QSSI Automatic Weather Station

Generated by Doxygen 1.8.14

Contents

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

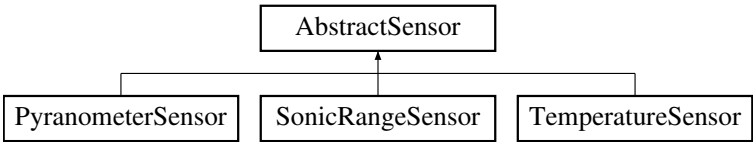
Here is a list of all documented files with brief descriptions:

Chapter 4

Class Documentation

4.1 AbstractSensor Class Reference

Inheritance diagram for AbstractSensor:



Public Member Functions

- virtual int8_t setup ()=0

  Set up a sensor.
- virtual uint8_t read (uint8_t *address)=0

  Read from a sensor.

4.1.1 Member Function Documentation

4.1.1.1 read()

```
virtual uint8_t AbstractSensor::read (
            uint8_t * address )  [pure virtual]
```

Read from a sensor.

Required method: read from the sensor Read from the initialized sensor and return its data.

Parameters

address	pointer to memory where output will be stored as bytes; this is to ensure consistency.

Returns

uint8_t length of byte string that needs to be read.

Implemented in TemperatureSensor, PyranometerSensor, and SonicRangeSensor.

4.1.1.2 setup()

```
virtual int8_t AbstractSensor::setup ( )  [pure virtual]
```

Set up a sensor.

Required method: set up the sensor. Do whatever this sensor needs to be ready for operation. This may include initializing other libraries, routing pins, etc.

Returns

Status code (-1 -> error, 0 -> success, other)

Implemented in TemperatureSensor, PyranometerSensor, and SonicRangeSensor.

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

- AbstractSensor.h

4.2 NVS_container::Data Struct Reference

Public Attributes

- uint8_t **time_buf** [64]
- uint8_t **temp_buf** [64]
- uint8_t **snow_buf** [64]
- uint8_t **pyro_buf** [64]

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

- NVS_container.h

4.3 NVS_container Class Reference

Classes

- struct Data

Public Member Functions

- uint8_t setup ()

  Setup for use.
- uint8_t close ()

  Close before system reset.
- uint8_t read_data (uint16_t index)

  Get a value from NVS.
- uint8_t write_data ()

  Write a value to NVS.
- uint8_t zero_data ()

  Zero the data `struct`.
- uint8_t clear ()

  Clear all entries from NVS.
- uint16_t get_counter ()

  Give the current value of the counter in memory.
- uint16_t get_fails ()

  Get the fail state for SD card writes.
- uint8_t set_fails (uint16_t fails)

  get the fail state for SD card writes.

Public Attributes

- struct NVS_container::Data data

  All fields of a single timepoint observation.

4.3.1 Member Function Documentation

4.3.1.1 clear()

```
uint8_t NVS_container::clear ( )
```

Clear all entries from NVS.

Returns

status (0 -> success, -1 -> failure)

This is the last step of writing our data to SD card: we must make sure that NVS is empty so we can write to it again.

4.3.1.2 close()

```
uint8_t NVS_container::close ( )
```

Close before system reset.

Returns

status (0 -> success, -1 -> failure)

Fix counter, disable NVS connection, etc.

4.3.1.3 get_counter()

```
uint16_t NVS_container::get_counter ( )
```

Give the current value of the counter in memory.

Returns

counter The current counter of the object.

This is needed to check whether it's time to move data from NVS -> SD.

4.3.1.4 get_fails()

```
uint16_t NVS_container::get_fails ( )
```

Get the fail state for SD card writes.

SD cards don't work very well with esp32. This is part of a hack to retry bad writes.

4.3.1.5 read_data()

```
uint8_t NVS_container::read_data (
            uint16_t index )
```

Get a value from NVS.

Parameters

index	Identifier for NVS key.

Returns

status (0 -> success, -1 -> failure)

This will pull out data from NVS and hand it to us in a usable format: as members of our data `struct`. Make sure you get it out before you call it again!

4.3.1.6 set_fails()

```
uint8_t NVS_container::set_fails (
            uint16_t fails )
```

get the fail state for SD card writes.

SD cards don't work very well with esp32. This is part of a hack to retry bad writes.

4.3.1.7 setup()

```
uint8_t NVS_container::setup ( )
```

Setup for use.

Returns

status (0 -> success, other -> failure)

If there is no counter in the NVS, make it now. Make namespace. Zero data. Do anything we need to ensure we're ready for reading/writing.

