Oregon Scientific Example Documentation

Generated by Doxygen 1.8.4

Wed Jul 23 2014 11:38:34

Contents

Chapter 1

Todo List

File OregonScientificExample.ino

Add the ability to dynamicall add and remove sensors.

Class OregonScientificSensor

Add more message formats and titles to the class.

Member OregonScientificSensor::makeJSONMessage (uint8_t *message)

Shore up what the exact format will be for communicating with the server.

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 files with brief descriptions:

Chapter 4

Class Documentation

4.1 id_type Union Reference

A union for ease of converting between the array and integer representations of the device ID.

```
#include "OregonScientificSensor.h"
```

Public Attributes

- uint8_t array [4]
- uint32_t value

4.1.1 Detailed Description

A union for ease of converting between the array and integer representations of the device ID.

Definition at line 46 of file OregonScientificSensor.h.

4.1.2 Member Data Documentation

4.1.2.1 uint8_t id_type::array[4]

Definition at line 47 of file OregonScientificSensor.h.

4.1.2.2 uint32_t id_type::value

Definition at line 48 of file OregonScientificSensor.h.

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

- OregonScientificSensor/OregonScientificSensor.h

4.2 ManchesterDecoder Class Reference

```
#include "ManchesterDecoder.h"
```

Public Member Functions

- ManchesterDecoder ()

  The Default Constructor.
- ∼ManchesterDecoder ()

  The Destructor.
- uint8_t getNextPulse ()

  Gets the next result from the decoder (ZERO, ONE, RESET).
- boolean hasNextPulse ()

  Checks if the data buffer is empty.
- void reset ()

  Resets the decoder by clearing the input and output buffers and re-initializing the state machine.

Private Member Functions

- virtual void interruptResponder ()

  The private virtual iterrupt handler which is called by the isr.
- void decode (word width)

  Decodes the pulse width and updates the state machine, which could in turn add data to the data buffer.
- void toggle (unsigned int ∗state)

  A helper function used to toggle the state of the state machine.

Static Private Member Functions

- static void isr2 ()

  The private static interrupt service routine.

Private Attributes

- unsigned int state

  The state variable used by the state machine.
- uint8_t halfClock

  A variable used by the state machine to determine what state to go to next.
- boolean start

  A boolean variable which is used to ensure that special considerations are met when decoding the manchester encoded data fro the Oregon Scientific Sensors.
- volatile word pulse

  The volatile variable pulse is used to record the time between transition on the data line.
- WordBuffer ∗ pulse_buffer

  The input buffer in which the pulse values are stored.
- WordBuffer ∗ data_buffer

  The data buffer in which the decoded data is placed.

Static Private Attributes

- static ManchesterDecoder ∗ selfPointer

  The static self pointer which is necessary in order for the interrupt handler to be able to add data to the input buffer.

4.2.1 Detailed Description

Definition at line 43 of file ManchesterDecoder.h.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 ManchesterDecoder::ManchesterDecoder ( )

The Default Constructor.

Definition at line 9 of file ManchesterDecoder.cpp.

```
9                                      {
10      // Configures the interrupt pin as  INPUT
11      pinMode(3, INPUT);
12      // Configures the interrupt pin with internal pullup resistor
13      digitalWrite(3, 1);
14      // Allocates memory for the buffers
15      data_buffer = new WordBuffer(DEFAULT_SIZE);
16      pulse_buffer = new WordBuffer(DEFAULT_SIZE);
17      // Initializes the member variables
18      halfClock = 1;
19      start = true;
20      state = ZERO;
21      selfPointer = this;
22      // Attaches the interrupt to the IRS on pin change
23      attachInterrupt(1, ManchesterDecoder::isr2, CHANGE);
24      // Enable interrupts
25      interrupts();
26 }
```

4.2.2.2 ManchesterDecoder::∼ManchesterDecoder ( )

The Destructor.

Definition at line 31 of file ManchesterDecoder.cpp.

```
31                                       {
32      // Frees the memory occupied by the buffers
33      delete data_buffer;
34      delete pulse_buffer;
35 }
```

4.2.3 Member Function Documentation

4.2.3.1 void ManchesterDecoder::decode (word *width*) [private]

Decodes the pulse width and updates the state machine, which could in turn add data to the data buffer.

Definition at line 91 of file ManchesterDecoder.cpp.

```
91                                          {
92      if (50 <= width && width < 1400) {
93          boolean w = width >= 750;
94          switch(w){
95               // Short pulses increase count by 1
96              // There is a boundary condition between the two protocols
97              case SHORT_PULSE:
98                  if(!start){
99                      halfClock++;
100                      break;
101                  }
102               // Long pulses increase count by 2
103             case LONG_PULSE:
104                 start = false;
105                 if(state == ZERO){
106                     state = ONE;
107                 }else{
108                     state = ZERO;
109                 }
110                 halfClock+=2;
111                 break;
112         }
113         halfClock %= 2;
114         if(halfClock == 0){
115             data_buffer->insert(state);
116         }
117     }
118     else{
119         data_buffer->insert(RESET);
120         reset();
121     }
122 }
```

4.2.3.2 uint8_t ManchesterDecoder::getNextPulse ( )

Gets the next result from the decoder (ZERO, ONE, RESET).

Returns

The next result from the data buffer.

Definition at line 79 of file ManchesterDecoder.cpp.

```
79                                         {
80     return data_buffer->remove();
81 }
```

4.2.3.3 boolean ManchesterDecoder::hasNextPulse ( )

Checks if the data buffer is empty.

Returns

True if the buffer is not empty; false otherwise.

Definition at line 70 of file ManchesterDecoder.cpp.

```
70                                         {
71     if(!pulse_buffer->isEmpty()){
72         while(!pulse_buffer->isEmpty()){
73             decode(pulse_buffer->remove());
74         }
75     }
76     return !data_buffer->isEmpty();
77 }
```

4.2.3.4 void ManchesterDecoder::interruptResponder ( ) [private], [virtual]

The private virtual iterrupt handler which is called by the isr.

Definition at line 50 of file ManchesterDecoder.cpp.

```
50                                        {
51      // Static variable records the last time when the function was called
52      static word last;
53      // Computes the time since the last function call
54      pulse = micros() - last;
55      last += pulse;
56      // Assumes that the buffer will always be empty
57      // Insert the pulse into the buffer
58      pulse_buffer->insert(pulse);
59 }
```

4.2.3.5 void ManchesterDecoder::isr2 ( ) [static], [private]

The private static interrupt service routine.

Responds to the interrupts by calling the interrupt handler.

Definition at line 40 of file ManchesterDecoder.cpp.

```
40                           {
41      //cli();
42      // Calls the interrupt handler
43      ManchesterDecoder::selfPointer->
     interruptResponder();
44      //sei();
45 }
```

4.2.3.6 void ManchesterDecoder::reset ( )

Resets the decoder by clearing the input and output buffers and re-initializing the state machine.

Definition at line 64 of file ManchesterDecoder.cpp.

```
64                            {
65      halfClock = 0;
66      state = ZERO;
67      start = true;
68 }
```

4.2.3.7 void ManchesterDecoder::toggle (unsigned int * *state*) [private]

A helper function used to toggle the state of the state machine.

Definition at line 83 of file ManchesterDecoder.cpp.

```
83                                                {
84      if(state == ZERO){
85          *state = ONE;
86      }else{
87          *state = ZERO;
88      }
89 }
```

4.2.4 Member Data Documentation

4.2.4.1 WordBuffer* ManchesterDecoder::data_buffer [private]

The data buffer in which the decoded data is placed.

Definition at line 95 of file ManchesterDecoder.h.

4.2.4.2 uint8_t ManchesterDecoder::halfClock [private]

A variable used by the state machine to determine what state to go to next.

It also determines whether the given pulse was valid or if it produced output.

Definition at line 75 of file ManchesterDecoder.h.

4.2.4.3 volatile word ManchesterDecoder::pulse [private]

The volatile variable pulse is used to record the time between transition on the data line.

It must volatile, because it appears to the compiler that the value should never change as it is never called. However since it is inside an isr it does change thereby requiring the volatile keyword.

Definition at line 91 of file ManchesterDecoder.h.

4.2.4.4 WordBuffer∗ ManchesterDecoder::pulse_buffer [private]

The input buffer in which the pulse values are stored.

Definition at line 93 of file ManchesterDecoder.h.

4.2.4.5 ManchesterDecoder ∗ ManchesterDecoder::selfPointer [static], [private]

The static self pointer which is necessary in order for the interrupt handler to be able to add data to the input buffer.

Definition at line 69 of file ManchesterDecoder.h.

4.2.4.6 boolean ManchesterDecoder::start [private]

A boolean variable which is used to ensure that special considerations are met when decoding the manchester encoded data fro the Oregon Scientific Sensors.

This is because the version 3.0 and 2.1 protocols differ in the way in which they start their messages. In version 3.0 messages you do not consider the first transition to be decoded as a logical 1, whereas in the version 2.1 protocol you do in order to prodce the correct output.

Definition at line 84 of file ManchesterDecoder.h.

4.2.4.7 unsigned int ManchesterDecoder::state [private]

The state variable used by the state machine.

Definition at line 71 of file ManchesterDecoder.h.

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

- ManchesterDecoder/ManchesterDecoder.h
- ManchesterDecoder/ManchesterDecoder.cpp

4.3 OregonScientific Class Reference

OregonScientific defines a parser capable of parsing both version 2.1 and version 3.0 messages from Oregon Scientific sensors.

```
#include "OregonScientific.h"
```

Public Member Functions

- OregonScientific ()

  The default constructor.
- OregonScientific (uint8_t msgLen)

  The constructor which takes the length of the message.
- ∼OregonScientific ()

  The destructor.
- boolean parseOregonScientificV3 (uint8_t width)

  The member function that parses the version 3.0 protocol.
- boolean parseOregonScientificV2 (uint8_t width)

  The member function that parses the version 2.1 protocol.
- void addSensor (OregonScientificSensor ∗sensor)

  The member function that "listens" for a message that was sent by its sensor.
- virtual void printResults (uint8_t protocol)

  Prints the results of the two sensors that this code has been tested with.
- virtual void reset ()

  Allows the parser to be reset manually.
- OregonScientificSensor ∗ getCurrentSensor ()

  Returns the sensor that sent the message.

Private Member Functions

- boolean validate (uint8_t value)

  Validates the message by computing the checksum and checking to see if it matches the checksum that was sent by the sensor.
- boolean findSensor ()

  Finds the sensor that matches the device id and channel number of the message that is currently being received.

Private Attributes

- uint8_t subNibbleCount

  The counter that is used to track the number of bits added to each nibble.
- int idx

  The current index into the message array.
- uint8_t bitCount

  Used by the version 2.1 protocol parser to determine which bits to throw away.
- uint8_t messageSize

  The variable that stores the message size when find sensor is called.
- uint8_t numSensors

Holds the number of sensors that are currently attached to the parser.

- OregonScientificSensor * currentSensor

  The sensor that sent the current message.
- OregonScientificSensor * sensors [MAX_SENSOR_NUM]

  The array of sensors that are currently being listened for.
- OregonScientific_ParseState state

  The variable that holds the current state of the parser.
- uint8_t * data

  The array that holds the message.

4.3.1 Detailed Description

OregonScientific defines a parser capable of parsing both version 2.1 and version 3.0 messages from Oregon Scientific sensors.

Author

Joel D. Sabol

Date

June 2014

The parser will take the output of the Manchester decoder and parse that data until it finds the sync nibble. It will then parse the device id and the channel. These two data members will be used to lookup the sensor that sent the message. If the sensor is found it will be placed in the current sensor variable

Definition at line 49 of file OregonScientific.h.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 OregonScientific::OregonScientific ( )

The default constructor.

Definition at line 3 of file OregonScientific.cpp.

```
3                                  {
4     data = new uint8_t[DEFAULT_SIZE];
5     numSensors = 0;
6     messageSize = DEFAULT_SIZE;
7     reset();
8 }
```

4.3.2.2 OregonScientific::OregonScientific (uint8_t *msgLen* = DEFAULT_SIZE)

The constructor which takes the length of the message.

Parameters

msgLen	The expected length of the message.

Definition at line 10 of file OregonScientific.cpp.

```
10                                                                {
11      data = new uint8_t[messageSize];
12      OregonScientific::messageSize = messageSize;
13      reset();
14 }
```

4.3.2.3 OregonScientific::∼OregonScientific ( )

The destructor.

Definition at line 16 of file OregonScientific.cpp.

```
16                                     {
17      delete[] data;
18 }
```

4.3.3 Member Function Documentation

4.3.3.1 void OregonScientific::addSensor (OregonScientificSensor ∗ *sensor*)

The member function that "listens" for a message that was sent by its sensor.

So whenever the parser parses a device id and channel id it will search for the sensor that matches the device id - channel id combination.

Parameters

∗*sensor*	The sensor that will be listened for by the parser.

Definition at line 184 of file OregonScientific.cpp.

```
184                                                         {
185      sensors[numSensors] = sen1;
186      numSensors++;
187 }
```

4.3.3.2 boolean OregonScientific::findSensor ( ) [private]

Finds the sensor that matches the device id and channel number of the message that is currently being received.

