  Deletes configuration data from NVS.

4.35.1 Function Documentation

4.35.1.1 delete_config_from_nvs()

```
esp_err_t delete_config_from_nvs (
            void )
```

Deletes configuration data from NVS.

Returns

esp_err_t ESP_OK on success, or an error code on failure.

Definition at line 155 of file nvs_utils.c.

4.35.1.2 load_config_from_nvs()

```
esp_err_t load_config_from_nvs (
            char ** data,
            size_t * data_len)
```

Loads configuration data from NVS.

Parameters

data	Pointer to a buffer where the loaded data will be stored.
data_len	Pointer to a variable where the length of the loaded data will be stored.

Returns

esp_err_t ESP_OK on success, or an error code on failure.

Definition at line 88 of file nvs_utils.c.

4.35.1.3 nvs_init()

```
esp_err_t nvs_init (
            void )
```

Initializes the Non-Volatile Storage (NVS) system.

Returns

esp_err_t ESP_OK on success, or an error code on failure.

Definition at line 25 of file nvs_utils.c.

4.35.1.4 save_config_to_nvs()

```
void save_config_to_nvs (
            const char * data,
            int data_len)
```

Saves configuration data to NVS.

Parameters

data	Pointer to the configuration data to be saved.
data_len	Length of the configuration data in bytes.

Definition at line 53 of file nvs_utils.c.

4.36 nvs_utils.h

Go to the documentation of this file.

```
00001 #ifndef NVS_UTILS_H
00002 #define NVS_UTILS_H
00003
00004 #include <esp_err.h>
00005
00010 esp_err_t nvs_init(void);
00011
00017 void save_config_to_nvs(const char *data, int data_len);
00018
00025 esp_err_t load_config_from_nvs(char **data, size_t *data_len);
00026
00031 esp_err_t delete_config_from_nvs(void);
00032
00033 #endif // NVS_UTILS_H
```

4.37 main/one_wire_detect.c File Reference

```
#include "one_wire_detect.h"
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "freertos/FreeRTOS.h"
#include "driver/gpio.h"
#include "cJSON.h"
#include "onewire.h"
#include "device_config.h"
```

Data Structures

- struct SensorState

Macros

- #define DETECTION_THRESHOLD 3

  Number of consecutive detections to confirm a sensor.
- #define MISS_THRESHOLD 3

  Number of consecutive misses to remove a sensor.

Functions

- char * search_for_one_wire_sensors (void)

  Search for one-wire sensors on configured pins and return their addresses as JSON.

Variables

- static const char * TAG = "ONE_WIRE_DETECT"

  Tag for logging messages from the one-wire detection module.
- static SensorState * sensor_states = NULL

  Array of sensor states.
- static size_t sensor_count = 0

  Current number of detected sensors.
- static size_t sensor_capacity = 0

  Capacity of the sensor_states array.

4.37.1 Macro Definition Documentation

4.37.1.1 DETECTION_THRESHOLD

```
#define DETECTION_THRESHOLD 3
```

Number of consecutive detections to confirm a sensor.

Definition at line 18 of file one_wire_detect.c.

4.37.1.2 MISS_THRESHOLD

```
#define MISS_THRESHOLD 3
```

Number of consecutive misses to remove a sensor.

Definition at line 19 of file one_wire_detect.c.

4.37.2 Function Documentation

4.37.2.1 search_for_one_wire_sensors()

```
char * search_for_one_wire_sensors (
            void )
```

Search for one-wire sensors on configured pins and return their addresses as JSON.

Searches for one-wire sensors on configured GPIO pins and returns their addresses as a JSON string.

Returns

char* JSON string containing detected sensor pins and addresses, or NULL on error.

Definition at line 37 of file one_wire_detect.c.

4.37.3 Variable Documentation

4.37.3.1 sensor_capacity

```
size_t sensor_capacity = 0  [static]
```

Capacity of the sensor_states array.

Definition at line 31 of file one_wire_detect.c.

4.37.3.2 sensor_count

```
size_t sensor_count = 0  [static]
```

Current number of detected sensors.

Definition at line 30 of file one_wire_detect.c.

4.37.3.3 sensor_states

```
SensorState* sensor_states = NULL  [static]
```

Array of sensor states.

Definition at line 29 of file one_wire_detect.c.

4.37.3.4 TAG

```
const char* TAG = "ONE_WIRE_DETECT"  [static]
```

Tag for logging messages from the one-wire detection module.

Definition at line 15 of file one_wire_detect.c.

4.38 one_wire_detect.c

Go to the documentation of this file.

```
00001 #include "one_wire_detect.h"
00002 #include <string.h>
00003 #include <stdio.h>
00004 #include <stdlib.h>
00005 #include "freertos/FreeRTOS.h"
00006 #include "driver/gpio.h"
00007 #include "cJSON.h"
00008 #include "onewire.h"
00009
00010 #include "device_config.h"
00011
00015 static const char *TAG = "ONE_WIRE_DETECT";
00016
00017 // Constants for debouncing
00018 #define DETECTION_THRESHOLD 3
00019 #define MISS_THRESHOLD 3
00020
00021 // Structure to track sensor state
00022 typedef struct {
00023     int pin;
00024     char address[17];
00025     int detection_count;
00026 } SensorState;
00027
00028 // Static variables
00029 static SensorState *sensor_states = NULL;
00030 static size_t sensor_count = 0;
00031 static size_t sensor_capacity = 0;
00032
00037 char *search_for_one_wire_sensors(void) {
00038     onewire_search_t search;
00039     onewire_addr_t addr;
00040
00041     // Validate device configuration
00042     if (!_device.one_wire_inputs || _device.one_wire_inputs_len == 0) {
00043         // Create empty JSON
00044         cJSON *root = cJSON_CreateObject();
00045         cJSON *pins = cJSON_CreateArray();
00046         cJSON_AddItemToObject(root, "pins", pins);
00047         char *json_str = cJSON_PrintUnformatted(root);
00048         cJSON_Delete(root);
00049         return json_str;
00050     }
00051
00052     // Temporary array to mark which sensors were seen in this scan
00053     bool *seen = calloc(sensor_count, sizeof(bool));
00054     if (!seen && sensor_count > 0) {
00055         ESP_LOGE(TAG, "Failed to allocate seen array");
00056         return NULL;
00057     }
00058
00059     // Create JSON object for stable sensors
00060     cJSON *root = cJSON_CreateObject();
00061     cJSON *pins = cJSON_CreateArray();
00062     cJSON_AddItemToObject(root, "pins", pins);
00063
00064     // Scan each pin
00065     for (size_t pin_index = 0; pin_index < _device.one_wire_inputs_len; pin_index++) {
00066         int one_wire_gpio = _device.one_wire_inputs[pin_index];
00067
00068         // Create JSON for the current pin
00069         cJSON *pin_obj = cJSON_CreateObject();
00070         cJSON_AddNumberToObject(pin_obj, "pin", one_wire_gpio);
00071         cJSON *addresses = cJSON_CreateArray();
00072         cJSON_AddItemToObject(pin_obj, "addresses", addresses);
00073
```

```
00074         // Find all devices on the OneWire bus for the current pin
00075         onewire_search_start(&search);
00076         while ((addr = onewire_search_next(&search, one_wire_gpio)) != ONEWIRE_NONE) {
00077             // Format address as hexadecimal string
00078             char addr_str[17];
00079             snprintf(addr_str, sizeof(addr_str), "%016llX", addr);
00080 
00081             // Check if sensor exists in state list
00082             int sensor_index = -1;
00083             for (size_t i = 0; i < sensor_count; i++) {
00084                 if (sensor_states && sensor_states[i].pin == one_wire_gpio &&
      strcmp(sensor_states[i].address, addr_str) == 0) {
00085                     sensor_index = i;
00086                     if (seen) seen[i] = true;
00087                     break;
00088                 }
00089             }
00090 
00091             // Update or add sensor state
00092             if (sensor_index >= 0) {
00093                 // Existing sensor, increment detection count
00094                 if (sensor_states[sensor_index].detection_count < DETECTION_THRESHOLD) {
00095                     sensor_states[sensor_index].detection_count++;
00096                 }
00097             } else {
00098                 // New sensor, add to state list
00099                 if (sensor_count >= sensor_capacity) {
00100                     size_t new_capacity = sensor_capacity ? sensor_capacity * 2 : 8;
00101                     SensorState *new_states = realloc(sensor_states, new_capacity *
      sizeof(SensorState));
00102                     if (!new_states) {
00103                         ESP_LOGE(TAG, "Failed to allocate sensor states");
00104                         free(seen);
00105                         cJSON_Delete(root);
00106                         return NULL;
00107                     }
00108                     sensor_states = new_states;
00109                     sensor_capacity = new_capacity;
00110                 }
00111                 sensor_states[sensor_count] = (SensorState){
00112                     .pin = one_wire_gpio,
00113                     .detection_count = 1
00114                 };
00115                 strncpy(sensor_states[sensor_count].address, addr_str,
      sizeof(sensor_states[sensor_count].address));
00116                 if (seen) seen[sensor_count] = true;
00117                 sensor_count++;
00118             }
00119         }
00120 
00121         // Add stable sensors to JSON (detection_count >= DETECTION_THRESHOLD)
00122         for (size_t i = 0; i < sensor_count; i++) {
00123             if (sensor_states && sensor_states[i].pin == one_wire_gpio &&
      sensor_states[i].detection_count >= DETECTION_THRESHOLD) {
00124                 cJSON_AddItemToArray(addresses, cJSON_CreateString(sensor_states[i].address));
00125             }
00126         }
00127 
00128         // Add pin object to the pins array
00129         cJSON_AddItemToArray(pins, pin_obj);
00130     }
00131 
00132     // Update miss counts for sensors not seen in this scan
00133     for (size_t i = 0; i < sensor_count; i++) {
00134         if (seen && !seen[i]) {
00135             if (sensor_states && sensor_states[i].detection_count > -MISS_THRESHOLD) {
00136                 sensor_states[i].detection_count--;
00137             }
00138         }
00139     }
00140 
00141     // Remove sensors that have been missed too many times
00142     for (size_t i = 0; i < sensor_count;) {
00143         if (sensor_states && sensor_states[i].detection_count <= -MISS_THRESHOLD) {
00144             for (size_t j = i; j < sensor_count - 1; j++) {
00145                 sensor_states[j] = sensor_states[j + 1];
00146             }
00147             sensor_count--;
00148         } else {
00149             i++;
00150         }
00151     }
00152 
00153     if (seen) {
00154         free(seen);
00155     }
00156 
```

```
00157     char *json_str = cJSON_PrintUnformatted(root);
00158     cJSON_Delete(root);
00159     return json_str;
00160 }
```

4.39 main/one_wire_detect.h File Reference

Functions

- char * search_for_one_wire_sensors (void)

  Searches for one-wire sensors on configured GPIO pins and returns their addresses as a JSON string.

4.39.1 Function Documentation

4.39.1.1 search_for_one_wire_sensors()

```
char * search_for_one_wire_sensors (
            void )
```

Searches for one-wire sensors on configured GPIO pins and returns their addresses as a JSON string.

Returns

char* JSON string containing detected sensor pins and addresses, or NULL on error.

Searches for one-wire sensors on configured GPIO pins and returns their addresses as a JSON string.

Returns

char* JSON string containing detected sensor pins and addresses, or NULL on error.

Definition at line 37 of file one_wire_detect.c.

