**Turbidity Sensor & Modular Sensors Library**

By Aaron Nightingale (night030@umn.edu)

**Summary:**

The turbidity sensor made for the spring 2018 senior design project can communicate with any device running the modular sensors library (like the MayFly data logger). This is accomplished using the “Turbidity.h” and “TurbidityHelper.h” classes. This document will describe how to set up this sensor with the modular sensors library. In addition, this document will explain various functions in each class, and how to use them.

**Adding a Turbidity Sensor**

This section will walk you through adding a turbidity sensor to the data logger code. See “MayFly-Logging => turbidity\_logging => turbidity\_logging.ino” for an example of using the turbidity sensor with the modular sensors library on the MayFly data logger (In the “Code Backups” folder on the Google drive).

1. Change the logging interval to be 10 minutes or more. For example:

const float loggingInterval = 10.0;

1. Include the “Turbidity.h” and “TurbidityHelper.h” classes:

#include "Turbidity.h"

#include "TurbidityHelper.h"

1. Add these constants to define the sensor’s address and the power pin:
   * Note: The turbidity sensor’s slave I2C address can be changed by altering the code found in “turbidity\_slave.ino”.

uint8\_t Turbidityi2c\_addr = 0x08;

const int I2CPower = 22;

1. Create new “Turbidity” and “TurbidityHelper” objects as follows:
   * Note: The turbidity class had to be split up into two separate classes. This is because the turbidity sensor logs 7 different variables, and there is a limit of 4 variables per class in the modular sensors library.

Turbidity ntu01(I2CPower, Turbidityi2c\_addr);

TurbidityHelper ntu02(I2CPower, Turbidityi2c\_addr);

1. Add the following objects to the “variableList” array. These will be logged to an SD card.

new Turbidity\_FS(&ntu01),

new Turbidity\_SS(&ntu01),

new Turbidity\_DARK(&ntu01),

new Turbidity\_NTU(&ntu01),

new TurbidityHelper\_FS(&ntu02),

new TurbidityHelper\_SS(&ntu02),

new TurbidityHelper\_DARK(&ntu02),

1. Congratulations! The turbidity sensor is now ready to be used with the MayFly. Upload the program to the MayFly, and connect the turbidity sensor to the data logger as shown in figure 1 below:

|  |  |  |
| --- | --- | --- |
| **Data Logger Connection** | **Turbidity Sensor Connection** | **Purpose** |
| +3.3v | Red | Sensor power |
| GND | Black | Sensor ground |
| (SDA) | Blue | I2C data line |
| (SCL) | Yellow | I2C clock line |
| N / A | Green | Sensor busy pin (Optional) |

*Figure 1 - Connections between the data logger and the turbidity sensor*

**“Turbidity.h” Class Reference**

**Summary**

The “Turbidity.h” class is responsible for reading data from the turbidity sensor, telling the turbidity sensor to take a new measurement (update), and logging four of seven variables. This class is used in conjunction with “TurbidityHelper.h” to log all of the turbidity sensor’s data. The constants and functions in this class are described below:

**Constants**

* #define TURBIDITY\_NUM\_VARIABLES 4
  + This defines the number of variables in the “Turbidity.h” class. These four variables will be logged to an SD card.
* #define TURBIDITY\_WARM\_UP\_TIME\_MS 100
  + This defines the number of milliseconds necessary for the turbidity sensor to properly start up. 100ms was chosen as a “safe guestimate”.
* #define TURBIDITY\_STABILIZATION\_TIME\_MS 0
  + This is the stabilization time for the turbidity sensor. This is unused in our setup.
* #define TURBIDITY\_MEASUREMENT\_TIME\_MS 1000
  + This parameter defines the measurement time of the sensor. The default value of 1s was used.
* #define TURBIDITY\_FS\_RESOLUTION 2
  + This parameter sets the precision of the logged variable. In this case, the Turbidity\_FS variable will be logged as a float with 2 digits past the decimal point.