If it is found it will place the sensor in the current sensor variable. In addition to this it will also get the size of the message that is being received so that the parser will know when to stop. This message size is also used to determine where the checksum is located.

Returns

True if the sensor was found, false otherwise.

Definition at line 189 of file OregonScientific.cpp.

```
189                                     {
190     id_type temp;
191     // Reverse copy the data into the union to convert
192     // the dev_id to an integer representation.
193     for(uint8_t i = 0; i < 4; i++){
194         temp.array[i] = data[DEV_ID_END-i];
195     }
196     // Check all the sensors to find the sensor that matches the dev ID and channel
197     for (uint8_t i = 0; i < numSensors; i++){
198         if(sensors[i]->getSensorID() == temp.value &&
199             sensors[i]->getSensorChannel() == data[
      CHANNEL_NIBBLE]){
200             currentSensor = sensors[i];
201             messageSize = currentSensor->getMessageSize();
202             return true;
203         }
204     }
205     return false;
206 }
```

4.3.3.3 OregonScientificSensor ∗ OregonScientific::getCurrentSensor ( )

Returns the sensor that sent the message.

Definition at line 31 of file OregonScientific.cpp.

```
31                                                          {
32     return currentSensor;
33 }
```

4.3.3.4 boolean OregonScientific::parseOregonScientificV2 (uint8_t *width*)

The member function that parses the version 2.1 protocol.

It parses the message as it receives the data from an outside source.

Parameters

width	The value to be shifted into the current nibble.

Definition at line 90 of file OregonScientific.cpp.

```
90                                                              {
91     if(idx > messageSize){
92         //reset();
93         return false;
94     }
95     bitCount++;
96     if(bitCount & 0x01){
97         data[idx] = data[idx] >> 1;
98         data[idx] = data[idx] | value;
99 
100         switch(state){
101             case SYNCING:
102                 if(data[idx] == 0x0A){
103                     state = GET_ID;
104                     subNibbleCount = 0;
105                 }
106                 break;
107             case GET_ID:
108                 if ((subNibbleCount & 0x03) == 3){
109                     idx++;
110                 }
111                 subNibbleCount++;
112                 if (idx > CHANNEL_NIBBLE){
113                     if(findSensor()){
114                         state = GET_MSG;
115                         //messageSize = currentSensor->getMessageSize();
116                     }else{
```

```
117                         state = SYNCING;
118                     }
119 
120                 }
121                 break;
122             case GET_MSG:
123                 if ((subNibbleCount & 0x03) == 3){
124                     idx++;
125                 }
126                 subNibbleCount++;
127                 if(idx >= messageSize){
128                     state = DONE;
129                 }
130                 break;
131             case DONE:
132                 currentSensor->makeJSONMessage(data);
133                 return validate(messageSize-3);
134         }
135     }
136     return false;
137 }
```

4.3.3.5 boolean OregonScientific::parseOregonScientificV3 (uint8_t *width*)

The member function that parses the version 3.0 protocol.

It parses the message as it receives the data from an outside source.

Parameters

width	The value to be shifted into the current nibble.

Definition at line 139 of file OregonScientific.cpp.

```
139                                                               {
140     if(idx > messageSize){
141         return false;
142     }
143     data[idx] = data[idx] >> 1;
144     data[idx] = data[idx] | value;
145 
146     switch(state){
147             case SYNCING:
148                 if(data[idx] == 0x0A){
149                     state = GET_ID;
150                     //idx++;
151                     subNibbleCount = 0;
152                 }
153                 break;
154             case GET_ID:
155                 if ((subNibbleCount & 0x03) == 3){
156                     idx++;
157                 }
158                 subNibbleCount++;
159                 if (idx > CHANNEL_NIBBLE){
160                     if(findSensor()){
161                         state = GET_MSG;
162                         //messageSize = currentSensor->getMessageSize();
163                     }else{
164                         state = SYNCING;
165                     }
166                 }
167                 break;
168             case GET_MSG:
169                 if ((subNibbleCount & 0x03) == 3){
170                     idx++;
171                 }
172                 subNibbleCount++;
173                 if(idx >= messageSize){
174                     state = DONE;
175                 }
176                 break;
177             case DONE:
178                 currentSensor->makeJSONMessage(data);
179                 return validate(messageSize-3);
180         }
181     return false;
182 }
```

4.3.3.6 void OregonScientific::printResults (uint8_t *protocol*) [virtual]

Prints the results of the two sensors that this code has been tested with.

Definition at line 35 of file OregonScientific.cpp.

```
35                                                     {
36   Serial.println();
37   switch(protocol){
38   case OSCV_3:
39     Serial.print(F("OSCV_3:\t"));
40     break;
41   case OSCV_2_1:
42     Serial.print(F("OSCV_2_1:\t"));
43     break;
44   }
45   for(int i = DEV_ID_BEGIN; i < idx; i++){
46     Serial.print(data[i], HEX);
47   }
48   Serial.println();
49   Serial.print(F("Dev ID:\t\t"));
50   for(int i = DEV_ID_BEGIN; i <= DEV_ID_END; i++){
51     Serial.print(data[i], HEX);
52   }
53   Serial.print(F("\nBattery:\t"));
54   if(data[FLAGS] >> 2){
55     Serial.println(F("Low"));
56   }else{
57     Serial.println(F("Ok"));
58   }
59   Serial.print(F("Channel:\t"));
60   if(protocol == OSCV_3){
61     Serial.println(data[CHANNEL_NIBBLE]);
62   }else{
63     switch(data[CHANNEL_NIBBLE]){
64         case 0x01: Serial.println(F("1"));
65         break;
66         case 0x02: Serial.println(F("2"));
67         break;
68         case 0x04: Serial.println(F("3"));
69         break;
70         default: Serial.println(F("Channel Error"));
71     }
72   }
73   Serial.print(F("Temp:\t\t"));
74   if(!(data[13] & 0x08) >> 3){
75     Serial.print(F("-"));
76   }
77   Serial.print(data[10], HEX);
78   Serial.print(data[9], HEX);
79   Serial.print(F("."));
80   Serial.print(data[8], HEX);
81   Serial.println(F("C"));
82   if(protocol == OSCV_2_1){
83     Serial.print(F("Humidity:\t"));
84     Serial.print(data[13], HEX);
85     Serial.print(data[12], HEX);
86     Serial.println(F("%\n"));
87   }
88 }
```

4.3.3.7 void OregonScientific::reset ( ) [virtual]

Allows the parser to be reset manually.

Though it is used internally by the class as well.

Definition at line 20 of file OregonScientific.cpp.

```
20                              {
21     subNibbleCount = 0;
22     idx = 0;
23     state = SYNCING;
24     bitCount = 0;
25     for (int i = 0; i < messageSize; ++i)
26     {
27         data[i] = 0;
28     }
29 }
```

4.3.3.8 boolean OregonScientific::validate (uint8_t *value*) [private]

Validates the message by computing the checksum and checking to see if it matches the checksum that was sent by the sensor.

Returns

True if the checksums matched, false otherwise.

Definition at line 208 of file OregonScientific.cpp.

```
208                                              {
209     // Converts the checksum to a single integer value for comparison
210     uint8_t chksum_dev = (data[value+1] << 4) | data[value];
211     uint8_t chksum_computed = 0;
212     // Computes the checksum of the message
213     for(uint8_t i = DEV_ID_BEGIN; i < value; i++){
214         chksum_computed += data[i];
215     }
216 
217     return (chksum_computed == chksum_dev);
218 }
```

4.3.4 Member Data Documentation

4.3.4.1 uint8_t OregonScientific::bitCount [private]

Used by the version 2.1 protocol parser to determine which bits to throw away.

Definition at line 102 of file OregonScientific.h.

4.3.4.2 OregonScientificSensor* OregonScientific::currentSensor [private]

The sensor that sent the current message.

Definition at line 108 of file OregonScientific.h.

4.3.4.3 uint8_t* OregonScientific::data [private]

The array that holds the message.

Definition at line 114 of file OregonScientific.h.

4.3.4.4 int OregonScientific::idx [private]

The current index into the message array.

Definition at line 99 of file OregonScientific.h.

4.3.4.5 uint8_t OregonScientific::messageSize [private]

The variable that stores the message size when find sensor is called.

Definition at line 104 of file OregonScientific.h.

4.3.4.6 uint8_t OregonScientific::numSensors [private]

Holds the number of sensors that are currently attached to the parser.

Definition at line 106 of file OregonScientific.h.

4.3.4.7 OregonScientificSensor∗ OregonScientific::sensors[MAX_SENSOR_NUM] [private]

The array of sensors that are currently being listened for.

Definition at line 110 of file OregonScientific.h.

4.3.4.8 OregonScientific_ParseState OregonScientific::state [private]

The variable that holds the current state of the parser.

Definition at line 112 of file OregonScientific.h.

4.3.4.9 uint8_t OregonScientific::subNibbleCount [private]

The counter that is used to track the number of bits added to each nibble.

Definition at line 97 of file OregonScientific.h.

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

- OregonScientific/OregonScientific.h
- OregonScientific/OregonScientific.cpp

4.4 OregonScientificSensor Class Reference

A class that encompasses the necessary information about Oregon Scientific sensors and the methods for accessing that information.

```
#include "OregonScientificSensor.h"
```

Public Member Functions

- OregonScientificSensor (const uint32_t id, const uint8_t dev_channel, const uint8_t size, const uint8_t ∗msg_format, const String ∗msg_spec)

The default constructor which requires information about the sensor.

- ∼OregonScientificSensor ()

  The destructor.

- uint32_t getSensorID ()

  Gets the sensor ID.

- uint8_t getSensorChannel ()

  Gets the channel that the sensor is on.

- void makeJSONMessage (uint8_t ∗message)

  Creates a JSON message given the data in the standard Oregon Scientific format.

- String getJSONMessage ()

  Gets the String representation of the JSON formated message.

- void getCharMessage (char &msg)

  Gets the char array representation of the JSON formated message.

- uint8_t getMessageSize ()

  Gets the expected size of the message.

Static Public Attributes

- static const uint8_t THGR122NX_FORMAT [] = { 0,18, 0,3, 4,4, 7,7, 8,11, 12,13, 15,16}

  The message format of the THGR122NX.

- static const uint8_t THWR800_FORMAT [] ={ 0,15, 0,3, 4,4, 7,7, 8,11, 12,13}

  The message format of the THWR800.

- static const String THGR122NX_TITLES [] = {"THGR122NX", "DevID", "Channel", "Battery", "Temp", "Humidity", "Checksum"}

  The titles corresponding to the THGR122NX_FORMAT message format.

- static const String THWR800_TITLES [] = {"THWR800","DevID", "Channel", "Battery", "Temp", "Checksum"}

  The titles corresponding to the THWR800_FORMAT message format.

Private Attributes

- uint32_t dev_id

  The member variable holding the sensors device ID.

- uint8_t ∗ format
- uint8_t channel
- uint8_t msg_size

  The member variable holding the expected size of the message.

- String ∗ titles

  The member array holding the titles corresponding to the format.

- String json_msg

  The member variable that holds the JSON message after it is created.

4.4.1 Detailed Description

A class that encompasses the necessary information about Oregon Scientific sensors and the methods for accessing that information.

Author

Joel D. Sabol

Todo Add more message formats and titles to the class.

Contains information pertaining to the sensors such as message format, message length, channel id, and device ID. Additionally it contains helper methods to turn the message that is received into a JSON message that can be sent to a server or application.

Definition at line 62 of file OregonScientificSensor.h.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 OregonScientificSensor::OregonScientificSensor (const uint32_t *id,* const uint8_t *dev_channel,* const uint8_t *size,* const uint8_t ∗ *msg_format,* const String ∗ *msg_spec*)

The default constructor which requires information about the sensor.

Parameters

id	The device id - Best to use the defined device IDs, however you can create one for a sensor.
dev_channel	The channel on which the device is "broadcasting".
size	The size of just the message, not including the basic sensor information.
∗*msg_format*	The array of number pairs comprised of the beginning index and end index of every data member of the message (See examples in .cpp file).
∗*msg_spec*	The title corresponding to each number pair.

See Also

THGR122NX Example of id parameter.
V2_CHANNEL_1 Example of dev_channel parameter.
THGR122NX_FORMAT[] Example of msg_format parameter.
THGR122NX_TITLES[] Example of msg_spec parameter.

Definition at line 9 of file OregonScientificSensor.cpp.

```
10                                                                                              {
11      dev_id = (uint32_t) id;
12      format = (uint8_t*) msg_format;
13      titles = (String*) msg_titles;
14      channel = (uint8_t) dev_channel;
15      msg_size = (uint8_t) size;
16 }
```

4.4.2.2 OregonScientificSensor::∼OregonScientificSensor ( )

The destructor.

Definition at line 17 of file OregonScientificSensor.cpp.

```
17 {}
```

4.4.3 Member Function Documentation

4.4.3.1 void OregonScientificSensor::getCharMessage (char & *msg*)

Gets the char array representation of the JSON formated message.