4.40 one_wire_detect.h

Go to the documentation of this file.

```
00001 #ifndef ONE_WIRE_DETECT_H
00002 #define ONE_WIRE_DETECT_H
00003
00008 char *search_for_one_wire_sensors(void);
00009
00010 #endif // ONE_WIRE_DETECT_H
```

4.41 main/sensor.c File Reference

```
#include "sensor.h"
#include "esp_log.h"
#include "driver/gpio.h"
#include <string.h>
#include "ds18x20.h"
```

Functions

- static onewire_addr_t parse_sensor_address (const char ∗sensor_address)

  Convert a hex string to a onewire_addr_t (64-bit address).
- float read_one_wire_sensor (const char ∗sensor_type, const char ∗sensor_address, int pin)

  Read data from a one-wire sensor.

Variables

- static const char ∗ TAG = "SENSORS"

  Tag for logging messages from the sensors module.

4.41.1 Function Documentation

4.41.1.1 parse_sensor_address()

```
static onewire_addr_t parse_sensor_address (
            const char * sensor_address)  [static]
```

Convert a hex string to a onewire_addr_t (64-bit address).

Parameters

sensor_address	Hex string representing the sensor address.

Returns

onewire_addr_t Parsed address, or DS18X20_ANY on failure.

Definition at line 19 of file sensor.c.

4.41.1.2 read_one_wire_sensor()

```
float read_one_wire_sensor (
            const char * sensor_type,
            const char * sensor_address,
            int pin)
```

Read data from a one-wire sensor.

Parameters

sensor_type	Type of the one-wire sensor (e.g., DS18B20).
sensor_address	Address or identifier of the sensor.
pin	GPIO pin number connected to the one-wire bus.

Returns

float Sensor reading value, or a default value (e.g., 0.0) on error.

Definition at line 33 of file sensor.c.

4.41.2 Variable Documentation

4.41.2.1 TAG

```
const char* TAG = "SENSORS"  [static]
```

Tag for logging messages from the sensors module.

Definition at line 12 of file sensor.c.

4.42 sensor.c

Go to the documentation of this file.

```
00001 #include "sensor.h"
00002 #include "esp_log.h"
00003 #include "driver/gpio.h"
00004 #include <string.h>
00005
00006 // Temperature sensors
00007 #include "ds18x20.h"
00008
00012 static const char *TAG = "SENSORS";
00013
00019 static onewire_addr_t parse_sensor_address(const char *sensor_address) {
00020     if (!sensor_address || strlen(sensor_address) != 16) {
00021         ESP_LOGE(TAG, "Invalid sensor address format");
00022         return DS18X20_ANY;
00023     }
00024
00025     uint64_t addr = 0;
00026     if (sscanf(sensor_address, "%016llx", &addr) != 1) {
00027         ESP_LOGE(TAG, "Failed to parse sensor address");
00028         return DS18X20_ANY;
00029     }
00030     return (onewire_addr_t)addr;
00031 }
00032
00033 float read_one_wire_sensor(const char *sensor_type, const char *sensor_address, int pin) {
00034     float value = 0.0f;
00035     esp_err_t err;
00036     onewire_addr_t addr = parse_sensor_address(sensor_address);
00037
00038     if (!sensor_type) {
00039         ESP_LOGE(TAG, "Invalid sensor type");
00040         return 0;
00041     }
00042
00043     // Read temperature from a DS18x20 sensor
00044     if (strcmp(sensor_type, "DS18S20/DS1820 (Temperature Sensor)") == 0 || strcmp(sensor_type, "DS1822
      (Temperature Sensor)") == 0) {
00045         err = ds18s20_measure_and_read(pin, addr, &value);
00046     } else if (strcmp(sensor_type, "DS18B20 (Temperature Sensor)") == 0) {
00047         err = ds18b20_measure_and_read(pin, addr, &value);
00048     } else if (strcmp(sensor_type, "MAX31850 (Temperature Sensor)") == 0) {
00049         err = max31850_measure_and_read(pin, addr, &value);
00050     }
00051     // Add other sensor types here
00052     else {
00053         ESP_LOGE(TAG, "Unknown sensor type: %s", sensor_type);
00054         return 0;
00055     }
00056
00057     if (err != ESP_OK) {
00058         ESP_LOGE(TAG, "Failed to read sensor value: %s", esp_err_to_name(err));
00059         return 0;
00060     }
00061
00062     return value;
00063 }
```

4.43 main/sensor.h File Reference

Functions

- float read_one_wire_sensor (const char ∗sensor_type, const char ∗sensor_address, int pin)

  Read data from a one-wire sensor.

4.43.1 Function Documentation

4.43.1.1 read_one_wire_sensor()

```
float read_one_wire_sensor (
            const char * sensor_type,
            const char * sensor_address,
            int pin)
```

Read data from a one-wire sensor.

Parameters

sensor_type	Type of the one-wire sensor (e.g., DS18B20).
sensor_address	Address or identifier of the sensor.
pin	GPIO pin number connected to the one-wire bus.

Returns

float Sensor reading value, or a default value (e.g., 0.0) on error.

Definition at line 33 of file sensor.c.

4.44 sensor.h

Go to the documentation of this file.

```
00001 #ifndef SENSORS_H
00002 #define SENSORS_H
00003
00011 float read_one_wire_sensor(const char *sensor_type, const char *sensor_address, int pin);
00012
00013 #endif // SENSORS_H
```

4.45 main/TM7711.c File Reference

```
#include "TM7711.h"
#include "esp_system.h"
#include "esp_err.h"
#include "rom/ets_sys.h"
```

Functions

- esp_err_t tm7711_init (int dout_pin, int sck_pin)

  Initialize the TM7711 ADC with specified pins.
- esp_err_t tm7711_read (unsigned char next_select, int dout_pin, int sck_pin, unsigned long *data)

  Read data from the TM7711 ADC.

4.45.1 Function Documentation

4.45.1.1 tm7711_init()

```
esp_err_t tm7711_init (
            int dout_pin,
            int sck_pin)
```

Initialize the TM7711 ADC with specified pins.

Parameters

dout_pin	GPIO pin for data output.
sck_pin	GPIO pin for serial clock.

Returns

esp_err_t ESP_OK on success, or an error code on failure.

Definition at line 6 of file TM7711.c.

4.45.1.2 tm7711_read()

```
esp_err_t tm7711_read (
            unsigned char next_select,
            int dout_pin,
            int sck_pin,
            unsigned long * data)
```

Read data from the TM7711 ADC.

Parameters

next_select	Mode selection for the next reading (e.g., CH1_10HZ, CH1_40HZ, CH2_TEMP).
dout_pin	GPIO pin for data output.
sck_pin	GPIO pin for serial clock.
data	Pointer to store the read data.

Returns

esp_err_t ESP_OK on success, or an error code on failure.

Definition at line 23 of file TM7711.c.

4.46 TM7711.c

Go to the documentation of this file.

```
00001 #include "TM7711.h"
00002 #include "esp_system.h"
00003 #include "esp_err.h"
00004 #include "rom/ets_sys.h"
00005
00006 esp_err_t tm7711_init(int dout_pin, int sck_pin) {
00007     esp_err_t ret = ESP_OK;
00008
00009     // Reset GPIO pins
00010     ret |= gpio_reset_pin(sck_pin);
00011     ret |= gpio_set_direction(sck_pin, GPIO_MODE_OUTPUT);
00012     ret |= gpio_reset_pin(dout_pin);
00013     ret |= gpio_set_direction(dout_pin, GPIO_MODE_INPUT);
00014
00015     // Send reset pulse (SCK high for >200us)
00016     gpio_set_level(sck_pin, 1);
00017     ets_delay_us(200);
00018     gpio_set_level(sck_pin, 0);
00019
00020     return ret;
00021 }
00022
00023 esp_err_t tm7711_read(unsigned char next_select, int dout_pin, int sck_pin, unsigned long *data) {
00024     unsigned char i;
00025     unsigned long data_temp = 0;
00026     unsigned char pulses;
00027     int timeout;
00028     int retries = 3; // Maximum 3 attempts
00029
00030     // Determine additional clock pulses and timeout based on mode
00031     switch (next_select) {
00032         case CH1_10HZ:
00033             timeout = 120000; // 120ms for 10Hz
00034             pulses = CH1_10HZ_CLK - 24;  // 1 pulse
00035             break;
00036         case CH1_40HZ:
00037             timeout = 30000;  // 30ms for 40Hz
00038             pulses = CH1_40HZ_CLK - 24;  // 3 pulses
00039             break;
00040         case CH2_TEMP:
00041             timeout = 60000;  // 60ms for temperature
00042             pulses = CH2_TEMP_CLK - 24;  // 2 pulses
00043             break;
00044         default:
00045             return ESP_ERR_INVALID_ARG;
00046     }
00047
00048     if (data == NULL) {
00049         return ESP_ERR_INVALID_ARG;
00050     }
00051
00052     while (retries--) {
00053         // Wait for DOUT to go low (data ready)
00054         int temp_timeout = timeout;
00055         while (gpio_get_level(dout_pin) && temp_timeout--) {
00056             ets_delay_us(1);
00057         }
00058         if (temp_timeout <= 0) {
00059             if (retries == 0) {
00060                 return ESP_ERR_TIMEOUT;
00061             }
00062             continue; // Retry
00063         }
00064
00065         // Read 24 bits
00066         for (i = 0; i < 24; i++) {
00067             gpio_set_level(sck_pin, 1);  // SCK high
00068             ets_delay_us(5);             // Wait 5us
00069             data_temp «= 1;             // Shift data left
00070             if (gpio_get_level(dout_pin)) {
00071                 data_temp |= 1;          // Read bit from DOUT
00072             }
00073             gpio_set_level(sck_pin, 0);  // SCK low
00074             ets_delay_us(5);             // Wait 5us
00075         }
00076
00077         // Send additional clock pulses for next mode
00078         for (i = 0; i < pulses; i++) {
00079             gpio_set_level(sck_pin, 1);
00080             ets_delay_us(1);
00081             gpio_set_level(sck_pin, 0);
00082             ets_delay_us(1);
```

```
00083         }
00084
00085         *data = data_temp;
00086         return ESP_OK;
00087     }
00088     return ESP_ERR_TIMEOUT;
00089 }
```

4.47 main/TM7711.h File Reference

```
#include <driver/gpio.h>
```

Macros

- #define CH1_10HZ 0x01

  Mode for Channel 1 with 10 Hz sampling rate.
- #define CH1_40HZ 0x02

  Mode for Channel 1 with 40 Hz sampling rate.
- #define CH2_TEMP 0x03

  Mode for Channel 2, temperature measurement.
- #define CH1_10HZ_CLK 25

  Number of clock pulses for Channel 1 at 10 Hz.
- #define CH1_40HZ_CLK 27

  Number of clock pulses for Channel 1 at 40 Hz.
- #define CH2_TEMP_CLK 26

  Number of clock pulses for Channel 2 temperature measurement.

Functions

- esp_err_t tm7711_init (int dout_pin, int sck_pin)

  Initialize the TM7711 ADC with specified pins.
- esp_err_t tm7711_read (unsigned char next_select, int dout_pin, int sck_pin, unsigned long *data)

  Read data from the TM7711 ADC.

4.47.1 Macro Definition Documentation

4.47.1.1 CH1_10HZ

```
#define CH1_10HZ 0x01
```

Mode for Channel 1 with 10 Hz sampling rate.

Definition at line 9 of file TM7711.h.

4.47.1.2 CH1_10HZ_CLK

```
#define CH1_10HZ_CLK 25
```

Number of clock pulses for Channel 1 at 10 Hz.

Definition at line 24 of file TM7711.h.

4.47.1.3 CH1_40HZ

```
#define CH1_40HZ 0x02
```

Mode for Channel 1 with 40 Hz sampling rate.

Definition at line 14 of file TM7711.h.