* #define TURBIDITY\_FS\_VAR\_NUM 0
  + This constant defines which variable is in position 0 of the variable array. In this case, the forward scattered light variable (from LED #1) is in position 0.
* #define TURBIDITY\_SS\_RESOLUTION 2
  + This parameter sets the precision of the logged variable. In this case, the Turbidity\_SS variable will be logged as a float with 2 digits past the decimal point.
* #define TURBIDITY\_SS\_VAR\_NUM 1
  + This constant defines which variable is in position 1 of the variable array. In this case, the side scattered light variable (from LED #1) is in position 1.
* #define TURBIDITY\_DARK\_RESOLUTION 2
  + This parameter sets the precision of the logged variable. In this case, the Turbidity\_Dark variable will be logged as a float with 2 digits past the decimal point.
* #define TURBIDITY\_DARK\_VAR\_NUM 2
  + This constant defines which variable is in position 2 of the variable array. In this case, the dark counts variable (from sensor #1) is in position 2.
* #define TURBIDITY\_NTU\_RESOLUTION 2
  + This parameter sets the precision of the logged variable. In this case, the Turbidity\_NTU variable will be logged as a float with 2 digits past the decimal point.
* #define TURBIDITY\_NTU\_VAR\_NUM 3
  + This constant defines which variable is in position 3 of the variable array. In this case, the turbidity variable (NTU) is in position 3.
* #define DATA\_LEN 6
  + This constant defines the size of the response from the turbidity sensor. The turbidity sensor sends 6 float-type variables, causing the data length constant to be 6.
* #define COMMAND\_LEN 7
  + The data logger can send various commands to the turbidity sensor. The commands are sent as a character array, and the length of said array is 7 bytes. Note that the last byte must be a zero.
* #define RES\_DARK1 0
  + The turbidity sensor returns an array of 6 float-type variables to the datalogger. The float at position 0 of this array is the number of dark counts from sensor #1. The units for this are Hertz.
* #define RES\_TRAN1 2
  + The turbidity sensor returns an array of 6 float-type variables to the datalogger. The float at position 2 of this array is the number of transmission counts from LED #1. The units for this are Hertz.
* #define RES\_SCAT1 3
  + The turbidity sensor returns an array of 6 float-type variables to the datalogger. The float at position 3 of this array is the number of side scatter counts from LED #1. The units for this are Hertz.
* #define RES\_DARK2 1
  + The turbidity sensor returns an array of 6 float-type variables to the datalogger. The float at position 1 of this array is the number of dark counts from sensor #2. The units for this are Hertz.
* #define RES\_TRAN2 4
  + The turbidity sensor returns an array of 6 float-type variables to the datalogger. The float at position 4 of this array is the number of transmission counts from LED #2. The units for this are Hertz.
* #define RES\_SCAT2 5
  + The turbidity sensor returns an array of 6 float-type variables to the datalogger. The float at position 5 of this array is the number of side scatter counts from LED #2. The units for this are Hertz.

**Class Members**

* uint8\_t \_i2cAddressHex;
  + This variable contains the I2C address of the turbidity slave sensor.
* char command[COMMAND\_LEN];
  + This array contains the command that will be sent to the turbidity sensor. This array is a character array.
* float response[DATA\_LEN];
  + This array contains the response from the turbidity sensor. This array consists of 6 float-type variables. Using these 6 variables, the turbidity of the water can be calculated. The variable at each position of the array is given by the “RES\_XXX” constants. (Note that RES stands for RESponse array).

**Class Methods**

* Turbidity(int powerPin, uint8\_t i2cAddressHex = 0x08, int

measurementsToAverage = 1);