Parameters

The	buffer to place the JSON formatted message into.

Definition at line 34 of file OregonScientificSensor.cpp.

```
34                                                  {
35     json_msg.toCharArray(&msg, json_msg.length());
36 }
```

4.4.3.2 String OregonScientificSensor::getJSONMessage ( )

Gets the String representation of the JSON formated message.

Returns

The string object containing the JSON message.

Definition at line 30 of file OregonScientificSensor.cpp.

```
30                                           {
31     return json_msg;
32 }
```

4.4.3.3 uint8_t OregonScientificSensor::getMessageSize ( )

Gets the expected size of the message.

Returns

The integer containing the expected size of the message.

Generally used by the parser to determine when to stop parsing a message.

Definition at line 26 of file OregonScientificSensor.cpp.

```
26                                            {
27     return format[1];
28 }
```

4.4.3.4 uint8_t OregonScientificSensor::getSensorChannel ( )

Gets the channel that the sensor is on.

Returns

The channel id as an 8 bit integer.

This method is generally used by the Oregon Scientific class when searching for a sensor.

Definition at line 22 of file OregonScientificSensor.cpp.

```
22                                              {
23     return channel;
24 }
```

4.4.3.5 uint32_t OregonScientificSensor::getSensorID ( )

Gets the sensor ID.

Returns

the ID of the sensor as a 32 bit integer.

This method is generally used by the Oregon Scientific class when searching for sensors.

Definition at line 18 of file OregonScientificSensor.cpp.

```
18                                          {
19     return dev_id;
20 }
```

4.4.3.6 void OregonScientificSensor::makeJSONMessage (uint8_t ∗ *message*)

Creates a JSON message given the data in the standard Oregon Scientific format.

Parameters

∗*message*	The standard Oregon Scientific message.

Todo Shore up what the exact format will be for communicating with the server.

Definition at line 38 of file OregonScientificSensor.cpp.

```
38                                                          {
39     const char hexToChar[] = {’0’,’1’,’2’,’3’,’4’,’5’, ’6’, ’7’, ’8’, ’9’, ’A’, ’B’, ’C’, ’D’, ’E’, ’F’};
40     //String hexToChar(dt);
41      json_msg = "\"sensor_datum\":{";
42     // json_msg += titles[0];
43     // json_msg += "\":[";
44     /*for(uint8_t i = format[0]; i <= format[1]; i++){
45         json_msg+= hexToChar[message[i]];
46     }
47     json_msg += "\"}";*/
48     uint8_t formatCounter = 2;
49     for (int8_t i = 1; i < msg_size; i++)
50     {
51         json_msg += "\"";
52         json_msg += titles[i];
53         json_msg += "\":\"";
54         for(int8_t j = format[formatCounter+1]; j >= format[formatCounter]; j--){
55             json_msg +=hexToChar[message[j]];
56         }
57         json_msg += "\"";
58         if(i + 1 < msg_size){
59             json_msg += ",";
60         }
61         formatCounter+=2;
62     }
63     json_msg += "}";
64 }
```

4.4.4 Member Data Documentation

4.4.4.1 uint8_t OregonScientificSensor::channel [private]

Definition at line 117 of file OregonScientificSensor.h.

4.4.4.2 uint32_t OregonScientificSensor::dev_id [private]

The member variable holding the sensors device ID.

Definition at line 113 of file OregonScientificSensor.h.

4.4.4.3 uint8_t* OregonScientificSensor::format [private]

Definition at line 115 of file OregonScientificSensor.h.

4.4.4.4 String OregonScientificSensor::json_msg [private]

The member variable that holds the JSON message after it is created.

Definition at line 123 of file OregonScientificSensor.h.

4.4.4.5 uint8_t OregonScientificSensor::msg_size [private]

The member variable holding the expected size of the message.

Definition at line 119 of file OregonScientificSensor.h.

4.4.4.6 const uint8_t OregonScientificSensor::THGR122NX_FORMAT = { 0,18, 0,3, 4,4, 7,7, 8,11, 12,13, 15,16} [static]

The message format of the THGR122NX.

Definition at line 104 of file OregonScientificSensor.h.

4.4.4.7 const String OregonScientificSensor::THGR122NX_TITLES = {"THGR122NX", "DevID", "Channel", "Battery", "Temp", "Humidity", "Checksum"} [static]

The titles corresponding to the THGR122NX_FORMAT message format.

Definition at line 108 of file OregonScientificSensor.h.

4.4.4.8 const uint8_t OregonScientificSensor::THWR800_FORMAT ={ 0,15, 0,3, 4,4, 7,7, 8,11, 12,13} [static]

The message format of the THWR800.

Definition at line 106 of file OregonScientificSensor.h.

4.4.4.9 const String OregonScientificSensor::THWR800_TITLES = {"THWR800","DevID", "Channel", "Battery", "Temp", "Checksum"} [static]

The titles corresponding to the THWR800_FORMAT message format.

Definition at line 110 of file OregonScientificSensor.h.

4.4.4.10 String∗ OregonScientificSensor::titles [private]

The member array holding the titles corresponding to the format.

Definition at line 121 of file OregonScientificSensor.h.

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

- OregonScientificSensor/OregonScientificSensor.h
- OregonScientificSensor/OregonScientificSensor.cpp

Chapter 5

File Documentation

5.1 CC3000Operations.ino File Reference

CC3000 Operations contains the configuration settings of the CC3000 as well as the methods for using it to connect to the desired network.

Functions

- boolean connectToNetwork ()

  Controls the connection to the network including getting input from the user; whether that is from the Serial port or from a button press.
- void initMAC ()

  Initializes the address variable by reading the MAC address from the CC3000.
- void configure ()

  Configures the device by setting the encryption key and printing the devices MAC address so that it can be entered into the website for activation.
- boolean smartConfigCreate ()

  Attempts the SmartConfig Create feature of the CC3000.
- boolean displayConnectionDetails (void)

  Displays the connection details when the program is compiled in development mode.
- boolean smartConfigReconnect ()

  Attempts to reconnect to the previously used network.
- void mactoaddr (uint8_t ip[ ], char ∗string)

  Converts uint32t MAC address to its ASCII representation.
- char to_hex (uint8_t value)

  Converts uint8_t to hex char representation.
- boolean getButtonPress (int start)

  Polls for a button press while updating the LCD.
- void getSerialInput (char ∗data, uint16_t len, char delim)

  Gets a line of input from the serial port stopping at the desired delimiter.

5.1.1 Detailed Description

CC3000 Operations contains the configuration settings of the CC3000 as well as the methods for using it to connect to the desired network. This includes the ability to connect or reconnect using SmartConfig.

Definition in file CC3000Operations.ino.

5.1.2 Function Documentation

5.1.2.1 void configure ( )

Configures the device by setting the encryption key and printing the devices MAC address so that it can be entered into the website for activation.

This function should only be called when the config environment is defined. This should only occur when the device is first being programmed by setting the encryption key and accessing the devices MAC address.

Definition at line 97 of file CC3000Operations.ino.

```
97                   {
98    Serial.println(F("Configuration started.\nPlease make sure you are using a serial mode with newlines."));
99    // Checks for a valid previously stored encryption key
100    if(validEncryptionKey()){
101      // If one is found it will inform the user and verify that they want to keep it
102      Serial.println(F("\nOld encryption key found:"));
103      char buffer[32] = "";
104      getEncryptionKey(buffer);
105      Serial.println(buffer);
106      Serial.println(F("Overwrite? y/n"));
107    }
108    // Otherwise it will check if they want to create a new one
109    else {
110      Serial.println(F("\nEncryption key not found, make a new one? y/n"));
111    }
112    checkNPet();
113    while(!Serial.available()) {
114      delay(100);
115    }
116 
117    checkNPet();
118    char yesNo = Serial.read();
119    Serial.read(); //Get rid of newline
120    if(yesNo == ’Y’ || yesNo == ’y’) {
121      checkNPet();
122      setEncryptionKeyBySerial();
123    }
124 
125    Serial.println(F("Configuration finished, restarting."));
126    tinyWDT.force_reset();
127  }
```

5.1.2.2 boolean connectToNetwork ( )

Controls the connection to the network including getting input from the user; whether that is from the Serial port or from a button press.

returns: Whether the connection succeeded.

Definition at line 12 of file CC3000Operations.ino.

```
12                             {
13    checkNPet();
14  #ifdef CONFIG
15    configure();
16  #endif
17    // Tries to pet the watchdog
18   boolean valid = false;
19   // Loops while the not connected
20   while(!valid){
21     checkNPet();
22     // Outputs message for user
23     lcd_print_top("Press button for");
24     lcd_print_bottom("SmartConfig");
25 #ifdef DEVELOPMENT
26     Serial.println(F("Select an option."));
27     Serial.println(F("\t(1) SmartConfig Create"));
28     Serial.println(F("\t(2) SmartConfig Reconnect"));
29 #else
30     if(getButtonPress(USER_TIMEOUT)){
31       valid = smartConfigCreate();
32     }
33     else{
34       valid = smartConfigReconnect();
35     }
36 #endif
37     uint8_t numTrys = 0;
38     // While the user has not entered input wait until timeout
39 #ifdef DEVELOPMENT
40     while(!Serial.available()){
41       if(numTrys > 40){
42         Serial.println("Timeout");
43         return false;
44       }
45       // Keeps the watchdog from biting
46       checkNPet();
47       delay(200);
48       numTrys++;
49     }
50 
51     // Read the user input
52     char result = Serial.read();
53     // Reacts to user input
54     switch(result){
55     case '1':
56       return smartConfigCreate();
57     case '2':
58       return smartConfigReconnect();
59     default:
60       valid = false;
61     }
62 #else
63     return valid;
64 #endif
65   }
66 }
```

5.1.2.3 boolean displayConnectionDetails (void)

Displays the connection details when the program is compiled in development mode.

Definition at line 200 of file CC3000Operations.ino.

```
200                                         {
201   uint32_t addr, netmask, gateway, dhcpserv, dnsserv;
202 
203   if(!cc3000.getIPAddress(&addr, &netmask, &gateway, &dhcpserv, &dnsserv))
204     return false;
205 #if defined(DEVELOPMENT) || defined(CONFIG)
206   Serial.print(F("IP Addr: "));
207   cc3000.printIPdotsRev(addr);
208   Serial.print(F("\r\nNetmask: "));
209   cc3000.printIPdotsRev(netmask);
210   Serial.print(F("\r\nGateway: "));
211   cc3000.printIPdotsRev(gateway);
212   Serial.print(F("\r\nDHCPsrv: "));
213   cc3000.printIPdotsRev(dhcpserv);
214   Serial.print(F("\r\nDNSserv: "));
215   cc3000.printIPdotsRev(dnsserv);
216   Serial.println();
217 #endif
218   return true;
219 }
```

5.1.2.4 boolean getButtonPress (int *start*)

Polls for a button press while updating the LCD.

start: The value from which to count down.

Parameters

start	The value at which to start counting down from.

Returns

Whether the button was pressed at some point during the allotted time.

Definition at line 313 of file CC3000Operations.ino.

```
313                                  {
314   // Configures the pin connected to the button
315   pinMode(BUTTON_PIN, INPUT);
316   digitalWrite(BUTTON_PIN, INPUT_PULLUP);
317   // Initializes the variables
318   int thistime = millis();
319   int lasttime = millis();
320   // Counts down by one every second
321   for(int i = start; i >= 0; i--){
322     lcd_print_countdown(i);
323     // Stops polling every second to update the LCD
324     while(lasttime + 1000 > thistime){
325       thistime = millis();
326       // Checks to see if the button was pressed
327       if(digitalRead(BUTTON_PIN) == 0){
328         // Waits to debounce the button
329         delay(200);
330         if(digitalRead(BUTTON_PIN) == 0){
331           return true;
332         }
333       }
334     }
335     lasttime = thistime;
336   }
337   return false;
338 }
```

5.1.2.5 void getSerialInput (char ∗ *data*, uint16_t *len*, char *delim*)

Gets a line of input from the serial port stopping at the desired delimiter.

Parameters

∗*data*	A char buffer that will hold the input.
len	The max length of the buffer.
delim	The desired delimiter.

Definition at line 344 of file CC3000Operations.ino.

```
344                                                           {
345   // Gets rid of junk input etc. the newline char
346   while(Serial.available()){
347     Serial.read();
348   }
349   // Waits for valid input
350   while(!Serial.available()){
351     delay(200);
352   }
353   char temp = Serial.read();
354   uint16_t read_len = 0;
355   // Reads in data until it finds a newline
356   while(temp != delim && read_len < len){
357     data[read_len] = temp;
358     temp = Serial.read();
359     read_len++;
360   }
361   return;
362 }
```

5.1.2.6 void initMAC ( )

Initializes the address variable by reading the MAC address from the CC3000.

One consideration must be made in that this function should not be called before the CC3000 has been initialized by calling the begin() function.