4.47.1.4 CH1_40HZ_CLK

```
#define CH1_40HZ_CLK 27
```

Number of clock pulses for Channel 1 at 40 Hz.

Definition at line 29 of file TM7711.h.

4.47.1.5 CH2_TEMP

```
#define CH2_TEMP 0x03
```

Mode for Channel 2, temperature measurement.

Definition at line 19 of file TM7711.h.

4.47.1.6 CH2_TEMP_CLK

```
#define CH2_TEMP_CLK 26
```

Number of clock pulses for Channel 2 temperature measurement.

Definition at line 34 of file TM7711.h.

4.47.2 Function Documentation

4.47.2.1 tm7711_init()

```
esp_err_t tm7711_init (
            int dout_pin,
            int sck_pin)
```

Initialize the TM7711 ADC with specified pins.

Parameters

dout_pin	GPIO pin for data output.
sck_pin	GPIO pin for serial clock.

Returns

esp_err_t ESP_OK on success, or an error code on failure.

Definition at line 6 of file TM7711.c.

4.47.2.2 tm7711_read()

```
esp_err_t tm7711_read (
            unsigned char next_select,
            int dout_pin,
            int sck_pin,
            unsigned long * data)
```

Read data from the TM7711 ADC.

Parameters

next_select	Mode selection for the next reading (e.g., CH1_10HZ, CH1_40HZ, CH2_TEMP).
dout_pin	GPIO pin for data output.
sck_pin	GPIO pin for serial clock.
data	Pointer to store the read data.

Returns

esp_err_t ESP_OK on success, or an error code on failure.

Definition at line 23 of file TM7711.c.

4.48 TM7711.h

Go to the documentation of this file.

```
00001 #ifndef _TM7711_H_
00002 #define _TM7711_H_
00003
00004 #include <driver/gpio.h>
00005
00009 #define CH1_10HZ    0x01
00010
00014 #define CH1_40HZ    0x02
00015
00019 #define CH2_TEMP    0x03
00020
00024 #define CH1_10HZ_CLK  25
00025
00029 #define CH1_40HZ_CLK  27
00030
00034 #define CH2_TEMP_CLK  26
00035
00042 esp_err_t tm7711_init(int dout_pin, int sck_pin);
00043
00052 esp_err_t tm7711_read(unsigned char next_select, int dout_pin, int sck_pin, unsigned long *data);
00053
00054 #endif
```

4.49 main/variables.c File Reference

```
#include "variables.h"
#include <stdlib.h>
#include <string.h>
#include "esp_log.h"
#include <math.h>
#include "device_config.h"
#include "mqtt.h"
#include "cJSON.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "adc_sensor.h"
```

Functions

- static void one_wire_read_task (void *pvParameters)

  Task function to read OneWire sensor values.
- static void adc_sensor_read_task (void *pvParameters)

  Task function to read ADC sensor values.
- static void variables_list_init (void)

  Initialize the global variable list as empty.
- static bool variables_list_add (VariableType type, void *data)

  Add a variable to the global list with the specified type and data.
- static void free_variable (VariableType type, void *data)

  Free memory allocated for a single variable.
- static void variables_list_free (void)

  Free the global variable list and all associated variables.
- bool load_variables (cJSON *variables)

  Load variables from a cJSON object.
- VariableNode * find_variable (const char *search_name)

  Find a variable by name.
- VariableNode * find_current_time_variable (void)

  Find the current time variable.
- void parse_variable_name (const char *var_name, char *base_name, size_t base_name_size, const char **suffix)

  Parse a variable name into base name and suffix.
- bool read_variable (const char *var_name)

  Read a boolean variable by name.
- void write_variable (const char *var_name, bool value)

  Write a boolean value to a variable.
- double read_numeric_variable (const char *var_name)

  Read a numeric variable by name.
- void write_numeric_variable (const char *var_name, double value)

  Write a numeric value to a variable.
- char * read_variables_json (void)

  Read all variables as a JSON string.
- void update_variables_from_children (const char *json_str)

  Update variables from a JSON string received from child nodes.
- void send_variables_to_parents ()

  Send variable updates to parent nodes.

Variables

- static const char * TAG = "VARIABLES"

  Tag for logging messages from the variables module.
- VariablesList variables_list = {0}

  Global list of variables.
- static TaskHandle_t one_wire_task_handle = NULL

  Handle for the OneWire read task.
- static TaskHandle_t adc_sensor_task_handle = NULL

  Handle for the ADC sensor read task.

4.49.1 Function Documentation

4.49.1.1 adc_sensor_read_task()

```
static void adc_sensor_read_task (
            void * pvParameters)  [static]
```

Task function to read ADC sensor values.

Task function to periodically read ADC sensor values.

Parameters

pvParameters	Task parameters (unused).

Definition at line 696 of file variables.c.

4.49.1.2 find_current_time_variable()

```
VariableNode * find_current_time_variable (
            void )
```

Find the current time variable.

Returns

VariableNode* Pointer to the time variable node, or NULL if not found.

Definition at line 440 of file variables.c.

4.49.1.3 find_variable()

```
VariableNode * find_variable (
            const char * search_name)
```

Find a variable by name.

Parameters

search_name	Name of the variable to find.

Returns

VariableNode* Pointer to the variable node, or NULL if not found.

Definition at line 387 of file variables.c.

4.49.1.4 free_variable()

```
static void free_variable (
            VariableType type,
            void * data)  [static]
```

Free memory allocated for a single variable.

Parameters

type	Type of the variable.
data	Pointer to the variable data.

Definition at line 73 of file variables.c.

4.49.1.5 load_variables()

```
bool load_variables (
            cJSON * variables)
```

Load variables from a cJSON object.

Parameters

variables	cJSON object containing variable definitions.

Returns

bool True if loading succeeds, false otherwise.

Definition at line 172 of file variables.c.

4.49.1.6 one_wire_read_task()

```
static void one_wire_read_task (
            void * pvParameters)  [static]
```

Task function to read OneWire sensor values.

Task function to periodically read OneWire sensor values.

Parameters

pvParameters	Task parameters (unused).

Definition at line 674 of file variables.c.

4.49.1.7 parse_variable_name()

```
void parse_variable_name (
            const char * var_name,
            char * base_name,
            size_t base_name_size,
            const char ** suffix)
```

Parse a variable name into base name and suffix.

Parameters

var_name	Full variable name.
base_name	Buffer to store the base name.
base_name_size	Size of the base_name buffer.
suffix	Pointer to store the suffix (e.g., ".CU").

Definition at line 460 of file variables.c.

4.49.1.8 read_numeric_variable()

```
double read_numeric_variable (
            const char * var_name)
```

Read a numeric variable by name.

Parameters

var_name	Name of the variable to read.

Returns

double The value of the variable, or 0.0 if not found.

Definition at line 575 of file variables.c.

4.49.1.9 read_variable()

```
bool read_variable (
            const char * var_name)
```

Read a boolean variable by name.

Parameters

var_name	Name of the variable to read.

Returns

bool The value of the variable, or false if not found.

Definition at line 484 of file variables.c.

4.49.1.10 read_variables_json()

```
char * read_variables_json (
            void )
```

Read all variables as a JSON string.

Returns

char* JSON string containing all variables, or NULL on error.

Definition at line 728 of file variables.c.

4.49.1.11 send_variables_to_parents()

```
void send_variables_to_parents (
            void )
```

Send variable updates to parent nodes.

Definition at line 893 of file variables.c.

4.49.1.12 update_variables_from_children()

```
void update_variables_from_children (
            const char * json_str)
```

Update variables from a JSON string received from child nodes.

Parameters

json_str	JSON string containing variable updates.

Definition at line 851 of file variables.c.

4.49.1.13 variables_list_add()

```
static bool variables_list_add (
            VariableType type,
            void * data)  [static]
```

Add a variable to the global list with the specified type and data.

Parameters

type	Type of the variable.
data	Pointer to the variable data.

Returns

bool True if addition succeeds, false otherwise.

Definition at line 61 of file variables.c.

4.49.1.14 variables_list_free()

```
static void variables_list_free (
            void )  [static]
```

Free the global variable list and all associated variables.

Definition at line 134 of file variables.c.

4.49.1.15 variables_list_init()

```
static void variables_list_init (
            void )  [static]
```

Initialize the global variable list as empty.

Definition at line 49 of file variables.c.

4.49.1.16 write_numeric_variable()

```
void write_numeric_variable (
            const char * var_name,
            double value)
```

Write a numeric value to a variable.

Parameters

var_name	Name of the variable to write to.
value	Numeric value to write.

Definition at line 629 of file variables.c.

4.49.1.17 write_variable()

```
void write_variable (
            const char * var_name,
            bool value)
```

Write a boolean value to a variable.

Parameters

var_name	Name of the variable to write to.
value	Boolean value to write.

Definition at line 525 of file variables.c.

4.49.2 Variable Documentation

4.49.2.1 adc_sensor_task_handle

```
TaskHandle_t adc_sensor_task_handle = NULL  [static]
```

Handle for the ADC sensor read task.

Definition at line 38 of file variables.c.

4.49.2.2 one_wire_task_handle

```
TaskHandle_t one_wire_task_handle = NULL  [static]
```

Handle for the OneWire read task.

Definition at line 27 of file variables.c.

4.49.2.3 TAG

```
const char* TAG = "VARIABLES"  [static]
```

Tag for logging messages from the variables module.

Definition at line 19 of file variables.c.

4.49.2.4 variables_list

```
VariablesList variables_list = {0}
```

Global list of variables.

Definition at line 22 of file variables.c.