4.3.1.8 write_data()

```
uint8_t NVS_container::write_data ( )
```

Write a value to NVS.

Returns

status (0 -> success, -1 -> failure)

This will write everything in our data `struct` to NVS. This means of course that everything needs to be written into the struct before we call this method...

4.3.1.9 zero_data()

```
uint8_t NVS_container::zero_data ( )
```

Zero the data `struct`.

Returns

status (0 -> success, -1 -> failure)

This will simply zero the array for our next use.

4.3.2 Member Data Documentation

4.3.2.1 data

`NVS_container::data`

All fields of a single timepoint observation.

This wraps all of our sensors and timestamp into a single package. We use it for reading/writing purposes.

The documentation for this class was generated from the following files:

- NVS_container.h
- NVS_container.cpp

4.4 PowerState Class Reference

Public Member Functions

- uint8_t enter_basic_state ()
  Set the board's power state to mostly off.
- uint8_t enter_sensor_state ()
  Set the board's power state to read from sensors.
- uint8_t enter_SD_card_write_state (uint8_t number)
  Set the board's power state to write an SD card.
- uint8_t enter_wifi_station_state ()
  Set the board's power state to broadcast WiFi.
- uint8_t enter_sleep ()
  Perform any necessary configurations and enter deep sleep.

4.4.1 Member Function Documentation

4.4.1.1 enter_basic_state()

```
uint8_t PowerState::enter_basic_state ( )
```

Set the board's power state to mostly off.

Returns

status Indicates exit status of configuration.

This is mostly used to transition to the next state; by 'zeroing' everything at once, we don't need to do as much in other states.

4.4.1.2 enter_SD_card_write_state()

```
uint8_t PowerState::enter_SD_card_write_state (
            uint8_t number )
```

Set the board's power state to write an SD card.

Parameters

number	The number of the SD card to use (1 or 2 for now).

Returns

status Indicates exit status of configuration.

This should be turned on for an SD card before writing out from NVS. I'd like to make it more general (for n cards...) but that's harder.

4.4.1.3 enter_sensor_state()

```
uint8_t PowerState::enter_sensor_state ( )
```

Set the board's power state to read from sensors.

Returns

status Indicates exit status of configuration.

The state we spend most time in aside from asleep! This turns on the sensor power bus so that our sensors can read.

4.4.1.4 enter_sleep()

```
uint8_t PowerState::enter_sleep ( )
```

Perform any necessary configurations and enter deep sleep.

Returns

status Indicates exit status (any status return is bad here).

This method may undergo changes depending on microcontroller architecture and board functionality; for example, ESP32 Thing requires both slow and fast memory to be disabled to achieve optimal performance.

Currently, we allow 3 things to wake the board: the RTC alarm on pin 39, button press on pin 32, and button press on pin 25. These are each handled differently on wakeup (because each wakeup implies different desired behavior).

4.4.1.5 enter_wifi_station_state()

```
uint8_t PowerState::enter_wifi_station_state ( )
```

Set the board's power state to broadcast WiFi.

Returns

status Indicates exit status of configuration.

We need to broadcast information so we will make a local wifi access point and broadcase a simple html page with some data.

The documentation for this class was generated from the following files:

- PowerState.h
- PowerState.cpp

4.5 pyranometer Class Reference

a simple interface to an external ADC sensor.

`#include <PyranometerSensor.h>`

4.5.1 Detailed Description

a simple interface to an external ADC sensor.

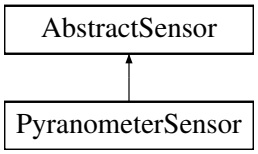
Our pyranometer requires an external adc to work with the esp32 but it should behave just like any other sensor. This ensures it acts according to established design criteria.

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

- PyranometerSensor.h

4.6 PyranometerSensor Class Reference

Inheritance diagram for PyranometerSensor:



Public Member Functions

- int8_t setup ()
  Set up the pyranometer.
- uint8_t read (uint8_t *address)
  Read from the pyranometer.
- int8_t end ()
  Delete adc variable created in setup.

4.6.1 Member Function Documentation

4.6.1.1 end()

`int8_t PyranometerSensor::end ( )`

Delete adc variable created in setup.

Returns

status The function exit state.

4.6.1.2 read()

```
uint8_t PyranometerSensor::read (
            uint8_t * address )  [virtual]
```

Read from the pyranometer.

Parameters

address	The memory location where ourput will be written.