Definition at line 73 of file CC3000Operations.ino.

```
73               {
74 #if defined(DEVELOPMENT) || defined(CONFIG)
75   Serial.print(F("Finding mac address ."));
76 #endif
77   uint8_t addr[6];
78   if(cc3000.getMacAddress(addr)){
79     mactoaddr(addr, address);
80   }
81   else{
82 #if defined(DEVELOPMENT) || defined(CONFIG)
83     Serial.println("Failed");
84 #endif
85   }
86 #if defined(DEVELOPMENT) || defined(CONFIG)
87   Serial.println(address);
88 #endif
89 }
```

5.1.2.7 void mactoaddr (uint8_t *ip[ ]*, char ∗ *string*)

Converts uint32t MAC address to its ASCII representation.

Parameters

ip	The numeric value of the MAC address that will be returned by the CC3000.
∗*string*	The buffer that will be used to hold the ASCII representation of the MAC address.

Definition at line 280 of file CC3000Operations.ino.

```
280                                           {
281   //Mac address to ascii
282   uint8_t idx = 0;
283   for(uint8_t i = 0; i < 6; i++) {
284     // i*2 is used more than once so only compute it once
285     idx = i * 2;
286     string[idx] = to_hex(ip[i] >> 4);
287     string[idx + 1] = to_hex(ip[i]);
288   }
289   string[12] = '\0';
290 }
```

5.1.2.8 boolean smartConfigCreate ( )

Attempts the SmartConfig Create feature of the CC3000.

Definition at line 130 of file CC3000Operations.ino.

```
                           {
#ifdef DEVELOPMENT
  Serial.println(F("\nInitializing the CC3000"));
#endif
  lcd_print_top("Enabling WiFi");
  // Initializes the CC3000
  if (!cc3000.begin(false))
  {
#ifdef DEVELOPMENT
    Serial.println("Enable Failed");
#endif
    lcd_print_bottom("Failed");
    return false;
  }

#ifdef DEVELOPMENT
  /* Try to use the smart config app (no AES encryption), saving */
  /* the connection details if we succeed. */
  Serial.println(F("Waiting for a SmartConfig connection (~60s) ..."));
#endif
  lcd_print_top("SmartConfig");
  lcd_print_bottom("Open App (~60s)");
  // Begins the smart config process
  if (!cc3000.startSmartConfig("CC3000"))
  {
#ifdef DEVELOPMENT
    Serial.println(F("SmartConfig failed"));
#endif
    lcd_print_bottom("Failed!");
    return false;
  }
  checkNPet();
  Serial.println(F("SmartConfig Success! AP connection details were saved"));
  lcd_print_bottom("Succeeded!");
#ifdef PRODUCTION
  delay(1000);
#endif
  uint16_t time = millis();

#ifdef DEVELOPMENT
  Serial.println(F("Request DHCP"));
#endif
  lcd_print_top("Requesting DHCP");
  // Requests DHCP
  while (!cc3000.checkDHCP()){
    checkNPet();
    if (millis() - time > DHCP_TIMEOUT) {
      time = 0;
#ifdef DEVELOPMENT
      Serial.println(F("DHCP failed!"));
#endif
      lcd_print_bottom("DHCPFailed!");
      return false;
    }
  }
#ifdef DEVELOPMENT
  Serial.println(F("DHCP Succeeded"));
#endif
  lcd_print_bottom("Succeeded!");
  // Prints out the connection details
  while(!displayConnectionDetails()){
    delay(1000);
  }
  // Initializes the MAC address
  //initMAC();
  return true;
}
```

5.1.2.9 boolean smartConfigReconnect ( )

Attempts to reconnect to the previously used network.

Definition at line 223 of file CC3000Operations.ino.

```
223                              {
224 #ifdef DEVELOPMENT
225   Serial.println(F("Attempting SmartConfig Reconnect"));
226 #endif
227   lcd_print_top("Reconnecting");
228   // Attempts to initialize the CC3000 and reconnect
229   if (!cc3000.begin(false, true, "CC3000")){
230 #ifdef DEVELOPMENT
231     Serial.println(F("Unable to re-connect!? Try Running SmartConfig Create"));
232 #endif
233     lcd_print_top("Reconnect Failed");
234     lcd_print_bottom("Try SmartConfig");
235     return false;
236   }
237   // Initializes the MAC address
238 
239   lcd_print_bottom("Reconnected");
240 #ifdef PRODUCTION
241   delay(1000);
242 #endif
243 #ifdef DEVELOPMENT
244   Serial.println(F("Reconnected!"));
245   // Wait for DHCP to complete
246   Serial.println(F("\nRequesting DHCP"));
247 #endif
248   lcd_print_bottom("Requesting DHCP");
249 
250   // Requests DHCP
251   uint16_t time = millis();
252   while (!cc3000.checkDHCP()) {
253     if (millis()-time > DHCP_TIMEOUT) {
254       lcd_print_bottom("DHCP Failed");
255 #ifdef DEVELOPMENT
256       Serial.println(F("DHCP failed!"));
257 #endif
258       return false;
259     }
260   }
261   lcd_print_bottom("DHCP Succeeded");
262 #ifdef DEVELOPMENT
263   Serial.println(F("DHCP Succeeded"));
264 
265   // Displays the connection details
266   while(!displayConnectionDetails()){
267     delay(1000);
268   }
269   //initMAC();
270 #endif
271   return true;
272 }
```

5.1.2.10 char to_hex (uint8_t *value*)

Converts uint8_t to hex char representation.

Parameters

value	The number to be converted to ASCII HEX

Returns

The ASCII representation of the HEX number value

Definition at line 296 of file CC3000Operations.ino.

```
296                            {
297   value &= 0xF;
298   // If it is greater than 9 add 55
299   // which will make 10 => ’A’ ...
300   if (value > 9) {
301     return (value + 55);
302   }
303   // Otherwise 0 => ’0’
304   else {
```

```
305     return (value + '0');
306   }
307 }
```

5.2 encryption.ino File Reference

Contains the encryption methods that are used to secure the data as it is transmitted over the network.

Functions

- void encrypt (char ∗plaintext, char ∗key, char ∗encrypted)

  The encryption method implementing a Vignere cipher.
- void decrypt (char ∗encrypted, char ∗key, char ∗plaintext)

  The decryption method for returning the data back to plaintext.
- void setEncryptionKeyBySerial ()

  Sets the encryption key that will be used by the encryption and decryption methods.

5.2.1 Detailed Description

Contains the encryption methods that are used to secure the data as it is transmitted over the network. It also contains a function to set the encryption key via the serial port.

Definition in file encryption.ino.

5.2.2 Function Documentation

5.2.2.1 void decrypt (char ∗ *encrypted,* char ∗ *key,* char ∗ *plaintext*)

The decryption method for returning the data back to plaintext.

Parameters

∗*encrypted*	The encrypted data.
∗*key*	The encryption key that will be used to decrypt the cipher text.
∗*plaintext*	The buffer in which the resulting decrypted data will be placed.

Definition at line 27 of file encryption.ino.

```
27                                                          {
28   int textLength = strlen(encrypted);
29   int keyLength =  strlen(key);
30   for(int i=0; i < textLength; i++) {
31     plaintext[i] = encrypted[i] - (key[i % keyLength] - 32);
32     if(plaintext[i] < 32 || plaintext[i] >= 127) {
33       plaintext[i] += (127-32);
34     }
35   }
36 }
```

5.2.2.2 void encrypt (char ∗ *plaintext,* char ∗ *key,* char ∗ *encrypted*)

The encryption method implementing a Vignere cipher.

Parameters

∗*plaintext*	The plaintext data that will be encrypted.
∗*key*	The key that will be used by the cipher to encrypt the data.
∗*encrypted*	The buffer in which the resulting encrypted data will be placed.

Definition at line 11 of file encryption.ino.

```
11                                                              {
12   int textLength = strlen(plaintext);
13   int keyLength =  strlen(key);
14   for(int i=0; i < textLength; i++) {
15     encrypted[i] = plaintext[i] + key[i % keyLength] - 32;
16     if((unsigned) encrypted[i] >= 127) {
17       encrypted[i] −= (unsigned) (127-32);
18     }
19   }
20 }
```

5.2.2.3 void setEncryptionKeyBySerial ( )

Sets the encryption key that will be used by the encryption and decryption methods.

Definition at line 39 of file encryption.ino.

```
39                                 {
40   Serial.println(F("\nPlease type in new encryption key. (<32 characters)"));
41 
42   boolean done = false;
43   char buffer[32] = "";
44   int index = 0;
45   char c = ’ ’;  //’ ’ is an arbitrary value
46 
47   while(c != ’\n’ && c != ’\0’) { //Until newline
48     checkNPet();
49     while(Serial.available()) {
50       c = Serial.read();
51       if(c == ’\n’) {
52         buffer[index] = ’\0’;
53         break;
54       }
55       buffer[index] = c;
56       index++;
57     }
58   }
59 
60   Serial.println(F("Your new encryption key is:"));
61   Serial.println(buffer);
62   Serial.println(F("Is this OK? y/n"));
63   while(!Serial.available()){}
64   c = Serial.read();
65   if(c == ’Y’ || c == ’y’) {
66     setEncryptionKey(buffer);
67     Serial.println(F("Key saved."));
68   } else {
69     while(Serial.available())
70     { Serial.read(); }
71     setEncryptionKeyBySerial();
72   }
73 }
```

5.3 header.h File Reference

Contains the definitions that are required for the program.

Macros

- #define PRODUCTION

  Defined for the production environment.
- #define PACKET_SIZE 50

  The size of the individual packets //Number of datapoints in a packet.
- #define DATA_MAX_LENGTH (PACKET_SIZE ∗ 35 + 150)

  The maximum length of the data.
- #define MAX_PACKET_LENGTH (160 + DATA_MAX_LENGTH + 64)

  The maximum packet length - used when allocating the packet buffer.
- #define SERIAL_BAUD 115200

  The Baud Rate of the Serial port.
- #define LISTEN_PORT 3000

  The port on which the server listens.
- #define IDLE_TIMEOUT_MS 3000

  The HTTP timeout (in milliseconds)
- #define DHT22_PIN A0

  The input from the DHT22.
- #define HOST "192.168.1.16"

  The Ruby on Rails host.
- #define DHCP_TIMEOUT 10000

  DHCP timeout (in milliseconds).
- #define USER_TIMEOUT 5

  The time to wait for user input during setup. (in seconds)
- #define BUTTON_PIN A3

  The input pin connected to the button.
- #define LCD_RS A2

  The pin used for the Read Select line for the LCD.
- #define LCD_E A1

  The pin used for the Enable line.
- #define LCD_D4 4

  The pin used for the data bus line 4.
- #define LCD_D5 5

  The pin used for the data bus line 5.
- #define LCD_D6 6

  The pin used for the data bus line 6.
- #define LCD_D7 8

  The pin used for the data bus line 7.

5.3.1 Detailed Description

Contains the definitions that are required for the program. These allow the functionality of the program to be modified in a simple manner. This includes the ability to change pin mappings, environments, and several other useful parameters.

Definition in file header.h.

5.3.2 Macro Definition Documentation

5.3.2.1 #define BUTTON_PIN A3

The input pin connected to the button.

Definition at line 30 of file header.h.

5.3.2.2 #define DATA_MAX_LENGTH (PACKET_SIZE ∗ 35 + 150)

The maximum length of the data.

Definition at line 15 of file header.h.

5.3.2.3 #define DHCP_TIMEOUT 10000

DHCP timeout (in milliseconds).

Definition at line 26 of file header.h.

5.3.2.4 #define DHT22_PIN A0

The input from the DHT22.

Definition at line 21 of file header.h.

5.3.2.5 #define HOST "192.168.1.16"

The Ruby on Rails host.

Definition at line 24 of file header.h.

5.3.2.6 #define IDLE_TIMEOUT_MS 3000

The HTTP timeout (in milliseconds)

Definition at line 20 of file header.h.

5.3.2.7 #define LCD_D4 4

The pin used for the data bus line 4.

Definition at line 37 of file header.h.

5.3.2.8 #define LCD_D5 5

The pin used for the data bus line 5.

Definition at line 38 of file header.h.

5.3.2.9 #define LCD_D6 6

The pin used for the data bus line 6.

Definition at line 39 of file header.h.

5.3.2.10 #define LCD_D7 8

The pin used for the data bus line 7.

Definition at line 40 of file header.h.

5.3.2.11 #define LCD_E A1

The pin used for the Enable line.

Definition at line 36 of file header.h.

5.3.2.12 #define LCD_RS A2

The pin used for the Read Select line for the LCD.

Definition at line 35 of file header.h.

5.3.2.13 #define LISTEN_PORT 3000

The port on which the server listens.

Definition at line 19 of file header.h.

5.3.2.14 #define MAX_PACKET_LENGTH (160 + DATA_MAX_LENGTH + 64)

The maximum packet length - used when allocating the packet buffer.

Definition at line 16 of file header.h.

5.3.2.15 #define PACKET_SIZE 50

The size of the individual packets //Number of datapoints in a packet.

Definition at line 11 of file header.h.