4.50 variables.c

Go to the documentation of this file.

```
00001 #include "variables.h"
00002 #include <stdlib.h>
00003 #include <string.h>
00004 #include "esp_log.h"
00005 #include <math.h>
00006 #include "device_config.h"
00007
00008 #include "mqtt.h"
00009 #include "cJSON.h"
00010
00011 #include "freertos/FreeRTOS.h"
00012 #include "freertos/task.h"
00013
00014 #include "adc_sensor.h"
00015
00019 static const char *TAG = "VARIABLES";
00020
00021 // Global variables
00022 VariablesList variables_list = {0};
00023
00027 static TaskHandle_t one_wire_task_handle = NULL;
00028
00033 static void one_wire_read_task(void *pvParameters);
00034
00038 static TaskHandle_t adc_sensor_task_handle = NULL;
00039
00044 static void adc_sensor_read_task(void *pvParameters);
00045
00049 static void variables_list_init(void) {
00050     variables_list.nodes = NULL;
00051     variables_list.count = 0;
00052     variables_list.capacity = 0;
00053 }
00054
00061 static bool variables_list_add(VariableType type, void *data) {
00062     variables_list.nodes[variables_list.count].type = type;
00063     variables_list.nodes[variables_list.count].data = data;
00064     variables_list.count++;
00065     return true;
00066 }
00067
00073 static void free_variable(VariableType type, void *data) {
00074     if (!data) return;
00075     Variable *base = NULL;
00076     switch (type) {
00077         case VAR_TYPE_DIGITAL_ANALOG_IO: {
00078             DigitalAnalogInputOutput *daio = (DigitalAnalogInputOutput *)data;
00079             base = &daio->base;
00080             if (daio->pin_number) free(daio->pin_number);
00081             break;
00082         }
00083         case VAR_TYPE_ONE_WIRE: {
00084             OneWireInput *owi = (OneWireInput *)data;
00085             base = &owi->base;
00086             if (owi->pin_number) free(owi->pin_number);
00087             break;
00088         }
00089         case VAR_TYPE_ADC_SENSOR: {
00090             ADCSensor *adcs = (ADCSensor *)data;
00091             base = &adcs->base;
00092             if (adcs->sensor_type) free(adcs->sensor_type);
00093             if (adcs->pd_sck) free(adcs->pd_sck);
00094             if (adcs->dout) free(adcs->dout);
00095             if (adcs->sampling_rate) free(adcs->sampling_rate);
00096             break;
00097         }
00098         case VAR_TYPE_BOOLEAN: {
00099             Boolean *b = (Boolean *)data;
00100             base = &b->base;
00101             break;
00102         }
00103         case VAR_TYPE_NUMBER: {
00104             Number *n = (Number *)data;
00105             base = &n->base;
00106             break;
00107         }
00108         case VAR_TYPE_COUNTER: {
00109             Counter *c = (Counter *)data;
00110             base = &c->base;
00111             break;
00112         }
00113         case VAR_TYPE_TIMER: {
```

```
00114             Timer *t = (Timer *)data;
00115             base = &t->base;
00116             break;
00117         }
00118         case VAR_TYPE_TIME: {
00119             Time *t = (Time *)data;
00120             base = &t->base;
00121             break;
00122         }
00123     }
00124     if (base) {
00125         if (base->name) free(base->name);
00126         if (base->type) free(base->type);
00127     }
00128     free(data);
00129 }
00130
00134 static void variables_list_free(void) {
00135     if (!variables_list.nodes) return;
00136
00137     for (size_t i = 0; i < variables_list.count; i++) {
00138         free_variable(variables_list.nodes[i].type, variables_list.nodes[i].data);
00139     }
00140
00141     free(variables_list.nodes);
00142     variables_list.nodes = NULL;
00143     variables_list.count = 0;
00144     variables_list.capacity = 0;
00145     ESP_LOGI(TAG, "Variables List freed");
00146
00147     // Delete one_wire_read_task if it exists
00148     if (one_wire_task_handle) {
00149         vTaskDelete(one_wire_task_handle);
00150         one_wire_task_handle = NULL;
00151         ESP_LOGI(TAG, "Deleted one_wire_read_task");
00152     }
00153
00154     // Delete adc_sensor_task_handle if it exists
00155     if (adc_sensor_task_handle) {
00156         vTaskDelete(adc_sensor_task_handle);
00157         adc_sensor_task_handle = NULL;
00158         ESP_LOGI(TAG, "Deleted adc_sensor_read_task");
00159     }
00160
00161     // Free memory for ADC sensor states
00162     extern ADCSensorState sensor_states[];
00163     extern int sensor_state_count;
00164     for (int i = 0; i < sensor_state_count; i++) {
00165         if (sensor_states[i].name) {
00166             free(sensor_states[i].name);
00167         }
00168     }
00169     sensor_state_count = 0;
00170 }
00171
00172 bool load_variables(cJSON *variables) {
00173     variables_list_free();
00174     variables_list_init();
00175
00176     // Count variables and allocate exact capacity
00177     size_t var_count = cJSON_GetArraySize(variables);
00178     variables_list.nodes = (VariableNode *)calloc(var_count, sizeof(VariableNode));
00179     if (!variables_list.nodes) {
00180         ESP_LOGE(TAG, "Memory allocation failure");
00181         return false;
00182     }
00183     variables_list.capacity = var_count;
00184
00185     cJSON *var = NULL;
00186     cJSON_ArrayForEach(var, variables) {
00187         cJSON *type_item = cJSON_GetObjectItem(var, "Type");
00188         cJSON *name_item = cJSON_GetObjectItem(var, "Name");
00189
00190         const char *type_str = type_item->valuestring;
00191         const char *name = name_item->valuestring;
00192
00193         VariableType var_type;
00194         if (strcmp(type_str, "Digital Input") == 0 || strcmp(type_str, "Digital Output") == 0 ||
00195             strcmp(type_str, "Analog Input") == 0 || strcmp(type_str, "Analog Output") == 0) {
00196             var_type = VAR_TYPE_DIGITAL_ANALOG_IO;
00197         } else if (strcmp(type_str, "One Wire Input") == 0) {
00198             var_type = VAR_TYPE_ONE_WIRE;
00199         } else if (strcmp(type_str, "ADC Sensor") == 0) {
00200             var_type = VAR_TYPE_ADC_SENSOR;
00201         } else if (strcmp(type_str, "Boolean") == 0) {
00202             var_type = VAR_TYPE_BOOLEAN;
00203         } else if (strcmp(type_str, "Number") == 0) {
```

```
00204             var_type = VAR_TYPE_NUMBER;
00205         } else if (strcmp(type_str, "Counter") == 0) {
00206             var_type = VAR_TYPE_COUNTER;
00207         } else if (strcmp(type_str, "Timer") == 0) {
00208             var_type = VAR_TYPE_TIMER;
00209         } else {
00210             var_type = VAR_TYPE_TIME;
00211         }
00212 
00213         void *data = NULL;
00214         switch (var_type) {
00215             case VAR_TYPE_DIGITAL_ANALOG_IO: {
00216                 DigitalAnalogInputOutput *daio = (DigitalAnalogInputOutput *)calloc(1,
    sizeof(DigitalAnalogInputOutput));
00217                 if (!daio) {
00218                     ESP_LOGE(TAG, "Memory allocation failure");
00219                     variables_list_free();
00220                     return false;
00221                 }
00222                 daio->base.name = strdup(name);
00223                 daio->base.type = strdup(type_str);
00224                 daio->pin_number = strdup(cJSON_GetObjectItem(var, "Pin")->valuestring);
00225                 data = daio;
00226                 break;
00227             }
00228             case VAR_TYPE_ONE_WIRE: {
00229                 OneWireInput *owi = (OneWireInput *)calloc(1, sizeof(OneWireInput));
00230                 if (!owi) {
00231                     ESP_LOGE(TAG, "Memory allocation failure");
00232                     variables_list_free();
00233                     return false;
00234                 }
00235                 owi->base.name = strdup(name);
00236                 owi->base.type = strdup(type_str);
00237                 owi->pin_number = strdup(cJSON_GetObjectItem(var, "Pin")->valuestring);
00238                 data = owi;
00239                 break;
00240             }
00241             case VAR_TYPE_ADC_SENSOR: {
00242                 ADCSensor *adcs = (ADCSensor *)calloc(1, sizeof(ADCSensor));
00243                 if (!adcs) {
00244                     ESP_LOGE(TAG, "Memory allocation failure");
00245                     variables_list_free();
00246                     return false;
00247                 }
00248                 adcs->base.name = strdup(name);
00249                 adcs->base.type = strdup(type_str);
00250                 adcs->sensor_type = strdup(cJSON_GetObjectItem(var, "Sensor Type")->valuestring);
00251                 adcs->pd_sck = strdup(cJSON_GetObjectItem(var, "PD_SCK")->valuestring);
00252                 adcs->dout = strdup(cJSON_GetObjectItem(var, "DOUT")->valuestring);
00253                 adcs->map_low = cJSON_GetObjectItem(var, "Map Low")->valuedouble;
00254                 adcs->map_high = cJSON_GetObjectItem(var, "Map High")->valuedouble;
00255                 adcs->gain = cJSON_GetObjectItem(var, "Gain")->valuedouble;
00256                 adcs->sampling_rate = strdup(cJSON_GetObjectItem(var, "Sampling Rate")->valuestring);
00257                 data = adcs;
00258 
00259                 // Initialize ADC sensor
00260                 esp_err_t ret = adc_sensor_init(adcs->sensor_type, adcs->pd_sck, adcs->dout);
00261                 if (ret != ESP_OK) {
00262                     ESP_LOGE(TAG, "Failed to initialize ADC Sensor '%s': %d", name, ret);
00263                     free_variable(VAR_TYPE_ADC_SENSOR, adcs);
00264                     continue; // Skip adding this sensor
00265                 }
00266                 break;
00267             }
00268             case VAR_TYPE_BOOLEAN: {
00269                 Boolean *b = (Boolean *)malloc(sizeof(Boolean));
00270                 if (!b) {
00271                     ESP_LOGE(TAG, "Memory allocation failure");
00272                     variables_list_free();
00273                     return false;
00274                 }
00275                 b->base.name = strdup(name);
00276                 b->base.type = strdup(type_str);
00277                 b->value = cJSON_IsTrue(cJSON_GetObjectItem(var, "Value"));
00278                 data = b;
00279                 break;
00280             }
00281             case VAR_TYPE_NUMBER: {
00282                 Number *n = (Number *)malloc(sizeof(Number));
00283                 if (!n) {
00284                     ESP_LOGE(TAG, "Memory allocation failure");
00285                     variables_list_free();
00286                     return false;
00287                 }
00288                 n->base.name = strdup(name);
00289                 n->base.type = strdup(type_str);
```

```
00290                 n->value = cJSON_GetObjectItem(var, "Value")->valuedouble;
00291                 data = n;
00292                 break;
00293             }
00294             case VAR_TYPE_COUNTER: {
00295                 Counter *c = (Counter *)malloc(sizeof(Counter));
00296                 if (!c) {
00297                     ESP_LOGE(TAG, "Memory allocation failure");
00298                     variables_list_free();
00299                     return false;
00300                 }
00301                 c->base.name = strdup(name);
00302                 c->base.type = strdup(type_str);
00303                 c->pv = cJSON_GetObjectItem(var, "PV")->valuedouble;
00304                 c->cv = cJSON_GetObjectItem(var, "CV")->valuedouble;
00305                 c->cu = cJSON_IsTrue(cJSON_GetObjectItem(var, "CU"));
00306                 c->cd = cJSON_IsTrue(cJSON_GetObjectItem(var, "CD"));
00307                 c->qu = cJSON_IsTrue(cJSON_GetObjectItem(var, "QU"));
00308                 c->qd = cJSON_IsTrue(cJSON_GetObjectItem(var, "QD"));
00309                 data = c;
00310                 break;
00311             }
00312             case VAR_TYPE_TIMER: {
00313                 Timer *t = (Timer *)malloc(sizeof(Timer));
00314                 if (!t) {
00315                     ESP_LOGE(TAG, "Memory allocation failure");
00316                     variables_list_free();
00317                     return false;
00318                 }
00319                 t->base.name = strdup(name);
00320                 t->base.type = strdup(type_str);
00321                 t->pt = cJSON_GetObjectItem(var, "PT")->valuedouble;
00322                 t->et = cJSON_GetObjectItem(var, "ET")->valuedouble;
00323                 t->in = cJSON_IsTrue(cJSON_GetObjectItem(var, "IN"));
00324                 t->q = cJSON_IsTrue(cJSON_GetObjectItem(var, "Q"));
00325                 data = t;
00326                 break;
00327             }
00328             case VAR_TYPE_TIME: {
00329                 Time *t = (Time *)malloc(sizeof(Time));
00330                 if (!t) {
00331                     ESP_LOGE(TAG, "Memory allocation failure");
00332                     variables_list_free();
00333                     return false;
00334                 }
00335                 t->base.name = strdup(name);
