```
/*
 * This is the .h file for the ccs821 VOC sensor
 */
#ifndef LIB_ADAFRUIT_CCS811_H
#define LIB_ADAFRUIT_CCS811_H

#if (ARDUINO >= 100)
 #include "Arduino.h"
#else
 #include "WProgram.h"
#endif

#include <Wire.h>

/*=========================================================================
    I2C ADDRESS/BITS
    -----------------------------------------------------------------------*/
    #define CCS811_ADDRESS                (0x5A)
/*=========================================================================*/

    #define MAX_READ_COUNT 5

/*=========================================================================
    REGISTERS
    -----------------------------------------------------------------------*/
    enum
    {
        CCS811_STATUS = 0x00,
        CCS811_MEAS_MODE = 0x01,
        CCS811_ALG_RESULT_DATA = 0x02,
        CCS811_RAW_DATA = 0x03,
        CCS811_ENV_DATA = 0x05,
        CCS811_NTC = 0x06,
        CCS811_THRESHOLDS = 0x10,
        CCS811_BASELINE = 0x11,
        CCS811_HW_ID = 0x20,
        CCS811_HW_VERSION = 0x21,
        CCS811_FW_BOOT_VERSION = 0x23,
        CCS811_FW_APP_VERSION = 0x24,
        CCS811_ERROR_ID = 0xE0,
        CCS811_SW_RESET = 0xFF,
    };

  //bootloader registers
  enum
  {
    CCS811_BOOTLOADER_APP_ERASE = 0xF1,
    CCS811_BOOTLOADER_APP_DATA = 0xF2,
    CCS811_BOOTLOADER_APP_VERIFY = 0xF3,
    CCS811_BOOTLOADER_APP_START = 0xF4
  };

  enum
  {
    CCS811_DRIVE_MODE_IDLE = 0x00,
    CCS811_DRIVE_MODE_1SEC = 0x01,
    CCS811_DRIVE_MODE_10SEC = 0x02,
    CCS811_DRIVE_MODE_60SEC = 0x03,
    CCS811_DRIVE_MODE_250MS = 0x04,
  };

/*=========================================================================*/

#define CCS811_HW_ID_CODE     0x81

#define CCS811_REF_RESISTOR     100000

/**************************************************************************/
/*!
```

```
    @brief  Class that stores state and functions for interacting with CCS811 gas sensor chips
*/
/**************************************************************************/
class Adafruit_CCS811 {
  public:
    //constructors
    Adafruit_CCS811(void) {};
    ~Adafruit_CCS811(void) {};

    bool start_voc(void);
    void run_voc(void);
    float get_eCO2_ave(void);
        float get_TVOC_ave(void);

    bool begin(uint8_t addr = CCS811_ADDRESS);

    void setEnvironmentalData(uint8_t humidity, double temperature);

    //calculate temperature based on the NTC register
    double calculateTemperature();

    void setThresholds(uint16_t low_med, uint16_t med_high, uint8_t hysteresis = 50);

    void SWReset();

    void setDriveMode(uint8_t mode);
    void enableInterrupt();
    void disableInterrupt();

        /**************************************************************************/
        /*!
            @brief  returns the stored total volatile organic compounds measurement. This does does not read the sensor. To do so, call readData()
            @returns TVOC measurement as 16 bit integer
        */
        /**************************************************************************/
    uint16_t getTVOC() { return _TVOC; }

        /**************************************************************************/
        /*!
            @brief  returns the stored estimated carbon dioxide measurement. This does does not read the sensor. To do so, call readData()
            @returns eCO2 measurement as 16 bit integer
        */
        /**************************************************************************/
    uint16_t geteCO2() { return _eCO2; }

        /**************************************************************************/
        /*!
            @brief  set the temperature compensation offset for the device. This is needed to offset errors in NTC measurements.
            @param offset the offset to be added to temperature measurements.