5.3.2.16 #define PRODUCTION

Defined for the production environment.

Definition at line 8 of file header.h.

5.3.2.17 #define SERIAL_BAUD 115200

The Baud Rate of the Serial port.

Definition at line 18 of file header.h.

5.3.2.18 #define USER_TIMEOUT 5

The time to wait for user input during setup. (in seconds)

Definition at line 28 of file header.h.

5.4 LCDHelper.ino File Reference

Contains the helper functions for the LCD.

Functions

- void lcd_print_top (char ∗message)

  Clears the entire screen then prints the specified string on the top line.
- void lcd_print_bottom (char ∗message)

  Clears the bottom line of the LCD screen before printing the desired message there.
- void lcd_print_countdown (uint16_t val)

  Prints a countdown in the bottom left corner of the LCD display.
- void lcd_print_dht22 (double temp, double humid)

  A helper function that will print the DHT22 temperature and humidity to the LCD.

5.4.1 Detailed Description

Contains the helper functions for the LCD.

Definition in file LCDHelper.ino.

5.4.2 Function Documentation

5.4.2.1 void lcd_print_bottom (char ∗ *message*)

Clears the bottom line of the LCD screen before printing the desired message there.

Parameters

message	The message to be printed on the bottom line of the LCD.

Definition at line 14 of file LCDHelper.ino.

```
14                                      {
15   // Clears the bottom row of the lcd
16   lcd.setCursor(0,1);
17   lcd.print("                ");
18   // Prints the message
19   lcd.setCursor(0,1);
20   lcd.print(message);
21 }
```

5.4.2.2 void lcd_print_countdown (uint16_t *val*)

Prints a countdown in the bottom left corner of the LCD display.

Parameters

val	The value to be displayed in the bottom left corner.

Definition at line 25 of file LCDHelper.ino.

```
25                                       {
26   uint8_t cursor_pos;
27   if(val > 9){
28     cursor_pos = 14;
29   }
30   else if(val > 100){
31     cursor_pos = 13;
32   }
33   else{
34     cursor_pos = 15;
35   }
36   lcd.setCursor(12, 1);
37   lcd.print("    ");
38   lcd.setCursor(cursor_pos, 1);
39   lcd.print(val);
40 }
```

5.4.2.3 void lcd_print_dht22 (double *temp,* double *humid*)

A helper function that will print the DHT22 temperature and humidity to the LCD.

Parameters

temp	The temperature provided by the DHT22.
humid	The humidity provided by the DHT22.

Definition at line 45 of file LCDHelper.ino.

```
45                                                {
46   lcd.clear();
47   lcd.setCursor(0,0);
48   lcd.print("Temp: ");
49   lcd.print(temp);
50   lcd.setCursor(0,1);
51   lcd.print("Humid: ");
52   lcd.print(humid);
53 }
```

5.4.2.4 void lcd_print_top (char ∗ *message*)

Clears the entire screen then prints the specified string on the top line.

Parameters

message	The message to be displayed on the top line of the screen.

Definition at line 7 of file LCDHelper.ino.

```
7                                    {
8   lcd.clear();
9   lcd.print(message);
10 }
```

5.5 ManchesterDecoder/ManchesterDecoder.cpp File Reference

```
#include <ManchesterDecoder.h>
```

5.6 ManchesterDecoder/ManchesterDecoder.h File Reference

This Manchester Decoder class is spcifically designed to decode messages from Oregon Scientific Sensors, and has been tested on both version 2.1 and version 3.0 protocols.

```
#include <Arduino.h>
#include <WordBuffer.h>
```

Classes

- class ManchesterDecoder

Macros

- #define LONG_PULSE 1

  Defines a long pulse as 1.
- #define SHORT_PULSE 0

  Defines a shor pulse as 0.
- #define DEFAULT_SIZE 1024u

  Defines the default size of the input and output buffers.
- #define RESET 0xFFu

  Defines reset as 0xFF so the parser will know that the decoder timed out.
- #define ONE 0x08u

  Defines One as 0x80 so it can be shifted into the variable.
- #define ZERO 0x00u

  Defines Zero as 0x00 for obvious reasons.

5.6.1 Detailed Description

This Manchester Decoder class is spcifically designed to decode messages from Oregon Scientific Sensors, and has been tested on both version 2.1 and version 3.0 protocols. It has not been tested with anything other than those devices. However it should work, or only need minor modifications to work. One consideration that must be made is that the code was designed to be used with a device that supports interrupts on the specified pin.

Author

Joel D. Sabol

Date

June 2014

Definition in file ManchesterDecoder.h.

5.6.2 Macro Definition Documentation

5.6.2.1 #define DEFAULT_SIZE 1024u

Defines the default size of the input and output buffers.

Definition at line 37 of file ManchesterDecoder.h.

5.6.2.2 #define LONG_PULSE 1

Defines a long pulse as 1.

Definition at line 34 of file ManchesterDecoder.h.

5.6.2.3 #define ONE 0x08u

Defines One as 0x80 so it can be shifted into the variable.

Definition at line 40 of file ManchesterDecoder.h.

5.6.2.4 #define RESET 0xFFu

Defines reset as 0xFF so the parser will know that the decoder timed out.

Definition at line 39 of file ManchesterDecoder.h.

5.6.2.5 #define SHORT_PULSE 0

Defines a shor pulse as 0.

Definition at line 35 of file ManchesterDecoder.h.

5.6.2.6 #define ZERO 0x00u

Defines Zero as 0x00 for obvious reasons.

Definition at line 41 of file ManchesterDecoder.h.

5.7 memory_management.ino File Reference

Contains the memory management function which control how the device interfaces with the EEPROM.

```
#include <avr/eeprom.h>
```

Macros

- #define SAVE_SPACE 1000
- #define ENCRYPTION_MAGIC_NUM_LOC ((byte ∗) 0)
- #define ENCRYPTION_KEY_PTR ((byte ∗) ENCRYPTION_MAGIC_NUM_LOC+1)
- #define MAGIC_NUM_LOC ((byte ∗) ENCRYPTION_KEY_PTR + 32)
- #define MAGIC_NUM_VAL 'D'
- #define EXPERIMENT_PTR ((uint16_t ∗) MAGIC_NUM_LOC+1)
- #define SENT_PTR (EXPERIMENT_PTR + 1)
- #define SAVE_START (SENT_PTR + sizeof(void ∗))
- #define SAVE_END (SAVE_START + (SAVE_SPACE ∗ sizeof(long int)))
- #define ILLEGAL_VALUE 65535
- #define SENT() eeprom_read_word(SENT_PTR)
- #define RECORD_SIZE (sizeof(long int) + sizeof(int))

- #define PPM_AT(n) ((uint16_t ∗)((n)∗RECORD_SIZE + SAVE_START))
- #define TIME_AT(n) ((uint32_t ∗)((n)∗RECORD_SIZE + SAVE_START + sizeof(int)))

Functions

- uint16_t savedValues ()

  Gets the number of saved values in the devices EEPROM.
- boolean outOfSpace (void)

  Determines whether or not there is space left.
- boolean hasMoreData (void)

  Determines whether there is more data to send.
- int mostRecentDataAvg (int numToAverage=5)
- void nextDatum (int &ppm, long &timestamp)

  Gets the next data-point from the devices EEPROM.
- void prevDataNotSent ()

  Allows sending of data that has already been read.
- void dataSent ()

  Records whether the data was sent or not to the devices EEPROM.
- void clearData ()

  Deletes all of the data and reconfigures the memory.
- boolean validMemory ()

  Checks for valid memory.
- int getExperimentId ()

  Gets the experiment id stored in the devices EEPROM.
- void setExperimentId (int experiment_id)
- boolean validEncryptionKey ()

  Checks to see if a valid encryption key has been stored in the devices EEPROM.
- void getEncryptionKey (char ∗buffer)
- void setEncryptionKey (char ∗key)

  Sets the encryption key by writing the value to the devices EEPROM.

Variables

- uint16_t dataRead = SENT()
- uint16_t savedCounter = 0
- boolean invalidMemory = false

5.7.1 Detailed Description

Contains the memory management function which control how the device interfaces with the EEPROM.

Definition in file memory_management.ino.

5.7.2 Macro Definition Documentation

5.7.2.1 #define ENCRYPTION_KEY_PTR ((byte ∗) ENCRYPTION_MAGIC_NUM_LOC+1)

Definition at line 18 of file memory_management.ino.

5.7.2.2 #define ENCRYPTION_MAGIC_NUM_LOC ((byte ∗) 0)

Definition at line 16 of file memory_management.ino.

5.7.2.3 #define EXPERIMENT_PTR ((uint16_t ∗) MAGIC_NUM_LOC+1)

Definition at line 23 of file memory_management.ino.

5.7.2.4 #define ILLEGAL_VALUE 65535

Definition at line 29 of file memory_management.ino.

5.7.2.5 #define MAGIC_NUM_LOC ((byte ∗) ENCRYPTION_KEY_PTR + 32)

Definition at line 20 of file memory_management.ino.

5.7.2.6 #define MAGIC_NUM_VAL 'D'

Definition at line 21 of file memory_management.ino.

5.7.2.7 #define PPM_AT(*n*) ((uint16_t ∗)((n)∗RECORD_SIZE + SAVE_START))

Definition at line 37 of file memory_management.ino.

5.7.2.8 #define RECORD_SIZE (sizeof(long int) + sizeof(int))

Definition at line 34 of file memory_management.ino.

5.7.2.9 #define SAVE_END (SAVE_START + (SAVE_SPACE ∗ sizeof(long int)))

Definition at line 27 of file memory_management.ino.

5.7.2.10 #define SAVE_SPACE 1000

Definition at line 14 of file memory_management.ino.

5.7.2.11 #define SAVE_START (SENT_PTR + sizeof(void ∗))

Definition at line 26 of file memory_management.ino.

5.7.2.12 #define SENT( ) eeprom_read_word(SENT_PTR)

Definition at line 32 of file memory_management.ino.

5.7.2.13 #define SENT_PTR (EXPERIMENT_PTR + 1)

Definition at line 24 of file memory_management.ino.

5.7.2.14 #define TIME_AT(n) ((uint32_t *)((n)*RECORD_SIZE + SAVE_START + sizeof(int)))

Definition at line 38 of file memory_management.ino.

5.7.3 Function Documentation

5.7.3.1 void clearData ( )

Deletes all of the data and reconfigures the memory.

Definition at line 151 of file memory_management.ino.

```
151                  {
152   eeprom_write_word( SENT_PTR, 0);
153   eeprom_write_word( PPM_AT(0), ILLEGAL_VALUE); //'removing' the saved data
154   eeprom_write_byte( MAGIC_NUM_LOC, MAGIC_NUM_VAL);
155   dataRead = SENT();//0
156   setExperimentId(0);
157 
158   //Verify newly written memory
159   if( eeprom_read_word( SENT_PTR ) != 0
160    || eeprom_read_word(PPM_AT(0)) != ILLEGAL_VALUE
161    || eeprom_read_byte(MAGIC_NUM_LOC) != MAGIC_NUM_VAL) {
162      invalidMemory = true;
163   }
164 }
```

5.7.3.2 void dataSent ( )

Records whether the data was sent or not to the devices EEPROM.

Definition at line 142 of file memory_management.ino.

```
142                 {
143   eeprom_write_word(SENT_PTR, dataRead);
144 
145   if(eeprom_read_word(SENT_PTR) != dataRead)
146   {  invalidMemory = true; }
147 }
```

5.7.3.3 void getEncryptionKey (char * *buffer*)

Definition at line 189 of file memory_management.ino.

```
189                                     {
190   char c;
191   int i = 0;
192   do {
193     c = eeprom_read_byte(ENCRYPTION_KEY_PTR + i);
194     buffer[i] = c;
195     i++;
196   } while(c != '\0');
197 }
```

5.7.3.4 int getExperimentId ( )

Gets the experiment id stored in the devices EEPROM.

Definition at line 172 of file memory_management.ino.

```
172                       {
173   return (int) eeprom_read_word(EXPERIMENT_PTR);
174 }
```

5.7.3.5 boolean hasMoreData (void)

Determines whether there is more data to send.

Definition at line 100 of file memory_management.ino.

```
100                           {
101   //Determines whether there are more data to send
102   uint16_t saved = savedValues();
103 
104   return (saved > dataRead ? true : false);
105 }
```

5.7.3.6 int mostRecentDataAvg (int *numToAverage* = 5)

Definition at line 108 of file memory_management.ino.

```
108                                             {
109   uint16_t saved = savedValues();
110   numToAverage = (numToAverage > saved) ? saved : numToAverage;
111 
112   unsigned long sum = 0;
113   for(int i = 1; i <= numToAverage; i++) {
114     sum += eeprom_read_word(PPM_AT(saved-i));
115   }
116 
117   return sum / numToAverage;
118 }
```

5.7.3.7 void nextDatum (int & *ppm*, long & *timestamp*)

Gets the next data-point from the devices EEPROM.

Parameters

&ppm	The address of the memory location containing the ppm value.
&timestamp	The address of the memory location containing the timestamp value.