00336                 t->base.type = strdup(type_str);
00337                 t->value = cJSON_GetObjectItem(var, "Value")->valuedouble;
00338                 data = t;
00339                 break;
00340             }
00341         }
00342 
00343         if (!variables_list_add(var_type, data)) {
00344             free_variable(var_type, data);
00345             variables_list_free();
00346             return false;
00347         }
00348     }
00349 
00350     // Create one_wire_read_task only if needed
00351     bool has_one_wire = false;
00352     for (size_t i = 0; i < variables_list.count; i++) {
00353         if (variables_list.nodes[i].type == VAR_TYPE_ONE_WIRE) {
00354             has_one_wire = true;
00355             break;
00356         }
00357     }
00358     if (has_one_wire) {
00359         if (xTaskCreate(one_wire_read_task, "one_wire_read_task", 4096, NULL, 5,
      &one_wire_task_handle) != pdPASS) {
00360             ESP_LOGE(TAG, "Failed to create one_wire_read_task");
00361             variables_list_free();
00362             return false;
00363         }
00364         ESP_LOGI(TAG, "Created one_wire_read_task");
00365     }
00366 
00367     // Create adc_sensor_read_task only if needed
00368     bool has_adc_sensor = false;
00369     for (size_t i = 0; i < variables_list.count; i++) {
00370         if (variables_list.nodes[i].type == VAR_TYPE_ADC_SENSOR) {
00371             has_adc_sensor = true;
00372             break;
00373         }
00374     }
00375     if (has_adc_sensor) {
```

```
00376         if (xTaskCreate(adc_sensor_read_task, "adc_sensor_read_task", 4096, NULL, 5,
      &adc_sensor_task_handle) != pdPASS) {
00377             ESP_LOGE(TAG, "Failed to create adc_sensor_read_task");
00378             variables_list_free();
00379             return false;
00380         }
00381         ESP_LOGI(TAG, "Created adc_sensor_read_task");
00382     }
00383
00384     return true;
00385 }
00386
00387 VariableNode *find_variable(const char *search_name) {
00388     for (size_t i = 0; i < variables_list.count; i++) {
00389         VariableNode *node = &variables_list.nodes[i];
00390         Variable *base = NULL;
00391
00392         switch (node->type) {
00393             case VAR_TYPE_DIGITAL_ANALOG_IO: {
00394                 DigitalAnalogInputOutput *dio = (DigitalAnalogInputOutput *)node->data;
00395                 base = &dio->base;
00396                 break;
00397             }
00398             case VAR_TYPE_ONE_WIRE: {
00399                 OneWireInput *owi = (OneWireInput *)node->data;
00400                 base = &owi->base;
00401                 break;
00402             }
00403             case VAR_TYPE_ADC_SENSOR: {
00404                 ADCSensor *adcs = (ADCSensor *)node->data;
00405                 base = &adcs->base;
00406                 break;
00407             }
00408             case VAR_TYPE_BOOLEAN: {
00409                 Boolean *b = (Boolean *)node->data;
00410                 base = &b->base;
00411                 break;
00412             }
00413             case VAR_TYPE_NUMBER: {
00414                 Number *n = (Number *)node->data;
00415                 base = &n->base;
00416                 break;
00417             }
00418             case VAR_TYPE_TIME: {
00419                 Time *t = (Time *)node->data;
00420                 base = &t->base;
00421                 break;
00422             }
00423             case VAR_TYPE_COUNTER: {
00424                 Counter *c = (Counter *)node->data;
00425                 base = &c->base;
00426                 break;
00427             }
00428             case VAR_TYPE_TIMER: {
00429                 Timer *t = (Timer *)node->data;
00430                 base = &t->base;
00431                 break;
00432             }
00433         }
00434         if (base && strcmp(base->name, search_name) == 0)
00435             return node;
00436     }
00437     return NULL;
00438 }
00439
00440 VariableNode *find_current_time_variable(void) {
00441     for (size_t i = 0; i < variables_list.count; i++) {
00442         VariableNode *node = &variables_list.nodes[i];
00443         if (node->type == VAR_TYPE_TIME) {
00444             Time *t = (Time *)node->data;
00445             if (strcmp(t->base.type, "Current Time") == 0) {
00446                 return node;
00447             }
00448         }
00449     }
00450     return NULL;
00451 }
00452
00460 void parse_variable_name(const char *var_name, char *base_name, size_t base_name_size, const char
      **suffix) {
00461     *suffix = NULL;
00462     base_name[0] = '\0';
00463
00464     if (strchr(var_name, '.')) {
00465         const char *dot = strrchr(var_name, '.');
00466         if (strcmp(dot, ".CU") == 0 || strcmp(dot, ".CD") == 0 ||
00467                 strcmp(dot, ".QU") == 0 || strcmp(dot, ".QD") == 0 ||
```

```
00468                 strcmp(dot, ".IN") == 0 || strcmp(dot, ".Q") == 0 ||
00469                 strcmp(dot, ".PV") == 0 || strcmp(dot, ".CV") == 0 ||
00470                 strcmp(dot, ".PT") == 0 || strcmp(dot, ".ET") == 0) {
00471             *suffix = dot;
00472             size_t base_len = dot - var_name;
00473             strncpy(base_name, var_name, base_len);
00474             base_name[base_len] = '\0';
00475         }
00476     }
00477
00478     if (!*suffix) {
00479         strncpy(base_name, var_name, base_name_size - 1);
00480         base_name[base_name_size - 1] = '\0';
00481     }
00482 }
00483
00484 bool read_variable(const char *var_name) {
00485     char base_name[MAX_VAR_NAME_LENGTH];
00486     const char *variable_parameter;
00487     parse_variable_name(var_name, base_name, sizeof(base_name), &variable_parameter);
00488     VariableNode *node = find_variable(base_name);
00489
00490     switch (node->type) {
00491         case VAR_TYPE_DIGITAL_ANALOG_IO: {
00492             DigitalAnalogInputOutput *dio = (DigitalAnalogInputOutput *)node->data;
00493             Variable *base = &dio->base;
00494             if (strcmp(base->type, "Digital Input") == 0)
00495                 return get_digital_input_value(dio->pin_number);
00496             else if (strcmp(base->type, "Digital Output") == 0)
00497                 return get_digital_output_value(dio->pin_number);
00498             break;
00499         }
00500         case VAR_TYPE_BOOLEAN: {
00501             Boolean *b = (Boolean *)node->data;
00502             return b->value;
00503             break;
00504         }
00505         case VAR_TYPE_COUNTER: {
00506             Counter *c = (Counter *)node->data;
00507             if (strcmp(variable_parameter, ".CU") == 0) return c->cu;
00508             else if (strcmp(variable_parameter, ".CD") == 0) return c->cd;
00509             else if (strcmp(variable_parameter, ".QU") == 0) return c->qu;
00510             else if (strcmp(variable_parameter, ".QD") == 0) return c->qd;
00511             break;
00512         }
00513         case VAR_TYPE_TIMER: {
00514             Timer *t = (Timer *)node->data;
00515             if (strcmp(variable_parameter, ".IN") == 0) return t->in;
00516             else if (strcmp(variable_parameter, ".Q") == 0) return t->q;
00517             break;
00518         }
00519         default:
00520             break;
00521     }
00522     return false;
00523 }
00524
00525 void write_variable(const char *var_name, bool value) {
00526     char base_name[MAX_VAR_NAME_LENGTH];
00527     const char *variable_parameter;
00528     parse_variable_name(var_name, base_name, sizeof(base_name), &variable_parameter);
00529     VariableNode *node = find_variable(base_name);
00530
00531     switch (node->type) {
00532         case VAR_TYPE_DIGITAL_ANALOG_IO: {
00533             DigitalAnalogInputOutput *dio = (DigitalAnalogInputOutput *)node->data;
00534             set_digital_output_value(dio->pin_number, value);
00535             return;
00536         }
00537         case VAR_TYPE_BOOLEAN: {
00538             Boolean *b = (Boolean *)node->data;
00539             b->value = value;
00540             return;
00541         }
00542         case VAR_TYPE_COUNTER: {
00543             Counter *c = (Counter *)node->data;
00544             if (strcmp(variable_parameter, ".CU") == 0) {
00545                 c->cu = value;
00546                 return;
00547             } else if (strcmp(variable_parameter, ".CD") == 0) {
00548                 c->cd = value;
00549                 return;
00550             } else if (strcmp(variable_parameter, ".QU") == 0) {
00551                 c->qu = value;
00552                 return;
00553             } else if (strcmp(variable_parameter, ".QD") == 0) {
00554                 c->qd = value;
```

```
00555                 return;
00556             }
00557             break;
00558         }
00559         case VAR_TYPE_TIMER: {
00560             Timer *t = (Timer *)node->data;
00561             if (strcmp(variable_parameter, ".IN") == 0) {
00562                 t->in = value;
00563                 return;
00564             } else if (strcmp(variable_parameter, ".Q") == 0) {
00565                 t->q = value;
00566                 return;
00567             }
00568             break;
00569         }
00570         default:
00571             break;
00572     }
00573 }
00574
00575 double read_numeric_variable(const char *var_name) {
00576     char base_name[MAX_VAR_NAME_LENGTH];
00577     const char *variable_parameter;
00578     parse_variable_name(var_name, base_name, sizeof(base_name), &variable_parameter);
00579     VariableNode *node = find_variable(base_name);
00580
00581     switch (node->type) {
00582         case VAR_TYPE_DIGITAL_ANALOG_IO: {
00583             DigitalAnalogInputOutput *dio = (DigitalAnalogInputOutput *)node->data;
00584             Variable *base = &dio->base;
00585             if (strcmp(base->type, "Analog Input") == 0)
00586                 return get_digital_input_value(dio->pin_number);
00587             else if (strcmp(base->type, "Analog Output") == 0)
00588                 return get_analog_output_value(dio->pin_number);
00589             break;
00590         }
00591         case VAR_TYPE_ONE_WIRE: {
00592             OneWireInput *owi = (OneWireInput *)node->data;
00593             return owi->value;
00594         }
00595         case VAR_TYPE_ADC_SENSOR: {
00596             ADCSensor *adcs = (ADCSensor *)node->data;
00597             return adcs->value;
00598         }
00599         case VAR_TYPE_NUMBER: {
00600             Number *n = (Number *)node->data;
00601             return n->value;
00602         }
00603         case VAR_TYPE_TIME: {
00604             Time *t = (Time *)node->data;
00605             return t->value;
00606         }
00607         case VAR_TYPE_COUNTER: {
00608             Counter *c = (Counter *)node->data;
00609             if (strcmp(variable_parameter, ".PV") == 0)
00610                 return c->pv;
00611             else if (strcmp(variable_parameter, ".CV") == 0)
00612                 return c->cv;
00613             break;
00614         }
00615         case VAR_TYPE_TIMER: {
00616             Timer *t = (Timer *)node->data;
00617             if (strcmp(variable_parameter, ".PT") == 0) {
00618                 return t->pt;
00619             } else if (strcmp(variable_parameter, ".ET") == 0) {
00620                 return t->et;
00621             }
00622         }
00623         default:
00624             break;
00625     }
00626     return 0;
00627 }
00628
00629 void write_numeric_variable(const char *var_name, double value) {
00630     char base_name[MAX_VAR_NAME_LENGTH];
00631     const char *variable_parameter;
00632     parse_variable_name(var_name, base_name, sizeof(base_name), &variable_parameter);
00633     VariableNode *node = find_variable(base_name);
00634
00635     switch (node->type) {
00636         case VAR_TYPE_DIGITAL_ANALOG_IO: {
00637             DigitalAnalogInputOutput *dio = (DigitalAnalogInputOutput *)node->data;
00638             int int_value = (int)round(value);
00639             uint8_t scaled_value = int_value < 0 ? 0 : (int_value > 255 ? 255 : (uint8_t)int_value);
00640             set_analog_output_value(dio->pin_number, scaled_value);
00641             break;
```

```
00642         }
00643         case VAR_TYPE_NUMBER: {
00644             Number *n = (Number *)node->data;
00645             n->value = value;
00646             break;
00647         }
00648         case VAR_TYPE_TIME: {
00649             Time *t = (Time *)node->data;
00650             t->value = value;
00651             break;
00652         }
00653         case VAR_TYPE_COUNTER: {
00654             Counter *c = (Counter *)node->data;
00655             if (strcmp(variable_parameter, ".PV") == 0) c->pv = value;
00656             else if (strcmp(variable_parameter, ".CV") == 0) c->cv = value;
00657             break;
00658         }
00659         case VAR_TYPE_TIMER: {
00660             Timer *t = (Timer *)node->data;
00661             if (strcmp(variable_parameter, ".PT") == 0) t->pt = value;