* + This is the constructor for the “Turbidity.h” class. In this function, the “i2cAddressHex” variable is saved to the “\_i2cAddressHex” member variable.
* bool wake(void) override;
  + This function is executed whenever the sensor is “waked” after a period of being idle. This function basically makes sure the sensor is powered on, and that the sensor has waited the proper warm up time.
* SENSOR\_STATUS setup(void) override;
  + This function is executed whenever the sensor is set up for the first time. This function doesn’t do anything, and only returns the value of “getStatus()”.
* SENSOR\_STATUS getStatus(void) override;
  + This function makes sure that the sensor is powered on. Then, this function checks to make sure that there is an I2C slave device at the “\_i2cAddressHex” address. If there are any errors in either of these processes, the “SENSOR\_ERROR” flag is returned. If the sensor was set up successfully, the “SENSOR\_READY” flag is returned.
* String getSensorLocation(void) override;
  + This function returns a string containing the I2C address of the sensor. For example, “I2C\_0x08”.
* bool addSingleMeasurementResult(void) override;
  + This function begins by calling the “sensorRead()” function. This function fills the “response[]” array with the current measurements from the turbidity sensor. Next, the values from the response array are copied to the “sensorValues[]” array. This allows the readings to be logged later. Afterwards, the “updateCommand()” is called to tell the sensor to take a new reading. This measurement process takes up to 8 minutes. Finally, the function returns true upon completion.

* bool sensorPresent();
  + This function verifies that a sensor is present at the current I2C address specified by the variable “\_i2cAddressHex”. If a sensor is not present, this function returns false. If a sensor is present, the function returns true.
* void sensorRead();
  + This function reads data from the slave turbidity sensor over I2C. This is done in a byte by byte process, and the results are stored in the float-type “response[]” array.
* void sensorWrite();
  + This function sends the contents of the “command[]” array to the slave turbidity sensor. By changing the contents of the “command[]” array, different commands can be sent to the turbidity sensor. For example, if the array consists of “{‘U’,’P’,’D’,’A’,’T’,’E’,0}, then the update command is issued and the turbidity sensor begins taking a new measurement.
* void updateCommand();
  + This function sets the contents of the “command[]” array to be {‘U’,’P’,’D’,’A’,’T’,’E’,0}. Then the “sensorWrite()” function is called to send the update command to the turbidity sensor. This command tells the turbidity sensor to start taking a new measurement.
* float calculate\_turbidity(float data[6]);
  + The “response[]” array should be passed to this function. This function calculates the turbidity using the data in the “response[]” array. The turbidity is returned as a float-type variable. This function can be changed to more accurately calibrate the turbidity sensor.

**“TurbidityHelper.h” Class Reference**

**Summary**

The turbidity sensor needs to log 7 variables to an SD card. However, the modular sensors library only allows 4 variables to be logged per sensor. To get around this, the “TurbidityHelper.h” class is used. This class is responsible for logging the side scatter counts from LED #2, the forward transmission counts from LED #2, and the dark counts from sensor #2. This class is very similar to the “Turbidity.h” class.