Definition at line 123 of file memory_management.ino.

```
123                                           {
124   //Gets the next data point
125 
126   ppm =       eeprom_read_word( PPM_AT( dataRead));
127   timestamp = eeprom_read_dword(TIME_AT(dataRead));
128 
129   dataRead++;
130 
131   return;
132 }
```

5.7.3.8 boolean outOfSpace (void)

Determines whether or not there is space left.

Returns

Whether the devices EEPROM is full or not.

Definition at line 88 of file memory_management.ino.

```
88                            {
89   uint16_t saved = savedValues();
90
91   if( saved >= SAVE_SPACE) {
92     Serial.println(F("Out of space"));
93     return true;
94   } else {
95     return false;
96   }
97 }
```

5.7.3.9 void prevDataNotSent ( )

Allows sending of data that has already been read.

Definition at line 135 of file memory_management.ino.

```
135                         {
136   //Allows sending of data that has been read already
137   dataRead = SENT();
138   return;
139 }
```

5.7.3.10 uint16_t savedValues ( ) [inline]

Gets the number of saved values in the devices EEPROM.

Returns

The number of saved values in the devices EEPROM.

Definition at line 51 of file memory_management.ino.

```
51                                 {
52   //Uses memoization
53
54   while(eeprom_read_word(PPM_AT(savedCounter)) ==
      ILLEGAL_VALUE)
55     { savedCounter++; }
56   return savedCounter;
57 }
```

5.7.3.11 void setEncryptionKey (char * *key*)

Sets the encryption key by writing the value to the devices EEPROM.

Definition at line 200 of file memory_management.ino.

```
200                                  {
201   int keyLength = strlen(key) > 32 ? 32 : strlen(key);
202   int i;
203   for(i = 0; i < keyLength; i++) {
204     eeprom_write_byte(ENCRYPTION_KEY_PTR + i, key[i]);
205
206     if(eeprom_read_byte(ENCRYPTION_KEY_PTR+i) != key[i]) {
207       //Validate newly set byte
208       invalidMemory = true;
209     }
210   }
211   eeprom_write_byte(ENCRYPTION_KEY_PTR + i, '\0');
212
213   eeprom_write_byte(ENCRYPTION_MAGIC_NUM_LOC,
      MAGIC_NUM_VAL);
214
215   //Validate '\0' && magic number
216   if( eeprom_read_byte(ENCRYPTION_KEY_PTR + i)    != '\0'
217    || eeprom_read_byte(ENCRYPTION_MAGIC_NUM_LOC) !=
      MAGIC_NUM_VAL) {
218     invalidMemory = true;
219    }
220 }
```

5.7.3.12 void setExperimentId (int *experiment_id*)

Definition at line 175 of file memory_management.ino.

```
175                                         {
176   eeprom_write_word(EXPERIMENT_PTR, (uint16_t) experiment_id);
177
178   //Verify newly written memory
179   if(eeprom_read_word(EXPERIMENT_PTR) != (uint16_t) experiment_id)
180   {  invalidMemory = true; }
181 }
```

5.7.3.13 boolean validEncryptionKey ( )

Checks to see if a valid encryption key has been stored in the devices EEPROM.

Returns

Whether or not there was a valid key.

Definition at line 185 of file memory_management.ino.

```
185                              {
186   return eeprom_read_byte(ENCRYPTION_MAGIC_NUM_LOC) ==
      MAGIC_NUM_VAL;
187 }
```

5.7.3.14 boolean validMemory ( )

Checks for valid memory.

Definition at line 167 of file memory_management.ino.

```
167                       {
168   return eeprom_read_byte(MAGIC_NUM_LOC) == MAGIC_NUM_VAL && !
      invalidMemory;
169 }
```

5.7.4 Variable Documentation

5.7.4.1 uint16_t dataRead = SENT()

Definition at line 40 of file memory_management.ino.

5.7.4.2 boolean invalidMemory = false

Definition at line 44 of file memory_management.ino.

5.7.4.3 uint16_t savedCounter = 0

Definition at line 42 of file memory_management.ino.

5.8 OregonScientific/OregonScientific.cpp File Reference

```
#include <OregonScientific.h>
```

5.9 OregonScientific/OregonScientific.h File Reference

```
#include <Arduino.h>
#include <OregonScientificSensor.h>
```

Classes

- class OregonScientific

  OregonScientific defines a parser capable of parsing both version 2.1 and version 3.0 messages from Oregon Scientific sensors.

Macros

- #define SYNC_NIBBLE 0

  Defines the location of the sync nibble in the message.
- #define OSCV_3 0x33

  Defines the version 3.0 protocol.
- #define OSCV_2_1 0x21

  Defines the version 2.1 protocol.
- #define DEFAULT_SIZE 32

  Defines the size of the message if none is provided.
- #define MAX_SENSOR_NUM 10

  Defines the maximum number of sensors that the parser will listen for.
- #define DEV_ID_BEGIN 0

  Defines the location of the start of the device id in the message.

- #define DEV_ID_END 3

  Defines the location of the end of the device id in the message.
- #define CHANNEL_NIBBLE 4

  Defines the location of the channel nibble in the message.
- #define ROLLING_CODE_BEGIN 5

  Defines the beginning of the rolling code in the message.
- #define ROLLING_CODE_END 6

  Defines the end of the rolling code in the message.
- #define FLAGS 7

  Defines where the flags are in the message.
- #define MESSAGE_BEGIN 8

  Defines the location of the data segment in the message.

Enumerations

- enum OregonScientific_ParseState { SYNCING, GET_ID, GET_MSG, DONE }

5.9.1 Macro Definition Documentation

5.9.1.1 #define CHANNEL_NIBBLE 4

Defines the location of the channel nibble in the message.

Definition at line 22 of file OregonScientific.h.

5.9.1.2 #define DEFAULT_SIZE 32

Defines the size of the message if none is provided.

Definition at line 18 of file OregonScientific.h.

5.9.1.3 #define DEV_ID_BEGIN 0

Defines the location of the start of the device id in the message.

Definition at line 20 of file OregonScientific.h.

5.9.1.4 #define DEV_ID_END 3

Defines the location of the end of the device id in the message.

Definition at line 21 of file OregonScientific.h.

5.9.1.5 #define FLAGS 7

Defines where the flags are in the message.

Definition at line 25 of file OregonScientific.h.

5.9.1.6 #define MAX_SENSOR_NUM 10

Defines the maximum number of sensors that the parser will listen for.

Definition at line 19 of file OregonScientific.h.

5.9.1.7 #define MESSAGE_BEGIN 8

Defines the location of the data segment in the message.

Definition at line 26 of file OregonScientific.h.

5.9.1.8 #define OSCV_2_1 0x21

Defines the version 2.1 protocol.

Definition at line 17 of file OregonScientific.h.

5.9.1.9 #define OSCV_3 0x33

Defines the version 3.0 protocol.

Definition at line 16 of file OregonScientific.h.

5.9.1.10 #define ROLLING_CODE_BEGIN 5

Defines the beginning of the rolling code in the message.

Definition at line 23 of file OregonScientific.h.

5.9.1.11 #define ROLLING_CODE_END 6

Defines the end of the rolling code in the message.

Definition at line 24 of file OregonScientific.h.

5.9.1.12 #define SYNC_NIBBLE 0

Defines the location of the sync nibble in the message.

Definition at line 15 of file OregonScientific.h.

5.9.2 Enumeration Type Documentation

5.9.2.1 enum OregonScientific_ParseState

Enumerator

SYNCING The parser is in this state while looking for the sync nibble which will determine where the other elements will be.

GET_ID The parser is in this state while parsing the device id.

GET_MSG The parser is in this state while parsing the message.

DONE The parser is in this state upon completion of parsing the message.

Definition at line 30 of file OregonScientific.h.

```
{
 SYNCING,
 GET_ID,
 GET_MSG,
 DONE
};
```

5.10 OregonScientificExample.ino File Reference

Oregon Scientific Example is the main program that handles the configuration and the main loop of the program.

```
#include <WildFire.h>
#include <WildFire_CC3000.h>
#include <WordBuffer.h>
#include <ManchesterDecoder.h>
#include <OregonScientific.h>
#include <OregonScientificSensor.h>
#include <LiquidCrystal.h>
#include <dht.h>
#include <SPI.h>
#include <TinyWatchdog.h>
#include "header.h"
```

Enumerations

- enum device_states { PING_SERVER, ACTIVATED, GEN_SENSOR }

Functions

- LiquidCrystal lcd (LCD_RS, LCD_E, LCD_D4, LCD_D5, LCD_D6, LCD_D7)

  The instantiation of the LCD.
- void resetParser ()

  Resets the both version protocol parsers.
- void stringToByteArr (String msg, char ∗data)

  Converts the string to its byte array representation.
- void readDHT22 ()

  Reads the DHT22 until it returns a valid reading then prints out that reading.
- String assembleDHT22JSON ()

  Assembles the necessary information to create the JSON string that contains the data from the DHT22.
- void checkNPet ()

  Checks to see if the watchdog timer needs to be petted and pets it if enough time has elapsed.
- void generateDeviceJSON (String sensor_msg, char ∗msg)

  Generates the JSON object that contains the data as well as the building id and the device address.
- void processMessages ()

  Processes the data as it comes from the Manchester Decoder and passes it to the parser to be interpreted.

- void setup ()

  Performs all of the initializations along with all of the necessary configurations.
- void loop ()

  The main loop that controls the operation of the device by using a state machine.

Variables

- WildFire wf

  The instantiation of the WildFire.
- WildFire_CC3000 cc3000

  The instantiation of the CC3000 radio.
- TinyWatchdog tinyWDT

  The Watchdog Timer.
- ManchesterDecoder md

  The Manchester Decoder.
- OregonScientific oscv3

  The Oregon Scientific Version 3.0 Parser.
- OregonScientific oscv2

  Oregon Scientific Version 2.1 parser.
- char packet_buffer [MAX_PACKET_LENGTH]

  The packet buffer used to hold the packet.
- char address [13]

  The hard-coded device address: This really should not be this way, however there is an issue with reading the MAC address from the CC3000 which when run keeps it from connecting to the server via TCP.
- int building_id = -1

  The variable holding the id of the building that the device is currently in.
- dht dht22

  The instantiation of the DHT22 object.
- uint32_t ip

  The variable holding the IP address of the server.
- uint16_t current_time

  Stores the time that the checkNPet function is called at so a comparison can be made.
- uint16_t time_last_pet

  Stores the time at which the watchdog timer was last petted.

5.10.1 Detailed Description

Oregon Scientific Example is the main program that handles the configuration and the main loop of the program.

Todo Add the ability to dynamicall add and remove sensors.

Definition in file OregonScientificExample.ino.

5.10.2 Enumeration Type Documentation

5.10.2.1 enum device_states

Enumerator

PING_SERVER In this state the device will check. the server for information regarding what sensors it has as well as where to send the sensor datum to.

ACTIVATED In this state the device will send the data. from the sensors to the URI specified by the immediately after receiving data from the sensors.

GEN_SENSOR In this state the device will generate the. sensors that the sever told it to listen for.

Definition at line 44 of file OregonScientificExample.ino.

```
44                  {
45   PING_SERVER,
46   ACTIVATED,
49   GEN_SENSOR
52 };
```

5.10.3 Function Documentation

5.10.3.1 String assembleDHT22JSON ( )

Assembles the necessary information to create the JSON string that contains the data from the DHT22.

Definition at line 90 of file OregonScientificExample.ino.

```
90                          {
91   char msg[DATA_MAX_LENGTH] = {
92     ’\0’       };
93   String temp = "\"sensor_datum\":{\"Temp\":\"";
94   temp += (int)dht22.temperature * 10;
95   temp += "\", \"Channel\":\"22\", \"DevID\":\"DHT\",";
96   temp += "\"humidity\":\"";
97   temp += (int)dht22.humidity;
98   temp += "\"}";
99   return temp;
100 }
```

5.10.3.2 void checkNPet ( )

Checks to see if the watchdog timer needs to be petted and pets it if enough time has elapsed.

Definition at line 104 of file OregonScientificExample.ino.

```
104                 {
105   current_time = millis();
106   if(current_time - time_last_pet >= 2000){
107 #ifdef DEVELOPMENT
108     Serial.print(".");
109 #endif
110     tinyWDT.pet();
111     time_last_pet = current_time;
112   }
113 }
```

5.10.3.3 void generateDeviceJSON (String *sensor_msg,* char ∗ *msg*)

Generates the JSON object that contains the data as well as the building id and the device address.

Definition at line 116 of file OregonScientificExample.ino.

```
116                                                     {
117   String js = "{";
118   js += sensor_msg;
119   js += ",\"building_id\":\"";
120   js += building_id;
121   js += "\",\"device_address\":\"";
122   js += address;
123   js += "\"}";
124   lcd_print_bottom("Got Message");
125   js.toCharArray(msg, js.length()+1);
126 }
```

5.10.3.4 LiquidCrystal lcd (LCD_RS , LCD_E , LCD_D4 , LCD_D5 , LCD_D6 , LCD_D7)

The instantiation of the LCD.