00662             else if (strcmp(variable_parameter, ".ET") == 0) t->et = value;
00663             break;
00664         }
00665         default:
00666             break;
00667     }
00668 }
00669
00674 static void one_wire_read_task(void *pvParameters)
00675 {
00676     while (1)
00677     {
00678         for (size_t i = 0; i < variables_list.count; i++)
00679         {
00680             VariableNode *node = &variables_list.nodes[i];
00681             if (node->type == VAR_TYPE_ONE_WIRE)
00682             {
00683                 OneWireInput *owi = (OneWireInput *)node->data;
00684                 owi->value = get_one_wire_value(owi->pin_number);
00685                 vTaskDelay(pdMS_TO_TICKS(1000)); // 1 second after each read
00686             }
00687         }
00688         vTaskDelay(pdMS_TO_TICKS(1000)); // 1 second at end
00689     }
00690 }
00691
00696 static void adc_sensor_read_task(void *pvParameters)
00697 {
00698     while (1)
00699     {
00700         for (size_t i = 0; i < variables_list.count; i++)
00701         {
00702             VariableNode *node = &variables_list.nodes[i];
00703             if (node->type == VAR_TYPE_ADC_SENSOR)
00704             {
00705                 ADCSensor *adcs = (ADCSensor *)node->data;
00706                 double value = adc_sensor_read(adcs->sensor_type, adcs->pd_sck, adcs->dout,
00707                                                adcs->map_low, adcs->map_high, adcs->gain,
00708                                                adcs->sampling_rate, adcs->base.name);
00709                 if (value != 0.0 || adcs->value == 0.0) { // Update only if value is valid or previous
    was 0
00710                     adcs->value = value;
00711                     // ESP_LOGI(TAG, "ADC Sensor '%s' value: %f", adcs->base.name, adcs->value);
00712                 } else {
00713                     ESP_LOGW(TAG, "Invalid value for ADC Sensor '%s', keeping old: %f",
    adcs->base.name, adcs->value);
00714                 }
00715                 // Adjust delay based on sampling_rate
00716                 int delay_ms = (strcmp(adcs->sampling_rate, "10Hz") == 0) ? 150 : 100;
00717                 vTaskDelay(pdMS_TO_TICKS(delay_ms));
00718             }
00719         }
00720         vTaskDelay(pdMS_TO_TICKS(1000)); // 1 second between cycles
00721     }
00722 }
00723
00728 char *read_variables_json(void) {
00729     // Create JSON array for all variables
00730     cJSON *variables_array = cJSON_CreateArray();
00731     if (!variables_array) {
00732         ESP_LOGE(TAG, "Failed to create JSON array");
00733         return NULL;
00734     }
00735
00736     // Iterate through all variables in the list
00737     for (size_t i = 0; i < variables_list.count; i++) {
00738         VariableNode *node = &variables_list.nodes[i];
```

```
00739         cJSON *var_json = cJSON_CreateObject();
00740         if (!var_json) {
00741             ESP_LOGE(TAG, "Failed to create JSON object for variable");
00742             continue;
00743         }
00744 
00745         Variable *base = NULL;
00746         switch (node->type) {
00747             case VAR_TYPE_DIGITAL_ANALOG_IO: {
00748                 DigitalAnalogInputOutput *dio = (DigitalAnalogInputOutput *)node->data;
00749                 base = &dio->base;
00750                 cJSON_AddStringToObject(var_json, "Type", base->type);
00751                 cJSON_AddStringToObject(var_json, "Name", base->name);
00752                 cJSON_AddStringToObject(var_json, "Pin", dio->pin_number);
00753                 if (strcmp(base->type, "Digital Input") == 0 || strcmp(base->type, "Digital Output")
      == 0)
00754                     cJSON_AddNumberToObject(var_json, "Value", read_variable(base->name));
00755                 else if (strcmp(base->type, "Analog Input") == 0 || strcmp(base->type, "Analog
      Output") == 0)
00756                     cJSON_AddNumberToObject(var_json, "Value", read_numeric_variable(base->name));
00757                 break;
00758             }
00759             case VAR_TYPE_ONE_WIRE: {
00760                 OneWireInput *owi = (OneWireInput *)node->data;
00761                 base = &owi->base;
00762                 cJSON_AddStringToObject(var_json, "Type", base->type);
00763                 cJSON_AddStringToObject(var_json, "Name", base->name);
00764                 cJSON_AddStringToObject(var_json, "Pin", owi->pin_number);
00765                 cJSON_AddNumberToObject(var_json, "Value", owi->value);
00766                 break;
00767             }
00768             case VAR_TYPE_ADC_SENSOR: {
00769                 ADCSensor *adcs = (ADCSensor *)node->data;
00770                 base = &adcs->base;
00771                 cJSON_AddStringToObject(var_json, "Type", base->type);
00772                 cJSON_AddStringToObject(var_json, "Name", base->name);
00773                 cJSON_AddStringToObject(var_json, "SensorType", adcs->sensor_type);
00774                 cJSON_AddStringToObject(var_json, "PD_SCK", adcs->pd_sck);
00775                 cJSON_AddStringToObject(var_json, "DOUT", adcs->dout);
00776                 cJSON_AddNumberToObject(var_json, "MapLow", adcs->map_low);
00777                 cJSON_AddNumberToObject(var_json, "MapHigh", adcs->map_high);
00778                 cJSON_AddNumberToObject(var_json, "Gain", adcs->gain);
00779                 cJSON_AddStringToObject(var_json, "SamplingRate", adcs->sampling_rate);
00780                 cJSON_AddNumberToObject(var_json, "Value", adcs->value);
00781                 break;
00782             }
00783             case VAR_TYPE_BOOLEAN: {
00784                 Boolean *b = (Boolean *)node->data;
00785                 base = &b->base;
00786                 cJSON_AddStringToObject(var_json, "Type", base->type);
00787                 cJSON_AddStringToObject(var_json, "Name", base->name);
00788                 cJSON_AddBoolToObject(var_json, "Value", b->value);
00789                 break;
00790             }
00791             case VAR_TYPE_NUMBER: {
00792                 Number *n = (Number *)node->data;
00793                 base = &n->base;
00794                 cJSON_AddStringToObject(var_json, "Type", base->type);
00795                 cJSON_AddStringToObject(var_json, "Name", base->name);
00796                 cJSON_AddNumberToObject(var_json, "Value", n->value);
00797                 break;
00798             }
00799             case VAR_TYPE_COUNTER: {
00800                 Counter *c = (Counter *)node->data;
00801                 base = &c->base;
00802                 cJSON_AddStringToObject(var_json, "Type", base->type);
00803                 cJSON_AddStringToObject(var_json, "Name", base->name);
00804                 cJSON_AddNumberToObject(var_json, "PV", c->pv);
00805                 cJSON_AddNumberToObject(var_json, "CV", c->cv);
00806                 cJSON_AddBoolToObject(var_json, "CU", c->cu);
00807                 cJSON_AddBoolToObject(var_json, "CD", c->cd);
00808                 cJSON_AddBoolToObject(var_json, "QU", c->qu);
00809                 cJSON_AddBoolToObject(var_json, "QD", c->qd);
00810                 break;
00811             }
00812             case VAR_TYPE_TIMER: {
00813                 Timer *t = (Timer *)node->data;
00814                 base = &t->base;
00815                 cJSON_AddStringToObject(var_json, "Type", base->type);
00816                 cJSON_AddStringToObject(var_json, "Name", base->name);
00817                 cJSON_AddNumberToObject(var_json, "PT", t->pt);
00818                 cJSON_AddNumberToObject(var_json, "ET", t->et);
00819                 cJSON_AddBoolToObject(var_json, "IN", t->in);
00820                 cJSON_AddBoolToObject(var_json, "Q", t->q);
00821                 break;
00822             }
00823             case VAR_TYPE_TIME: {
```

```
00824                 Time *t = (Time *)node->data;
00825                 base = &t->base;
00826                 cJSON_AddStringToObject(var_json, "Type", base->type);
00827                 cJSON_AddStringToObject(var_json, "Name", base->name);
00828                 cJSON_AddNumberToObject(var_json, "Value", t->value);
00829                 break;
00830             }
00831         }
00832
00833         cJSON_AddItemToArray(variables_array, var_json);
00834     }
00835
00836     // Convert JSON to string
00837     char *json_str = cJSON_PrintUnformatted(variables_array);
00838     if (!json_str) {
00839         ESP_LOGE(TAG, "Failed to print JSON");
00840         cJSON_Delete(variables_array);
00841         return NULL;
00842     }
00843
00844     // Free JSON array
00845     cJSON_Delete(variables_array);
00846
00847     // Return JSON string
00848     return json_str;
00849 }
00850
00851 void update_variables_from_children(const char *json_str) {
00852     // Parse JSON string
00853     cJSON *json = cJSON_Parse(json_str);
00854     if (!json) {
00855         ESP_LOGE(TAG, "Failed to parse JSON: %s", cJSON_GetErrorPtr());
00856         return;
00857     }
00858
00859     // Iterate through variables in the global list
00860     for (size_t i = 0; i < variables_list.count; i++) {
00861         VariableNode *node = &variables_list.nodes[i];
00862         Variable *base = NULL;
00863
00864         // Process only Boolean and Number variables
00865         if (node->type == VAR_TYPE_BOOLEAN) {
00866             Boolean *b = (Boolean *)node->data;
00867             base = &b->base;
00868
00869             // Find matching field in JSON
00870             cJSON *json_item = cJSON_GetObjectItem(json, base->name);
00871             if (json_item && cJSON_IsBool(json_item)) {
00872                 b->value = cJSON_IsTrue(json_item);
00873                 //ESP_LOGI(TAG, "Updated Boolean variable '%s' to %s", base->name, b->value ? "true" :
      "false");
00874             }
00875         }
00876         else if (node->type == VAR_TYPE_NUMBER) {
00877             Number *n = (Number *)node->data;
00878             base = &n->base;
00879
00880             // Find matching field in JSON
00881             cJSON *json_item = cJSON_GetObjectItem(json, base->name);
00882             if (json_item && cJSON_IsNumber(json_item)) {
00883                 n->value = json_item->valuedouble;
00884                 //ESP_LOGI(TAG, "Updated Number variable '%s' to %f", base->name, n->value);
00885             }
00886         }
00887     }
00888
00889     // Free JSON memory
00890     cJSON_Delete(json);
00891 }
00892
00893 void send_variables_to_parents() {
00894     // Create JSON object for Boolean and Number variables
00895     cJSON *variables_json = cJSON_CreateObject();
00896     if (!variables_json) {
00897         ESP_LOGE(TAG, "Failed to create JSON object");
00898         return;
00899     }
00900
00901     // Iterate through variables in the global list
00902     for (size_t i = 0; i < variables_list.count; i++) {
00903         VariableNode *node = &variables_list.nodes[i];
00904         Variable *base = NULL;
00905
00906         // Process only Boolean and Number variables
00907         if (node->type == VAR_TYPE_BOOLEAN) {
00908             Boolean *b = (Boolean *)node->data;
00909             base = &b->base;
```

```
00910             cJSON_AddBoolToObject(variables_json, base->name, b->value);
00911         }
00912         else if (node->type == VAR_TYPE_NUMBER) {
00913             Number *n = (Number *)node->data;
00914             base = &n->base;
00915             cJSON_AddNumberToObject(variables_json, base->name, n->value);
00916         }
00917     }
00918
00919     // Convert JSON to string
00920     char *json_str = cJSON_PrintUnformatted(variables_json);
00921     if (!json_str) {
00922         ESP_LOGE(TAG, "Failed to print JSON");
00923         cJSON_Delete(variables_json);
00924         return;
00925     }
00926
00927     // Iterate through parent devices and send JSON to each topic
00928     for (size_t i = 0; i < _device.parent_devices_len; i++) {
00929         if (_device.parent_devices[i]) {
00930             // Form topic: parent_devices[i] + TOPIC_CHILDREN_LISTENER
00931             size_t topic_len = strlen(_device.parent_devices[i]) + strlen(TOPIC_CHILDREN_LISTENER) +
      1;
00932             char *topic = (char *)malloc(topic_len);
00933             if (!topic) {
00934                 ESP_LOGE(TAG, "Failed to allocate memory for topic");
00935                 continue;
00936             }
00937             snprintf(topic, topic_len, "%s%s", _device.parent_devices[i], TOPIC_CHILDREN_LISTENER);
00938
00939             // Publish JSON string to MQTT topic
00940             mqtt_publish(json_str, topic, 0);
00941             //ESP_LOGI(TAG, "Sent variables to parent '%s' on topic '%s'", _device.parent_devices[i],
      topic);
00942
00943             // Free topic memory
00944             free(topic);
00945         }
00946     }
00947
00948     // Free memory
00949     cJSON_Delete(variables_json);
00950     free(json_str);
00951 }
```