Returns

len The length written into memory at `address`.

Implements AbstractSensor.

4.6.1.3 setup()

```
int8_t PyranometerSensor::setup ( )  [virtual]
```

Set up the pyranometer.

Returns

status The function exit state.

Implements AbstractSensor.

The documentation for this class was generated from the following files:

- PyranometerSensor.h
- PyranometerSensor.cpp

4.7 RTC_container Class Reference

Wraps the RTC_DS3231 object for ease of use.

```
#include <RTC_container.h>
```

Public Member Functions

- int8_t check_datetime ()
  Update RTC date/time if needed.
- void print_date (DateTime t)
  Print out the contents of a DateTime (human readable).
- int8_t setup ()
  Prepare the RTC for operation.
- int8_t set_next_alarm (int32_t seconds, int8_t alarm_num)
  Set the next alarm to wake up the chip.

Public Attributes

- RTC_DS3231 rtc

  A real-time clock object.

4.7.1 Detailed Description

Wraps the RTC_DS3231 object for ease of use.

As written, the RTC_DS3231 lacks some features we would like for our project, such as a dedicated timestamp method and a simple repeating alarm system. In addition, we want to simplify the setup procedure into a smaller call.

4.7.2 Member Function Documentation

4.7.2.1 check_datetime()

```
int8_t RTC_container::check_datetime ( )
```

Update RTC date/time if needed.

Returns

status An int (0 -> no change, 1 -> change, other -> error)

This check will only take place when the system isn't woken up from deep sleep (e.g. it is first powered on). TODO are there other cases we need to check?

4.7.2.2 print_date()

```
void RTC_container::print_date (
            DateTime t )
```

Print out the contents of a DateTime (human readable).

Parameters

t	A DateTime object.

4.7.2.3 set_next_alarm()

```
int8_t RTC_container::set_next_alarm (
```

```
int32_t seconds,
int8_t alarm_num )
```

Set the next alarm to wake up the chip.

Parameters

seconds	A count of seconds from current time at which the next alarm should be set.

Returns

status An int (0 -> success, -1 -> error)

This is here to get us around the limitations of the RTC which can only take one second intervals or some future time to set an alarm.

4.7.2.4 setup()

```
int8_t RTC_container::setup ( )
```

Prepare the RTC for operation.

Returns

status An int (0 -> success, -1 -> error)

4.7.3 Member Data Documentation

4.7.3.1 rtc

```
RTC_DS3231 RTC_container::rtc
```

A real-time clock object.

An instance of a RTC_DS3231, capable of sending both square waves and interrupts to a pin on the board.

The documentation for this class was generated from the following files:

- RTC_container.h
- RTC_container.cpp

4.8 SD_container Class Reference

Public Member Functions

- int8_t setup (const char *path, int path_len, int card_pin)

  Set up this card for writing.
- int8_t close ()

  Disengage the filesystem and close the SD card connection.
- int8_t print_file ()

  Read the file and print it out.
- int8_t make_file ()

  Write the file if it doesn't exist.
- int8_t append_to_file (String line)

  Append to the file this container tracks.
- int8_t make_line (NVS_container *nvs, int index)

  Grab data from NVS and format it into an entry for our file.

Related Functions

(Note that these are not member functions.)

- class SD_container

  SD cards should be able to manipulate this class.

4.8.1 Member Function Documentation

4.8.1.1 append_to_file()

```
int8_t SD_container::append_to_file (
            String line )
```

Append to the file this container tracks.

Returns

status (0 -> success, -1 -> failure)

This should be called for each line we want to write to the file system. It should be called many times every time we empty NVS.

4.8.1.2 close()

```
int8_t SD_container::close ( )
```

Disengage the filesystem and close the SD card connection.

Returns

status (0 -> success, -1 -> failure)

This should obviously be called when we're done using the SD card.

4.8.1.3 make_file()

```
int8_t SD_container::make_file ( )
```

Write the file if it doesn't exist.

Returns

status (0 -> success, -1 -> failure)

This could be called exactly one time: on system start. Every subsequent operation should instead append to the file.

4.8.1.4 make_line()

```
int8_t SD_container::make_line (
            NVS_container * nvs,
            int index )
```

Grab data from NVS and format it into an entry for our file.

Returns

status (0 -> success, -1 -> failure)

This method directly manipulates the NVS to populate its `data` struct with whatever is stored at `index`, format a new line for the file, and then append it to the file. It should be iteratively called on the NVS_container from 0 to MAX_NVS_COUNTER.