5.10.3.5 void loop ( )

The main loop that controls the operation of the device by using a state machine.

Definition at line 216 of file OregonScientificExample.ino.

```
216            {
217   device_states state = PING_SERVER;
218 #ifdef DEVELOPMENT
219   Serial.println("Listening on 433.92Mhz");
220 #endif
221   while(1){
222     checkNPet();
223     switch(state){
224     case PING_SERVER:
225       lcd_print_top("Querying Server");
226       building_id = getBuilding();
227 #ifdef DEVELOPMENT
228       Serial.print("B_ID");
229       Serial.println(building_id);
230 #endif
231       if(building_id > 0){
232 #ifdef DEVELOPMENT
233         Serial.println("Activated");
234 #endif
235         lcd_print_top("Activated");
236         state = ACTIVATED;
237       }
238       else{
239         lcd_print_top("Not In Building");
240         lcd_print_bottom("Add to Building");
241         delay(10000);
242       }
243       break;
244     case ACTIVATED:
245       building_id = getBuilding();
246       if(building_id < 0){
247         state = PING_SERVER;
248         lcd_print_top("Deactivated");
249       }
250       else{
251         processMessages();
252         delay(10000);
253       }
254       break;
255     case GEN_SENSOR:
256     break;
257     }
258   }
259 }
```

5.10.3.6 void processMessages ( )

Processes the data as it comes from the Manchester Decoder and passes it to the parser to be interpreted.

Definition at line 130 of file OregonScientificExample.ino.

```
130                       {
131   while(md.hasNextPulse()){
132     uint8_t data = md.getNextPulse();
133     //Serial.print(data, HEX);
134     // If value indicates timeout then resetParser
135     if(data == RESET){
136       resetParser();
137     } // Otherwise put the data in both parsers
138     else{
139       char payload[DATA_MAX_LENGTH] = {
140         '\0'             };
141       if(oscv3.parseOregonScientificV3(data)){
142         // Gets the sensor that broad-casted the message and print it.
143         lcd_print_top("Got Message");
144         generateDeviceJSON(oscv3.getCurrentSensor()->
      getJSONMessage(), payload);
145         assemblePacket(payload);
146         lcd_print_top("Sent Message");
147         resetParser();
148         //readDHT22();
149         //generateDeviceJSON(assembleDHT22JSON(), payload);
150         //assemblePacket(payload);
151       }
152       else if(oscv2.parseOregonScientificV2(data)){
153         generateDeviceJSON(oscv2.getCurrentSensor()->
      getJSONMessage(), payload);
154         assemblePacket(payload);
155         resetParser();
156       }
157     }
158   }
159   //Serial.println();
160 }
```

5.10.3.7 void readDHT22 ( )

Reads the DHT22 until it returns a valid reading then prints out that reading.

Definition at line 76 of file OregonScientificExample.ino.

```
76                  {
77 #ifdef DEVELOPMENT
78   Serial.print("DHT22, \t");
79 #endif
80   while(dht22.read22(DHT22_PIN) != DHTLIB_OK);
81 #ifdef DEVELOPMENT
82   Serial.print(dht22.humidity, 1);
83   Serial.print(",\t");
84   Serial.println(dht22.temperature, 1);
85 #endif
86   lcd_print_dht22(dht22.temperature, dht22.humidity);
87 }
```

5.10.3.8 void resetParser ( )

Resets the both version protocol parsers.

Definition at line 57 of file OregonScientificExample.ino.

```
57                    {
58   oscv3.reset();
59   oscv2.reset();
60 }
```

5.10.3.9 void setup ( )

Performs all of the initializations along with all of the necessary configurations.

Definition at line 163 of file OregonScientificExample.ino.

```
163              {
164   // Initializes the WildFire
165   wf.begin();
166   // Configures the WDT and check method
167   time_last_pet = 0;
168   tinyWDT.begin(1000, 60000);
169   lcd.begin(16,2);
170   lcd.clear();
171   lcd_print_top("Welcome to");
172   lcd_print_bottom("Home Monitor");
173   delay(1000);
174   lcd_print_top("Performing Setup");
175 #if defined(DEVELOPMENT) || defined(CONFIG)
176   Serial.begin(SERIAL_BAUD);
177   // Output compile information and server information
178   Serial.println(F("Compiled on " __DATE__ ", " __TIME__));
179   Serial.println(F("Server is " HOST));
180 #endif
181   for(uint16_t i = 0; i < MAX_PACKET_LENGTH; i++){
182     packet_buffer[i] = '\0';
183   }
184   // IF the connection attempts to the network fail sleep
185   if(!connectToNetwork()){
186     // TODO put the wildfire to sleep
187     while(1);
188   }
189
190   checkNPet();
191
192   // Resolve the IP address of the server
193   ip = 0;
194   while (ip == 0) {
195
196     if (! cc3000.getHostByName(HOST, &ip)) {
197 #ifdef DEVELOPMENT
198       Serial.println(F("Couldn't resolve!"));
199 #endif
200     }
201     delay(500);
202   }
203   checkNPet();
204 #ifdef DEVELOPMENT
205   Serial.println("Resolved the server");
206 #endif
207   // Adds sensors with the appropriate message formats
208   oscv2.addSensor(new OregonScientificSensor(
      THGR122NX, V2_CHANNEL_1, 7,
      OregonScientificSensor::THGR122NX_FORMAT,
      OregonScientificSensor::THGR122NX_TITLES));
209   oscv2.addSensor(new OregonScientificSensor(
      THGR122NX, V2_CHANNEL_3, 7,
      OregonScientificSensor::THGR122NX_FORMAT,
      OregonScientificSensor::THGR122NX_TITLES));
210   oscv3.addSensor(new OregonScientificSensor(
      THWR800, V2_CHANNEL_1, 6,
      OregonScientificSensor::THWR800_FORMAT,
      OregonScientificSensor::THWR800_TITLES));
211
212   lcd_print_top("Listening 492Mhz");
213 }
```

5.10.3.10 void stringToByteArr (String *msg,* char ∗ *data*)

Converts the string to its byte array representation.

Parameters

msg	The string to be converted.
∗*data*	The buffer to place the bytes of the string into.

Definition at line 65 of file OregonScientificExample.ino.

```
65                                              {
66   uint16_t len = msg.length();
67   for(int i = 0; i < len; i++){
68     data[i] = msg.charAt(i);
69   }
70   data[len] = '\0';
71 }
```

5.10.4 Variable Documentation

5.10.4.1 char address[13]

Initial value:

```
= {
  '0','8','0','0','2','8','5','7','5','A','0','E'}
```

The hard-coded device address: This really should not be this way, however there is an issue with reading the MAC address from the CC3000 which when run keeps it from connecting to the server via TCP.

Definition at line 29 of file OregonScientificExample.ino.

5.10.4.2 int building_id = -1

The variable holding the id of the building that the device is currently in.

Definition at line 32 of file OregonScientificExample.ino.

5.10.4.3 WildFire_CC3000 cc3000

The instantiation of the CC3000 radio.

Definition at line 21 of file OregonScientificExample.ino.

5.10.4.4 uint16_t current_time

Stores the time that the checkNPet function is called at so a comparison can be made.

Definition at line 40 of file OregonScientificExample.ino.

5.10.4.5 dht dht22

The instantiation of the DHT22 object.

Definition at line 36 of file OregonScientificExample.ino.

5.10.4.6 uint32_t ip

The variable holding the IP address of the server.

Definition at line 38 of file OregonScientificExample.ino.

5.10.4.7 ManchesterDecoder md

The Manchester Decoder.

Definition at line 23 of file OregonScientificExample.ino.

5.10.4.8 OregonScientific oscv2

Oregon Scientific Version 2.1 parser.

Definition at line 25 of file OregonScientificExample.ino.

5.10.4.9 OregonScientific oscv3

The Oregon Scientific Version 3.0 Parser.

Definition at line 24 of file OregonScientificExample.ino.

5.10.4.10 char packet_buffer[MAX_PACKET_LENGTH]

The packet buffer used to hold the packet.

Definition at line 27 of file OregonScientificExample.ino.

5.10.4.11 uint16_t time_last_pet

Stores the time at which the watchdog timer was last petted.

Definition at line 41 of file OregonScientificExample.ino.

5.10.4.12 TinyWatchdog tinyWDT

The Watchdog Timer.

Definition at line 22 of file OregonScientificExample.ino.

5.10.4.13 WildFire wf

The instantiation of the WildFire.

Definition at line 20 of file OregonScientificExample.ino.

5.11 OregonScientificSensor/OregonScientificSensor.cpp File Reference

```
#include <OregonScientificSensor.h>
```

5.12 OregonScientificSensor/OregonScientificSensor.h File Reference

```
#include <Arduino.h>
```

Classes

- union id_type

  A union for ease of converting between the array and integer representations of the device ID.
- class OregonScientificSensor

  A class that encompasses the necessary information about Oregon Scientific sensors and the methods for accessing that information.

Macros

- #define BTHR918 0x050A050D

  Device ID code for the Oregon Scientific BTHR918.
- #define BTHR968 0x050D0600

  Device ID code for the Oregon Scientific BTHR968.
- #define PCR800 0x02090104

  Device ID code for the Oregon Scientific PCR800.
- #define RGR918 0x020A010D

  Device ID code for the Oregon Scientific RGR918.
- #define RGR968 0x020D0100

  Device ID code for the Oregon Scientific RGR968.
- #define STR918 0x030A000D

  Device ID code for the Oregon Scientific STR918.
- #define THGN123N 0x010D0200

  Device ID code for the Oregon Scientific THGN123N.
- #define THGN801 0x0F080204

  Device ID code for the Oregon Scientific THGN801.
- #define THGR122NX 0x010D0200

  Device ID code for the Oregon Scientific THGR122NX.
- #define THGR228N 0x010A020D

  Device ID code for the Oregon Scientific THGR228N.
- #define THGR810 0x0F080204

  Device ID code for the Oregon Scientific THGR810.
- #define THGR8101 0x0F080B04

  Device ID code for the Oregon Scientific THGR8101.
- #define THGR918 0x010A030D

  Device ID code for the Oregon Scientific THGR918.

- #define THN132N 0x0E0C0400

  Device ID code for the Oregon Scientific THN132N.
- #define THR238NF 0x0E0C0400

  Device ID code for the Oregon Scientific THR238NF.
- #define THWR288A 0x0E0A040C

  Device ID code for the Oregon Scientific THWR288A.
- #define THWR800 0x0C080404

  Device ID code for the Oregon Scientific THWR800.
- #define UVN800 0x0D080704

  Device ID code for the Oregon Scientific UVN800.
- #define UVR128 0x0E0C0700

  Device ID code for the Oregon Scientific UVR128.
- #define WGR8002 0x01090904

  Device ID code for the Oregon Scientific WGR8002.
- #define WGR8003 0x01090804

  Device ID code for the Oregon Scientific WGR8003.
- #define WGR918 0x030A000D

  Device ID code for the Oregon Scientific WGR918.
- #define V2_CHANNEL_1 0x01

  Protocol version 2.1 Channel 1.
- #define V2_CHANNEL_2 0x02

  Protocol version 2.1 Channel 2.
- #define V2_CHANNEL_3 0x04

  Protocol version 2.1 Channel 3.
- #define V3_CHANNEL_1 0x01

  Protocol version 3.0 Channel 1.
- #define V3_CHANNEL_2 0x02

  Protocol version 3.0 Channel 2.
- #define V3_CHANNEL_3 0x03

  Protocol version 3.0 Channel 3.

5.12.1 Macro Definition Documentation

5.12.1.1 #define BTHR918 0x050A050D

Device ID code for the Oregon Scientific BTHR918.

Definition at line 8 of file OregonScientificSensor.h.

5.12.1.2 #define BTHR968 0x050D0600

Device ID code for the Oregon Scientific BTHR968.

Definition at line 9 of file OregonScientificSensor.h.

5.12.1.3 #define PCR800 0x02090104

Device ID code for the Oregon Scientific PCR800.

Definition at line 10 of file OregonScientificSensor.h.

5.12.1.4 #define RGR918 0x020A010D

Device ID code for the Oregon Scientific RGR918.

Definition at line 11 of file OregonScientificSensor.h.

5.12.1.5 #define RGR968 0x020D0100

Device ID code for the Oregon Scientific RGR968.

Definition at line 12 of file OregonScientificSensor.h.

5.12.1.6 #define STR918 0x030A000D

Device ID code for the Oregon Scientific STR918.

Definition at line 13 of file OregonScientificSensor.h.

5.12.1.7 #define THGN123N 0x010D0200

Device ID code for the Oregon Scientific THGN123N.

Definition at line 14 of file OregonScientificSensor.h.

5.12.1.8 #define THGN801 0x0F080204

Device ID code for the Oregon Scientific THGN801.

Definition at line 15 of file OregonScientificSensor.h.

5.12.1.9 #define THGR122NX 0x010D0200

Device ID code for the Oregon Scientific THGR122NX.

Definition at line 16 of file OregonScientificSensor.h.