4.51 main/variables.h File Reference

```
#include <stdbool.h>
#include <stddef.h>
#include <cJSON.h>
```

Data Structures

- struct Variable

  Base structure for a variable.
- struct DigitalAnalogInputOutput

  Structure for digital/analog input/output variables.
- struct OneWireInput

  Structure for one-wire input variables.
- struct ADCSensor

  Structure for ADC sensor variables.
- struct Boolean

  Structure for boolean variables.
- struct Number

  Structure for numeric variables.
- struct Counter

Structure for counter variables.

- struct Timer

  Structure for timer variables.
- struct Time

  Structure for time variables.
- struct VariableNode

  Structure for a variable node in the variables list.
- struct VariablesList

  Structure for managing a list of variables.

Macros

- #define MAX_VAR_NAME_LENGTH 64

  Maximum length for variable names.

Enumerations

- enum VariableType {
  VAR_TYPE_DIGITAL_ANALOG_IO , VAR_TYPE_ONE_WIRE , VAR_TYPE_ADC_SENSOR , VAR_TYPE_BOOLEAN
  ,
  VAR_TYPE_NUMBER , VAR_TYPE_COUNTER , VAR_TYPE_TIMER , VAR_TYPE_TIME }

  Enum for different variable types.

Functions

- bool load_variables (cJSON *variables)

  Load variables from a cJSON object.
- VariableNode * find_variable (const char *search_name)

  Find a variable by name.
- VariableNode * find_current_time_variable (void)

  Find the current time variable.
- bool read_variable (const char *var_name)

  Read a boolean variable by name.
- void write_variable (const char *var_name, bool value)

  Write a boolean value to a variable.
- double read_numeric_variable (const char *var_name)

  Read a numeric variable by name.
- void write_numeric_variable (const char *var_name, double value)

  Write a numeric value to a variable.
- char * read_variables_json (void)

  Read all variables as a JSON string.
- void update_variables_from_children (const char *json_str)

  Update variables from a JSON string received from child nodes.
- void send_variables_to_parents (void)

  Send variable updates to parent nodes.

Variables

- VariablesList variables_list

  Global list of variables.

4.51.1 Macro Definition Documentation

4.51.1.1 MAX_VAR_NAME_LENGTH

```
#define MAX_VAR_NAME_LENGTH 64
```

Maximum length for variable names.

Definition at line 11 of file variables.h.

4.51.2 Enumeration Type Documentation

4.51.2.1 VariableType

```
enum VariableType
```

Enum for different variable types.

Enumerator

VAR_TYPE_DIGITAL_ANALOG_IO	Digital or analog input/output.
VAR_TYPE_ONE_WIRE	One-wire input.
VAR_TYPE_ADC_SENSOR	ADC sensor.
VAR_TYPE_BOOLEAN	Boolean variable.
VAR_TYPE_NUMBER	Numeric variable.
VAR_TYPE_COUNTER	Counter variable.
VAR_TYPE_TIMER	Timer variable.
VAR_TYPE_TIME	Time variable.

Definition at line 16 of file variables.h.

4.51.3 Function Documentation

4.51.3.1 find_current_time_variable()

```
VariableNode * find_current_time_variable (
            void )
```

Find the current time variable.

Returns

VariableNode* Pointer to the time variable node, or NULL if not found.

Definition at line 440 of file variables.c.

4.51.3.2 find_variable()

```
VariableNode * find_variable (
            const char * search_name)
```

Find a variable by name.

Parameters

search_name	Name of the variable to find.

Returns

VariableNode* Pointer to the variable node, or NULL if not found.

Definition at line 387 of file variables.c.

4.51.3.3 load_variables()

```
bool load_variables (
            cJSON * variables)
```

Load variables from a cJSON object.

Parameters

variables	cJSON object containing variable definitions.

Returns

bool True if loading succeeds, false otherwise.

Definition at line 172 of file variables.c.

4.51.3.4 read_numeric_variable()

```
double read_numeric_variable (
            const char * var_name)
```

Read a numeric variable by name.

Parameters

var_name	Name of the variable to read.

Returns

double The value of the variable, or 0.0 if not found.

Definition at line 575 of file variables.c.

4.51.3.5 read_variable()

```
bool read_variable (
            const char * var_name)
```

Read a boolean variable by name.

Parameters

var_name	Name of the variable to read.

Returns

bool The value of the variable, or false if not found.

Definition at line 484 of file variables.c.

4.51.3.6 read_variables_json()

```
char * read_variables_json (
            void )
```

Read all variables as a JSON string.

Returns

char* JSON string containing all variables, or NULL on error.

Definition at line 728 of file variables.c.

4.51.3.7 send_variables_to_parents()

```
void send_variables_to_parents (
            void )
```

Send variable updates to parent nodes.

Definition at line 893 of file variables.c.

4.51.3.8 update_variables_from_children()

```
void update_variables_from_children (
            const char * json_str)
```

Update variables from a JSON string received from child nodes.

Parameters

json_str	JSON string containing variable updates.

Definition at line 851 of file variables.c.

4.51.3.9 write_numeric_variable()

```
void write_numeric_variable (
            const char * var_name,
            double value)
```

Write a numeric value to a variable.

Parameters

var_name	Name of the variable to write to.
value	Numeric value to write.

Definition at line 629 of file variables.c.

4.51.3.10 write_variable()

```
void write_variable (
            const char * var_name,
            bool value)
```

Write a boolean value to a variable.

Parameters

var_name	Name of the variable to write to.
value	Boolean value to write.

Definition at line 525 of file variables.c.

4.51.4 Variable Documentation

4.51.4.1 variables_list

```
VariablesList variables_list  [extern]
```

Global list of variables.

Definition at line 22 of file variables.c.

4.52 variables.h

Go to the documentation of this file.

```
00001 #ifndef VARIABLES_H
00002 #define VARIABLES_H
00003
00004 #include <stdbool.h>
00005 #include <stddef.h>
00006 #include <cJSON.h>
00007
00011 #define MAX_VAR_NAME_LENGTH 64
00012
00016 typedef enum {
00017     VAR_TYPE_DIGITAL_ANALOG_IO,
00018     VAR_TYPE_ONE_WIRE,
00019     VAR_TYPE_ADC_SENSOR,
00020     VAR_TYPE_BOOLEAN,
00021     VAR_TYPE_NUMBER,
00022     VAR_TYPE_COUNTER,
00023     VAR_TYPE_TIMER,
00024     VAR_TYPE_TIME
00025 } VariableType;
00026
00030 typedef struct {
```

```
00031     char *name;
00032     char *type;
00033 } Variable;
00034
00038 typedef struct {
00039     Variable base;
00040     char *pin_number;
00041 } DigitalAnalogInputOutput;
00042
00046 typedef struct {
00047     Variable base;
00048     char *pin_number;
00049     double value;
00050 } OneWireInput;
00051
00055 typedef struct {
00056     Variable base;
00057     char *sensor_type;
00058     char *pd_sck;
00059     char *dout;
00060     double map_low;
00061     double map_high;
00062     double gain;
00063     char *sampling_rate;
00064     double value;
00065 } ADCSensor;
00066
00070 typedef struct {
00071     Variable base;
00072     bool value;
00073 } Boolean;
00074
00078 typedef struct {
00079     Variable base;
00080     double value;
00081 } Number;
00082
00086 typedef struct {
00087     Variable base;
00088     double pv;
00089     double cv;
00090     bool cu;
00091     bool cd;
00092     bool qu;
00093     bool qd;
00094 } Counter;
00095
00099 typedef struct {
00100     Variable base;
00101     double pt;
00102     double et;
00103     bool in;
00104     bool q;
00105 } Timer;
00106
00110 typedef struct {
00111     Variable base;
00112     double value;
00113 } Time;
00114
00118 typedef struct {
00119     VariableType type;
00120     void *data;
00121 } VariableNode;
00122
00126 typedef struct {
00127     VariableNode *nodes;
00128     size_t count;
00129     size_t capacity;
00130 } VariablesList;
00131
00135 extern VariablesList variables_list;
00136
00142 bool load_variables(cJSON *variables);
00143
00149 VariableNode *find_variable(const char *search_name);
00150
00155 VariableNode *find_current_time_variable(void);
00156
00162 bool read_variable(const char *var_name);
00163
00169 void write_variable(const char *var_name, bool value);
00170
00176 double read_numeric_variable(const char *var_name);
00177
00183 void write_numeric_variable(const char *var_name, double value);
00184
```

```
00189 char *read_variables_json(void);
00190
00195 void update_variables_from_children(const char *json_str);
00196
00200 void send_variables_to_parents(void);
00201
00202 #endif // VARIABLES_H
```

4.53 main/wifi.c File Reference

```
#include "freertos/FreeRTOS.h"
#include "freertos/event_groups.h"
#include "esp_wifi.h"
#include "esp_event.h"
#include "esp_log.h"
#include "string.h"
#include "wifi.h"
#include "ntp.h"
#include "mqtt.h"
```

Functions

- static void wifi_event_handler (void *arg, esp_event_base_t event_base, int32_t event_id, void *event_data)

  WiFi event handler to manage connection states.
- void wifi_init (void)

  Initialize the WiFi module and start the connection process.
- EventGroupHandle_t wifi_get_event_group (void)

  Get the handle to the WiFi event group.
- void wifi_stop (void)

  Stop the WiFi module and disconnect.
- bool wifi_is_connected (void)

  Check if the WiFi is currently connected.

Variables

- static EventGroupHandle_t wifi_event_group

  Handle for the WiFi event group.
- static const char * TAG = "wifi_module"

  Tag for logging messages from the WiFi module.
- static int retry_count = 0

  Counter for reconnection attempts.
- static esp_event_handler_instance_t instance_any_id = NULL

  Instance for handling any WiFi event.
- static esp_event_handler_instance_t instance_got_ip = NULL

  Instance for handling IP event when STA gets an IP.

4.53.1 Function Documentation

4.53.1.1 wifi_event_handler()

```
static void wifi_event_handler (
            void * arg,
            esp_event_base_t event_base,
            int32_t event_id,
            void * event_data)  [static]
```

WiFi event handler to manage connection states.