        */
        /**************************************************************************/
    void setTempOffset(float offset) { _tempOffset = offset; }

    //check if data is available to be read
    bool available();
    uint8_t readData();

    bool checkError();

  private:
    float eCO2_buf[MAX_READ_COUNT];
        float TVOC_buf[MAX_READ_COUNT];
        float eCO2_ave;
        float TVOC_ave;
        void read_voc(void);
        void fill_buffer(void);
        void print_readings(void);
        void calculate_average_reading(void);
        void print_average_reading(void);
        int read_count;
        bool is_average_taken;

    uint8_t _i2caddr;
    float _tempOffset;

    uint16_t _TVOC;
    uint16_t _eCO2;

    void      write8(byte reg, byte value);
    void      write16(byte reg, uint16_t value);
        uint8_t   read8(byte reg);

    void read(uint8_t reg, uint8_t *buf, uint8_t num);
    void write(uint8_t reg, uint8_t *buf, uint8_t num);
    void _i2c_init();

/*=========================================================================
  REGISTER BITFIELDS
    -----------------------------------------------------------------------*/
    // The status register
        struct status {

            /* 0: no error
            *  1: error has occurred
            */
            uint8_t ERROR: 1;

            // reserved : 2

            /* 0: no samples are ready
            *  1: samples are ready
            */
            uint8_t DATA_READY: 1;
            uint8_t APP_VALID: 1;

      // reserved : 2

            /* 0: boot mode, new firmware can be loaded
            *  1: application mode, can take measurements
            */
            uint8_t FW_MODE: 1;

            void set(uint8_t data){
              ERROR = data & 0x01;
              DATA_READY = (data >> 3) & 0x01;
              APP_VALID = (data >> 4) & 0x01;
              FW_MODE = (data >> 7) & 0x01;
            }
        };
        status _status;

        //measurement and conditions register
        struct meas_mode {
          // reserved : 2

          /* 0: interrupt mode operates normally
            *  1: Interrupt mode (if enabled) only asserts the nINT signal (driven low)
            if the new
        ALG_RESULT_DATA crosses one of the thresholds set in the THRESHOLDS register
        by more than the hysteresis value (also in the THRESHOLDS register)
            */
          uint8_t INT_THRESH: 1;

          /* 0: int disabled
```

```
            *  1: The nINT signal is asserted (driven low) when a new sample is ready in
        ALG_RESULT_DATA. The nINT signal will stop being driven low when
        ALG_RESULT_DATA is read on the I²C interface.
            */
          uint8_t INT_DATARDY: 1;

          uint8_t DRIVE_MODE: 3;

          uint8_t get(){
            return (INT_THRESH << 2) | (INT_DATARDY << 3) | (DRIVE_MODE << 4);
          }
        };
        meas_mode _meas_mode;

        struct error_id {
          /* The CCS811 received an I²C write request addressed to this station but with
      invalid register address ID */
          uint8_t WRITE_REG_INVALID: 1;

          /* The CCS811 received an I²C read request to a mailbox ID that is invalid */
          uint8_t READ_REG_INVALID: 1;

          /* The CCS811 received an I²C request to write an unsupported mode to
      MEAS_MODE */
          uint8_t MEASMODE_INVALID: 1;

          /* The sensor resistance measurement has reached or exceeded the maximum
      range */
          uint8_t MAX_RESISTANCE: 1;

          /* The Heater current in the CCS811 is not in range */
          uint8_t HEATER_FAULT: 1;

          /*  The Heater voltage is not being applied correctly */
          uint8_t HEATER_SUPPLY: 1;

          void set(uint8_t data){
            WRITE_REG_INVALID = data & 0x01;
            READ_REG_INVALID = (data & 0x02) >> 1;
            MEASMODE_INVALID = (data & 0x04) >> 2;
            MAX_RESISTANCE = (data & 0x08) >> 3;
            HEATER_FAULT = (data & 0x10) >> 4;
            HEATER_SUPPLY = (data & 0x20) >> 5;
          }
        };
        error_id _error_id;

/*=========================================================================*/
};

#endif
```