5.12.1.10 #define THGR228N 0x010A020D

Device ID code for the Oregon Scientific THGR228N.

Definition at line 17 of file OregonScientificSensor.h.

5.12.1.11 #define THGR810 0x0F080204

Device ID code for the Oregon Scientific THGR810.

Definition at line 18 of file OregonScientificSensor.h.

5.12.1.12 #define THGR8101 0x0F080B04

Device ID code for the Oregon Scientific THGR8101.

Definition at line 19 of file OregonScientificSensor.h.

5.12.1.13 #define THGR918 0x010A030D

Device ID code for the Oregon Scientific THGR918.

Definition at line 20 of file OregonScientificSensor.h.

5.12.1.14 #define THN132N 0x0E0C0400

Device ID code for the Oregon Scientific THN132N.

Definition at line 21 of file OregonScientificSensor.h.

5.12.1.15 #define THR238NF 0x0E0C0400

Device ID code for the Oregon Scientific THR238NF.

Definition at line 22 of file OregonScientificSensor.h.

5.12.1.16 #define THWR288A 0x0E0A040C

Device ID code for the Oregon Scientific THWR288A.

Definition at line 23 of file OregonScientificSensor.h.

5.12.1.17 #define THWR800 0x0C080404

Device ID code for the Oregon Scientific THWR800.

Definition at line 24 of file OregonScientificSensor.h.

5.12.1.18 #define UVN800 0x0D080704

Device ID code for the Oregon Scientific UVN800.

Definition at line 25 of file OregonScientificSensor.h.

5.12.1.19 #define UVR128 0x0E0C0700

Device ID code for the Oregon Scientific UVR128.

Definition at line 26 of file OregonScientificSensor.h.

5.12.1.20 #define V2_CHANNEL_1 0x01

Protocol version 2.1 Channel 1.

Definition at line 32 of file OregonScientificSensor.h.

5.12.1.21 #define V2_CHANNEL_2 0x02

Protocol version 2.1 Channel 2.

Definition at line 33 of file OregonScientificSensor.h.

5.12.1.22 #define V2_CHANNEL_3 0x04

Protocol version 2.1 Channel 3.

Definition at line 34 of file OregonScientificSensor.h.

5.12.1.23 #define V3_CHANNEL_1 0x01

Protocol version 3.0 Channel 1.

Definition at line 37 of file OregonScientificSensor.h.

5.12.1.24 #define V3_CHANNEL_2 0x02

Protocol version 3.0 Channel 2.

Definition at line 38 of file OregonScientificSensor.h.

5.12.1.25 #define V3_CHANNEL_3 0x03

Protocol version 3.0 Channel 3.

Definition at line 39 of file OregonScientificSensor.h.

5.12.1.26 #define WGR8002 0x01090904

Device ID code for the Oregon Scientific WGR8002.

Definition at line 27 of file OregonScientificSensor.h.

5.12.1.27 #define WGR8003 0x01090804

Device ID code for the Oregon Scientific WGR8003.

Definition at line 28 of file OregonScientificSensor.h.

5.12.1.28 #define WGR918 0x030A000D

Device ID code for the Oregon Scientific WGR918.

Definition at line 29 of file OregonScientificSensor.h.

5.13 ServerOperations.ino File Reference

Contains the methods that interface with the server.

```
#include <string.h>
#include <stdlib.h>
```

Functions

- void assemblePacket (char ∗data)

  Assembles the HTTP packet that will be sent to the server.
- char ∗ makePacketHeader (char ∗request_type_and_location, char ∗mime_type, int datalength)

  Generates the packet header given the necessary information.
- boolean sendPacket ()

  Establishes the TCP connection between the device and the server then sends the packet and reads the servers response.
- int getBuilding ()

  Checks whether the device is activated inside a building and if so what building and what sensors does it have.
- void clearPacketBuffer ()

  The helper function used to clear the packet buffer to ensure that there is no chance of buffer overflow.

5.13.1 Detailed Description

Contains the methods that interface with the server. This includes assembling the HTTP packet that is sent to the server, establishing the connection to the server, and sending the packet with the CC3000.

Definition in file ServerOperations.ino.

5.13.2 Function Documentation

5.13.2.1 void assemblePacket (char ∗ *data*)

Assembles the HTTP packet that will be sent to the server.

Parameters

∗*data*	The sensor data that will form the payload of the packet.

Definition at line 12 of file ServerOperations.ino.

```
12                                  {
13   lcd_print_top("Assembling Packet");
14   // Clears the packet buffer to make sure that it is not dirty
15   clearPacketBuffer();
16
17   // Gets the encryption key
18   char vignere_key[32] = "";
19   getEncryptionKey(vignere_key);
20
21   // Encrypts the data using the encryption key
22   encrypt(data, vignere_key, data);
23
24   // Begins creating parts of the header
25   char putstr_buffer[64] = "POST /sensor_data/batch_create/";
26   strcat(putstr_buffer,address);
27   strcat_P(putstr_buffer, PSTR(".json HTTP/1.1"));
28
29   int additionalCharacters = 17; // the brackets, :, and "s
30   //Account for characters that will be escaped
31   uint8_t len = strlen(data);
32   for(int i=0; i<len; i++) {
33     if(data[i] == ’\\’ || data[i]==’"’)
34       additionalCharacters++;
35   }
36
37   // Completes the header
38   makePacketHeader(putstr_buffer, "application/json", len + additionalCharacters);
39   strcat_P(packet_buffer, PSTR("\n{\"encrypted\":\""));
40
41   //Copy encrypted text and escape " and \s
42   int packetSize = strlen(packet_buffer);
43   for(int i=0; i<len; i++) {
44     if(data[i] == ’"’ || data[i] == ’\\’) {
45       packet_buffer[packetSize] = ’\\’;
46       packetSize++;
47     }
48     packet_buffer[packetSize] = data[i];
49     packetSize++;
50   }
51   packet_buffer[packetSize] = ’\0’;
52 
53   strcat_P(packet_buffer, PSTR("\"}"));
54 
55   Serial.println(packet_buffer);
56   sendPacket();
57 }
```

5.13.2.2 void clearPacketBuffer ( )

The helper function used to clear the packet buffer to ensure that there is no chance of buffer overflow.

Definition at line 269 of file ServerOperations.ino.

```
269                          {
270   for(int i = 0; i < strlen(packet_buffer); i++){
271     packet_buffer[i] = ’\0’;
272   }
273 }
```

5.13.2.3 int getBuilding ( )

Checks whether the device is activated inside a building and if so what building and what sensors does it have.

Returns

The building id if the device is currently active in a building otherwise it will return -1.

Receiving reply

Definition at line 155 of file ServerOperations.ino.

```
155                    {
156   clearPacketBuffer();
157   Serial.println(F("Connecting to server...\nIf this is the first time, it may take a while"));
158   checkNPet();
159   Serial.println("Radio Connected");
160   WildFire_CC3000_Client client = cc3000.connectTCP(ip, LISTEN_PORT);
161   Serial.println(F("Established TCP Connection"));
162   int datalength = 0;
163   char data[1] = "";
164 
165 
166   //Sending request
167   char putstr_buffer[128] = "GET /first_contact/";
168   strcat(putstr_buffer, address);
169   Serial.print("Address is:");
170   Serial.println(address);
171 
172   strcat_P(putstr_buffer, PSTR(".html HTTP/1.1"));
173   makePacketHeader(putstr_buffer, "application/json", datalength);
174 
175   strcat(packet_buffer, data);
176   Serial.println(F("Sending request"));
177   checkNPet();
178   Serial.println(packet_buffer);
179   if(client.connected()){
180     client.fastrprintln(packet_buffer);
181   }
```

```
  else{
    Serial.println("Error");
  }
  Serial.print("Address is: ");
  Serial.println(address);
  checkNPet();

  char serverReply[512] = "";

  Serial.println(F("Getting Server reply"));

  //Ignoring the header:
  int i = 0;
  while(client.connected() && i < 511) {
    while(client.available()) {
      Serial.print(’*’);
      checkNPet();
      serverReply[i] = (char)client.read();
      i++;
      serverReply[i] = ’\0’;
      if(i >= 5 && !strcmp("start", serverReply+i-5)) {
        break;
      }
    }


    if(i >= 5 && !strcmp("start", serverReply+i-5)) {
      break;
    }
  }


  //Reading the body
  i=0;
  while(client.connected() && i < 511){
    Serial.print(’.’);
    if(client.available()) {
      checkNPet();
      serverReply[i] = (char)client.read();
      i = (i == 0 && ( serverReply[0] == ’ ’ || serverReply[0] == ’\n’) ) ?  i : i+1;
    }
    else {
      //delay(50);
    }

    if(i >= 3 && !strcmp("end", serverReply+i-3)) {
      i -= 3;
      break;
    }
  }
  serverReply[i] = ’\0’;

#ifdef DEVELOPMENT
  /* Serial.println("\nPacket to server:");
   Serial.println(packet_buffer);
   Serial.println("ServerReply:");
   Serial.println(serverReply);*/
#endif

  //Decoding server reply
  char vignere_key[32] = "";
  getEncryptionKey(vignere_key);
  decrypt(serverReply, vignere_key, serverReply);
  Serial.println();
  Serial.println(serverReply);

  long int time;
  int experiment_id_tmp, CO2_cutoff_tmp;
  int varsRead = sscanf(serverReply, "%ld %*s %d %*s %d", &time, &experiment_id_tmp, &CO2_cutoff_tmp);

  client.close();

  switch(varsRead){
  case 1:
    return -1;
    break;
  case 2:
  case 3:
    return experiment_id_tmp;
    break;
  default:
```

```
264     return -1;
265   }
266 }
```

5.13.2.4 char∗ makePacketHeader (char ∗ *request_type_and_location,* char ∗ *mime_type,* int *datalength*)

Generates the packet header given the necessary information.

Parameters

∗*request_type_-and_location*	Specifies the HTTP method as well as the URI.
∗*mime_type*	Specifies the type of data that the packet will be carrying (application-json) etc.
datalength	The length of the data that will be encapsulated in the content section of the packet.

Returns

The packet buffer containing the header.

Definition at line 64 of file ServerOperations.ino.

```
64                                                                                      {
65   char len_buffer[32] = "";
66   itoa(datalength, len_buffer, 10);
67   //packet_buffer[0] = '\0';
68   strcat(packet_buffer, request_type_and_location);
69   strcat_P(packet_buffer, PSTR("\nHost: " HOST "\nContent-Type: "));
70   strcat(packet_buffer, mime_type);
71   strcat_P(packet_buffer, PSTR("; charset=UTF-8\nContent-Length: "));
72   strcat(packet_buffer, len_buffer);
73   strcat_P(packet_buffer, PSTR("\nConnection: close\n"));
74 
75   return packet_buffer;
76 
77 }
```

5.13.2.5 boolean sendPacket ( )

Establishes the TCP connection between the device and the server then sends the packet and reads the servers response.

Returns

The whether or not the packet was successfully sent.

Definition at line 81 of file ServerOperations.ino.

```
81                     {
82   //Creates and sends a packet of data to the server containing CO2 results and timestamps
83   Serial.println(F("Sending data..."));
84   lcd_print_top("Sending Data");
85   WildFire_CC3000_Client client = cc3000.connectTCP(ip, LISTEN_PORT);
86   Serial.println("Established TCP Connection");
87   Serial.println(F("Connected"));
88   lcd_print_bottom("Connected");
89 #ifdef DEVELOPMENT
90 
91 #endif
92 
93   if (client.connected()) {
94     //Send packet
95     checkNPet();
```

```
   while(client.available()) { //flushing input buffer, just in case
     Serial.println(client.read());
   }

    client.fastrprintln(packet_buffer);
    Serial.println(F("Printed"));
    //#ifdef DEVELOPMENT

    Serial.println("Outgoing request: ");
    Serial.println(packet_buffer);
    Serial.println();
  }
  //#endif
  Serial.println(F("Packet sent.\nWaiting for response."));
  checkNPet();

  int timeLeft = 6000;
  char headerBuffer[7] = {
    0,0,0,0,0,0,0                   };
  while(timeLeft) {
    if(!strcmp("start\n", headerBuffer) && client.available()) {
      //When the header is over, and there is one character from the actual body
      break;
    }

    if(client.available()) {
      //Add the new character to the end of headerBuffer
      for(int i=0; i<5; i++) {
        headerBuffer[i] = headerBuffer[i+1];
      }
      headerBuffer[5] = client.read();
      Serial.print(headerBuffer[5]);
    }
    else {
      delay(50);
      timeLeft -= 50;
    }
  } //End ignoring header
  lcd_print_top("Listening 492Mhz");
  checkNPet();
  if(client.read() != 'S') {
    Serial.println(F("Upload failed"));
    //if uploading succeeded, the server will display a page that says "Success uploading data".
    // otherwise, it will show "Failed to upload"
    //On a timeout, client.read() gives -1.
    return false;
  }
  else {
    Serial.println(F("Upload succeeded"));
  }


  client.close();
  Serial.println("client closed");
  return false;
}
```

Index