**Constants**

* #define TURBIDITY\_HELPER\_NUM\_VARIABLES 3
  + This constant defines the number of variables that will be logged to an SD card using the modular sensors library.
* #define TURBIDITY\_HELPER\_WARM\_UP\_TIME\_MS 100
  + This parameter defines the warm up time (in milliseconds) for the turbidity sensor. The default value of 100ms was used.
* #define TURBIDITY\_HELPER\_STABILIZATION\_TIME\_MS 0
  + This parameter defines the stabilization time (in milliseconds) for the turbidity sensor. The default value of 0ms was used.
* #define TURBIDITY\_HELPER\_MEASUREMENT\_TIME\_MS 1000
  + This constant defines the measurement time in milliseconds. The default value of 1s was used.
* #define TURBIDITY\_HELPER\_FS\_RESOLUTION 2
  + This constant defines how many digits will be after the decimal point when saving the forward scattered light measurement to an SD card.
* #define TURBIDITY\_HELPER\_FS\_VAR\_NUM 0
  + This constant defines which variable is in position 0 of the variable array. In this case, the forward scatter counts variable (from LED #2) is in position 2.
* #define TURBIDITY\_HELPER\_SS\_RESOLUTION 2
  + This constant defines how many digits will be after the decimal point when saving the side scattered light measurement to an SD card.
* #define TURBIDITY\_HELPER\_SS\_VAR\_NUM 1
  + This constant defines which variable is in position 1 of the variable array. In this case, the side scatter counts variable (from LED #2) is in position 1.
* #define TURBIDITY\_HELPER\_DARK\_RESOLUTION 2
  + This constant defines how many digits will be after the decimal point when saving the dark counts measurement to an SD card.
* #define TURBIDITY\_HELPER\_DARK\_VAR\_NUM 2
  + This constant defines which variable is in position 2 of the variable array. In this case, the dark counts variable (from sensor #2) is in position 2.
* #define DATA\_LEN 6
  + This constant defines the size of the response from the turbidity sensor. The turbidity sensor sends 6 float-type variables, causing the data length constant to be 6.
* #define RES\_DARK1 0
  + The turbidity sensor returns an array of 6 float-type variables to the datalogger. The float at position 0 of this array is the number of dark counts from sensor #1. The units for this are Hertz.
* #define RES\_TRAN1 2
  + The turbidity sensor returns an array of 6 float-type variables to the datalogger. The float at position 2 of this array is the number of transmission counts from LED #1. The units for this are Hertz.
* #define RES\_SCAT1 3
  + The turbidity sensor returns an array of 6 float-type variables to the datalogger. The float at position 3 of this array is the number of side scatter counts from LED #1. The units for this are Hertz.
* #define RES\_DARK2 1
  + The turbidity sensor returns an array of 6 float-type variables to the datalogger. The float at position 1 of this array is the number of dark counts from sensor #2. The units for this are Hertz.
* #define RES\_TRAN2 4
  + The turbidity sensor returns an array of 6 float-type variables to the datalogger. The float at position 4 of this array is the number of transmission counts from LED #2. The units for this are Hertz.
* #define RES\_SCAT2 5
  + The turbidity sensor returns an array of 6 float-type variables to the datalogger. The float at position 5 of this array is the number of side scatter counts from LED #2. The units for this are Hertz.

**Class Members**

* uint8\_t \_i2cAddressHex;
  + This variable contains the I2C address of the turbidity slave sensor.
* float response[DATA\_LEN];
  + This array contains the response from the turbidity sensor. This array consists of 6 float-type variables. Using these 6 variables, the turbidity of the water can be calculated. The variable at each position of the array is given by the “RES\_XXX” constants. (Note that RES stands for RESponse array).

**Class Methods**

* Turbidity(int powerPin, uint8\_t i2cAddressHex = 0x08, int

measurementsToAverage = 1);

* + This is the constructor for the “Turbidity.h” class. In this function, the “i2cAddressHex” variable is saved to the “\_i2cAddressHex” member variable.
* bool wake(void) override;
  + This function is executed whenever the sensor is “waked” after a period of being idle. This function basically makes sure the sensor is powered on, and that the sensor has waited the proper warm up time.
* SENSOR\_STATUS setup(void) override;
  + This function is executed whenever the sensor is set up for the first time. This function doesn’t do anything, and only returns the value of “getStatus()”.
* SENSOR\_STATUS getStatus(void) override;
  + This function makes sure that the sensor is powered on. Then, this function checks to make sure that there is an I2C slave device at the “\_i2cAddressHex” address. If there are any errors in either of these processes, the “SENSOR\_ERROR” flag is returned. If the sensor was set up successfully, the “SENSOR\_READY” flag is returned.
* String getSensorLocation(void) override;
  + This function returns a string containing the I2C address of the sensor. For example, “I2C\_0x08”.
* bool addSingleMeasurementResult(void) override;
  + This function begins by calling the “sensorRead()” function. This function fills the “response[]” array with the current measurements from the turbidity sensor. Next, the values from the response array are copied to the “sensorValues[]” array. This allows the readings to be logged later. Finally, the function returns true upon completion.

* bool sensorPresent();
  + This function verifies that a sensor is present at the current I2C address specified by the variable “\_i2cAddressHex”. If a sensor is not present, this function returns false. If a sensor is present, the function returns true.
* void sensorRead();
  + This function reads data from the slave turbidity sensor over I2C. This is done in a byte by byte process, and the results are stored in the float-type “response[]” array.