Parameters

arg	User data (unused).
event_base	Event base (WIFI_EVENT or IP_EVENT).
event_id	Specific event ID.
event_data	Event data (unused).

Definition at line 44 of file wifi.c.

4.53.1.2 wifi_get_event_group()

```
EventGroupHandle_t wifi_get_event_group (
            void )
```

Get the handle to the WiFi event group.

Returns

EventGroupHandle_t Handle to the WiFi event group.

Definition at line 128 of file wifi.c.

4.53.1.3 wifi_init()

```
void wifi_init (
            void )
```

Initialize the WiFi module and start the connection process.

Definition at line 88 of file wifi.c.

4.53.1.4 wifi_is_connected()

```
bool wifi_is_connected (
            void )
```

Check if the WiFi is currently connected.

Returns

bool True if connected, false otherwise.

Definition at line 168 of file wifi.c.

4.53.1.5 wifi_stop()

```
void wifi_stop (
            void )
```

Stop the WiFi module and disconnect.

Definition at line 136 of file wifi.c.

4.53.2 Variable Documentation

4.53.2.1 instance_any_id

```
esp_event_handler_instance_t instance_any_id = NULL  [static]
```

Instance for handling any WiFi event.

Definition at line 30 of file wifi.c.

4.53.2.2 instance_got_ip

```
esp_event_handler_instance_t instance_got_ip = NULL  [static]
```

Instance for handling IP event when STA gets an IP.

Definition at line 35 of file wifi.c.

4.53.2.3 retry_count

```
int retry_count = 0  [static]
```

Counter for reconnection attempts.

Definition at line 25 of file wifi.c.

4.53.2.4 TAG

```
const char* TAG = "wifi_module"  [static]
```

Tag for logging messages from the WiFi module.

Definition at line 20 of file wifi.c.

4.53.2.5 wifi_event_group

```
EventGroupHandle_t wifi_event_group  [static]
```

Handle for the WiFi event group.

Definition at line 15 of file wifi.c.

4.54 wifi.c

Go to the documentation of this file.

```
00001 #include "freertos/FreeRTOS.h"
00002 #include "freertos/event_groups.h"
00003 #include "esp_wifi.h"
00004 #include "esp_event.h"
00005 #include "esp_log.h"
00006 #include "string.h"
00007 #include "wifi.h"
00008
00009 #include "ntp.h"
00010 #include "mqtt.h"
00011
00015 static EventGroupHandle_t wifi_event_group;
00016
00020 static const char *TAG = "wifi_module";
00021
00025 static int retry_count = 0;
00026
00030 static esp_event_handler_instance_t instance_any_id = NULL;
00031
00035 static esp_event_handler_instance_t instance_got_ip = NULL;
00036
00044 static void wifi_event_handler(void *arg, esp_event_base_t event_base, int32_t event_id, void
      *event_data)
00045 {
00046     // Check if the event group is valid
00047     if (wifi_event_group == NULL) {
00048         ESP_LOGW(TAG, "Event group is null, ignoring event");
00049         return;
00050     }
00051
00052     if (event_base == WIFI_EVENT && event_id == WIFI_EVENT_STA_START) {
00053         esp_wifi_connect();
00054     } else if (event_base == WIFI_EVENT && event_id == WIFI_EVENT_STA_DISCONNECTED) {
00055         vTaskDelay(pdMS_TO_TICKS(5000));
00056         esp_wifi_connect();
00057         xEventGroupClearBits(wifi_event_group, WIFI_CONNECTED_BIT);
00058
00059         if (MAX_RETRY_COUNT == 0) {
00060             // Infinite retry logic
00061             vTaskDelay(pdMS_TO_TICKS(WIFI_TIMEOUT_MS));
00062             esp_wifi_connect();
00063             ESP_LOGI(TAG, "Retrying WiFi connection");
00064         } else if (retry_count < MAX_RETRY_COUNT) {
00065             // Limited retry logic
00066             vTaskDelay(pdMS_TO_TICKS(WIFI_TIMEOUT_MS));
00067             esp_wifi_connect();
00068             retry_count++;
00069             ESP_LOGI(TAG, "Retry to connect to the AP (%d/%d)", retry_count, MAX_RETRY_COUNT);
00070         } else {
00071             // Max retry limit reached
00072             xEventGroupSetBits(wifi_event_group, WIFI_FAIL_BIT);
00073             ESP_LOGE(TAG, "Failed to connect after %d retries", MAX_RETRY_COUNT);
00074         }
00075     } else if (event_base == IP_EVENT && event_id == IP_EVENT_STA_GOT_IP) {
00076         retry_count = 0;  // Reset counter on successful connection
00077         xEventGroupSetBits(wifi_event_group, WIFI_CONNECTED_BIT);
00078         // Initialize NTP
00079         obtain_time();
00080         // Initialize MQTT
00081         mqtt_init();
00082     }
00083 }
00084
00088 void wifi_init(void)
00089 {
00090     wifi_event_group = xEventGroupCreate();
00091
00092     esp_netif_init();
00093     esp_event_loop_create_default();
00094     esp_netif_create_default_wifi_sta();
00095
00096     wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
00097     esp_wifi_init(&cfg);
00098
00099     esp_event_handler_instance_register(WIFI_EVENT, ESP_EVENT_ANY_ID, &wifi_event_handler, NULL,
      &instance_any_id);
00100     esp_event_handler_instance_register(IP_EVENT, IP_EVENT_STA_GOT_IP, &wifi_event_handler, NULL,
      &instance_got_ip);
00101
00102     wifi_config_t wifi_config = {
00103         .sta = {
00104             .ssid = {0},
```

```
00105             .password = {0},
00106         },
00107     };
00108     strncpy((char *)wifi_config.sta.ssid, WIFI_SSID, sizeof(wifi_config.sta.ssid) - 1);
00109     strncpy((char *)wifi_config.sta.password, WIFI_PASS, sizeof(wifi_config.sta.password) - 1);
00110
00111     esp_wifi_set_mode(WIFI_MODE_STA);
00112     esp_wifi_set_config(ESP_IF_WIFI_STA, &wifi_config);
00113     esp_wifi_start();
00114
00115     // Wait for connection or failure
00116     EventBits_t bits = xEventGroupWaitBits(wifi_event_group, WIFI_CONNECTED_BIT | WIFI_FAIL_BIT,
      pdFALSE, pdFALSE, portMAX_DELAY);
00117     if (bits & WIFI_CONNECTED_BIT) {
00118         ESP_LOGI(TAG, "Connected to Wi-Fi: %s", WIFI_SSID);
00119     } else {
00120         ESP_LOGE(TAG, "Failed to connect to Wi-Fi: %s", WIFI_SSID);
00121     }
00122 }
00123
00128 EventGroupHandle_t wifi_get_event_group(void)
00129 {
00130     return wifi_event_group;
00131 }
00132
00136 void wifi_stop(void) {
00137     // Deregister event handlers
00138     if (instance_any_id != NULL) {
00139         esp_event_handler_instance_unregister(WIFI_EVENT, ESP_EVENT_ANY_ID, instance_any_id);
00140         instance_any_id = NULL;
00141     }
00142     if (instance_got_ip != NULL) {
00143         esp_event_handler_instance_unregister(IP_EVENT, IP_EVENT_STA_GOT_IP, instance_got_ip);
00144         instance_got_ip = NULL;
00145     }
00146
00147     // Stop WiFi
00148     esp_wifi_disconnect();
00149     esp_wifi_stop();
00150     esp_wifi_deinit();
00151
00152     // Delete event loop
00153     esp_event_loop_delete_default();
00154
00155     // Delete event group
00156     if (wifi_event_group != NULL) {
00157         vEventGroupDelete(wifi_event_group);
00158         wifi_event_group = NULL;
00159     }
00160
00161     ESP_LOGI(TAG, "Disconnected from Wi-Fi: %s", WIFI_SSID);
00162 }
00163
00168 bool wifi_is_connected(void) {
00169     return wifi_event_group != NULL && (xEventGroupGetBits(wifi_event_group) & WIFI_CONNECTED_BIT);
00170 }
```

4.55 main/wifi.h File Reference

```
#include "freertos/FreeRTOS.h"
#include "freertos/event_groups.h"
#include "config.h"
```

Macros

- #define MAX_RETRY_COUNT 0

  Maximum number of reconnection attempts (0 means infinite retries).
- #define WIFI_TIMEOUT_MS 10000

  WiFi connection timeout in milliseconds (10 seconds).
- #define WIFI_CONNECTED_BIT BIT0

  WiFi credentials (defined in config.h).
- #define WIFI_FAIL_BIT BIT1

  Bit definition for WiFi failure status in event group.

Functions

- EventGroupHandle_t wifi_get_event_group (void)

  Get the handle to the WiFi event group.
- void wifi_init (void)

  Initialize the WiFi module and start the connection process.
- void wifi_stop (void)

  Stop the WiFi module and disconnect.
- bool wifi_is_connected (void)

  Check if the WiFi is currently connected.

4.55.1 Macro Definition Documentation

4.55.1.1 MAX_RETRY_COUNT

```
#define MAX_RETRY_COUNT 0
```

Maximum number of reconnection attempts (0 means infinite retries).

Definition at line 12 of file wifi.h.

4.55.1.2 WIFI_CONNECTED_BIT

```
#define WIFI_CONNECTED_BIT BIT0
```

WiFi credentials (defined in config.h).

Note

These are not defined here but expected to be set in config.h.

Bit definition for WiFi connected status in event group.

Definition at line 29 of file wifi.h.

4.55.1.3 WIFI_FAIL_BIT

```
#define WIFI_FAIL_BIT BIT1
```

Bit definition for WiFi failure status in event group.

Definition at line 34 of file wifi.h.

4.55.1.4 WIFI_TIMEOUT_MS

```
#define WIFI_TIMEOUT_MS 10000
```

WiFi connection timeout in milliseconds (10 seconds).

Definition at line 17 of file wifi.h.

4.55.2 Function Documentation

4.55.2.1 wifi_get_event_group()

```
EventGroupHandle_t wifi_get_event_group (
            void )
```

Get the handle to the WiFi event group.

Returns

EventGroupHandle_t Handle to the WiFi event group.

Definition at line 128 of file wifi.c.

4.55.2.2 wifi_init()

```
void wifi_init (
            void )
```

Initialize the WiFi module and start the connection process.

Definition at line 88 of file wifi.c.

4.55.2.3 wifi_is_connected()

```
bool wifi_is_connected (
            void )
```

Check if the WiFi is currently connected.

Returns

bool True if connected, false otherwise.

Definition at line 168 of file wifi.c.

4.55.2.4 wifi_stop()

```
void wifi_stop (
            void )
```

Stop the WiFi module and disconnect.

Definition at line 136 of file wifi.c.

4.56 wifi.h

Go to the documentation of this file.

```
00001 #ifndef WIFI_H
00002 #define WIFI_H
00003
00004 #include "freertos/FreeRTOS.h"
00005 #include "freertos/event_groups.h"
00006
00007 #include "config.h"
00008
00012 #define MAX_RETRY_COUNT 0
00013
00017 #define WIFI_TIMEOUT_MS 10000
00018
00023 // #define WIFI_SSID
00024 // #define WIFI_PASS
00025
00029 #define WIFI_CONNECTED_BIT BIT0
00030
00034 #define WIFI_FAIL_BIT      BIT1
00035
00040 EventGroupHandle_t wifi_get_event_group(void);
00041
00045 void wifi_init(void);
00046
00050 void wifi_stop(void);
00051
00056 bool wifi_is_connected(void);
00057
00058 #endif // WIFI_H
```

Index