4.8.1.5 print_file()

```
int8_t SD_container::print_file ( )
```

Read the file and print it out.

Returns

status (0 -> success, -1 -> failure)

This is entirely for diagnostic purposes; in operation, we won't be reading at all.

4.8.1.6 setup()

```
int8_t SD_container::setup (
            const char * path,
            int path_len,
            int card_pin )
```

Set up this card for writing.

Parameters

path	The file path to use.
card_pin	The slave select for this SD card.

Returns

status (0 -> success, -1 -> failure)

This method will wrap a lot of complex functionality. It needs to be sure the SD card is alive and talking, the filesystem is valid, the path is good, and more. Then it needs to inform the rest of the program that it's ready.

4.8.2 Friends And Related Function Documentation

4.8.2.1 SD_container

```
friend class SD_container  [friend]
```

SD cards should be able to manipulate this class.

SD cards will be able to load this class's `data` field as they need by requesting particular indices from storage; it will then format the data and write it to the card.

The documentation for this class was generated from the following files:

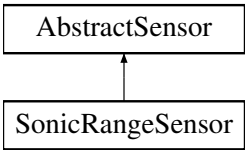
- SD_container.h
- NVS_container.h
- SD_container.cpp

4.9 SonicRangeSensor Class Reference

a simple interface to a pulse-measuring sensor.

```
#include <SonicRangeSensor.h>
```

Inheritance diagram for SonicRangeSensor:



Public Member Functions

- int8_t setup ()

  Set up the sonic ranger.
- uint8_t read (uint8_t *address)

  Read from the sonic ranger.

4.9.1 Detailed Description

a simple interface to a pulse-measuring sensor.

Our new sonic range sensor is fairly simple to use but it should behave just like any other sensor. This ensures it acts according to established design criteria.

4.9.2 Member Function Documentation

4.9.2.1 read()

```
uint8_t SonicRangeSensor::read (
            uint8_t * address )  [virtual]
```

Read from the sonic ranger.

Read from the SRS, format the data, and give it back to main. Formatting is an issue for utils.h, so this method is actually pretty simple.

Parameters

address	The memory location where ourput will be written.

Returns

len The length written into memory at `address`.

Implements AbstractSensor.

4.9.2.2 setup()

```
int8_t SonicRangeSensor::setup ( )  [virtual]
```

Set up the sonic ranger.

Returns

status The function exit state.

Implements AbstractSensor.

The documentation for this class was generated from the following files:

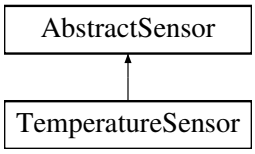
- SonicRangeSensor.h
- SonicRangeSensor.cpp

4.10 TemperatureSensor Class Reference

Wraps the DallasTemperature lib to simplify readings.

```
#include <TemperatureSensor.h>
```

Inheritance diagram for TemperatureSensor:



Public Member Functions

- int8_t setup ()
- uint8_t read (uint8_t *address)

  Read from the sensor and return it in appropriate format.

4.10.1 Detailed Description

Wraps the DallasTemperature lib to simplify readings.

The DallasTemperature library gives a lot of powerful tools but it's still kind of tricky to use. This simplifies the function calls for our central application logic.

4.10.2 Member Function Documentation

4.10.2.1 read()

```
uint8_t TemperatureSensor::read (
            uint8_t * address )  [virtual]
```

Read from the sensor and return it in appropriate format.

This sensor has some odd behavior in the prototype where it returns errors for a while after it is first powered on by the transistor-regulated ground. As such, we poll the device for temperatures until it gives us a number that isn't one of the error codes defined above (85 or -127) or it runs out of tries (MAX_TS_RETRIES).

Implements AbstractSensor.

4.10.2.2 setup()

```
int8_t TemperatureSensor::setup ( )  [virtual]
```

/brief Set up the sensor using its own values.

NOTE: if we end up using more sensors we may want to pass in a OneWire, DallasTemperature, etc object so that other temperature sensors can all use it...

Implements AbstractSensor.

The documentation for this class was generated from the following files:

- TemperatureSensor.h
- TemperatureSensor.cpp

Chapter 5

File Documentation

5.1 AbstractSensor.h File Reference

Base class for all sensors.

```
#include <stdint.h>
```

Classes

- class AbstractSensor

5.1.1 Detailed Description

Base class for all sensors.

All sensors will adhere to a simple guideline:

- they must be able to set up
- they must be able to read

Index