Application Note

# CCS811

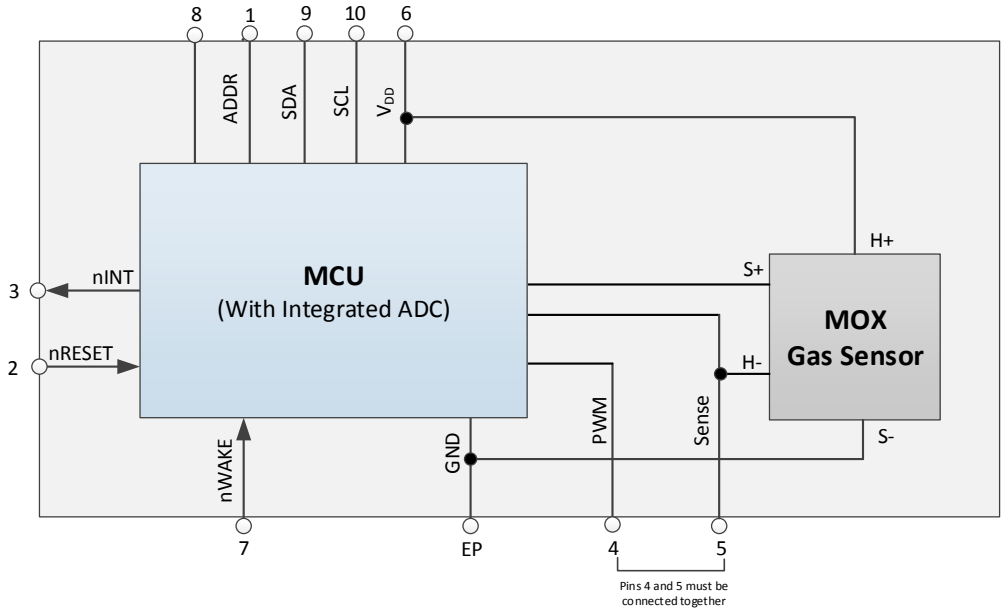
## Programming and Interfacing Guide

## Content Guide

# 1 Introduction

This application note details the programming model and recommended interfacing techniques for ams' CCS811 digital gas sensor. It is intended for software developers who will integrate the CCS811 device into an environment with another core (such as a sensor hub, sensor aggregator, phone, tablet or digital display product). It details how user software can interface to, control, enable and disable the sensor.

**Figure 1: CCS811 Block Diagram**




The CCS811 block diagram is shown in Figure 1. The host interfaces to the CCS811 using the I²C bus. All signals prepended with 'n' are active low and optional, except nWAKE. If not controlled by GPIO the nRESET signal must be pulled high, it is strongly recommended that the nWAKE input signal is controlled by a software configurable GPIO pin in power sensitive applications. In applications where power is not a major concern it is possible to tie nWAKE to ground.

Configuration and control of the CCS811 is accomplished by the host issuing transactions on the I²C interface to specific addresses (mailboxes) on the CCS811. The sensor can operate in polling mode, or can issue an interrupt on nINT when it has generated an $eCO_2$ reading, and/or when an $eCO_2$ threshold is reached.

## 2 Programming Model

For flexibility and software driver maintenance simplification the CCS811 does not support direct addressing of registers over the I²C bus. Instead it supports single byte mailboxes that behave as proxies for registers with specific functionality and data sizes.

To access a register an I²C transaction must be issued with the target address on the CCS811 being equal to the intended mailbox.

All I²C transactions must use the 7-bit address 0x5A (I2C _ADDR low) or 0x5B (I2C _ADDR high) when writing to and reading from the CCS811.

**Figure 2: Application Mode Mailbox Addresses and Sizes**

| Mailbox ID | Mailbox Size (Bytes) | Mailbox Name | Direction |
|---|---|---|---|
| 0x00 | 1 | STATUS | Read |
| 0x01 | 1 | MEAS_MODE | Read / Write |
| 0x02 | 8 | ALG_RESULT_DATA | Read |
| 0x03 | 2 | RAW_DATA | Read |
| 0x05 | 4 | ENV_DATA | Write |
| 0x10 | 5 | THRESHOLDS | Write |
| 0x11 | 2 | BASELINE | Read / Write |
| 0x20 | 1 | HW_ID | Read |
| 0x21 | 1 | HW_VERSION | Read |
| 0x23 | 2 | FW_Boot_Version | Read |
| 0x24 | 2 | FW_App_Version | Read |
| 0xE0 | 1 | ERROR_ID | Read |
| 0xFF | 4 | SW_RESET | Write |

For a detailed description a mailbox's functionality please refer to the CCS811 data sheet. Figure 2 details the mailboxes when the CCS811 is the application mode. The majority of the examples that follow will refer to the mailboxes visible in application mode. Readers should note that in boot mode the register map differs. The functionality in each mode is as follows:

- Boot mode allows the user to program or change the application firmware version. In boot mode the user software must write to mailbox 0xF4 to transition to application mode
- Application mode allows the user to configure and start the sensor and has features for runtime control such as environmental data compensation

In this document mailbox and register are used interchangeably when referring to an CCS811 mailbox.

# 3 ERROR_ID Register

Figure 3 shows the bit fields contained in the ERROR_ID register.

**Figure 3: ERROR_ID Register**

| 7:6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|
| - | HEATER SUPPLY | HEATER FAULT | MAX RESISTANCE | MEASMODE INVALID | READ_REG INVALID | MSG INVALID |

Figure 4 describes the causes and recommended actions for each error bit in ERROR_ID. This register is valid when the STATUS [ERROR] bit is equal to 1.

**Figure 4: ERROR_ID Fault Descriptions**

| Mailbox ID | Mailbox Size (Bytes) | Mailbox Name |
|---|---|---|
| MSG_INVALID | The CCS811 received an I²C write request addressed to this station but with invalid mailbox ID or the wrong size | Check that host is sending the correct sequence or for noise in physical interface. |
| READ_REG_INVALID | The CCS811 received an I²C read request to a mailbox ID that is invalid | Check that host is sending the correct sequence or for noise in physical interface |
| MEASMODE_INVALID | The CCS811 received an I²C request to write an unsupported mode to MEAS_MODE | Check that the correct mode appears on the I²C bus when writing to MEAS_MODE |
| MAX_RESISTANCE | Damage, exposure to contaminants or heater not operating | Check for heater fault flags. Ensure sensor is in a typical atmosphere and note any environments that the sensor may have previously seen |
| HEATER_FAULT | Soldering, PCB issue or damage | Check soldering. On an unpowered board there should be approximately 38 ohms between pin 5 and 6. |
| HEATER_SUPPLY | Soldering or PCB issue | Check soldering. On an unpowered board there should be approximately 38 ohms between pin 5 and 6. |

The ERROR_ID register will be cleared by the following events:

- The application software performs a read of the ERROR register on the I²C interface
- The application software performs the SW_RESET sequence by writing the appropriate code to the SW_RESET mailbox

- A power on reset is issued
- The nRESET signal is asserted

It is possible that more than 1 error flag is set in this register, thus application software should check each bit individually every time this register is read.

# 4 Examples

The following examples will use the pseudo code below to help illustrate the functionality and programming flow required for the CCS811.

```
#define STATUS_REG 0x00
#define MEAS_MODE_REG 0x01
#define ALG_RESULT_DATA 0x02
#define ENV_DATA 0x05
#define THRESHOLDS 0x10
#define BASELINE 0x11
#define HW_ID_REG 0x20
#define ERROR_ID_REG 0xE0
#define APP_START_REG 0xF4
#define SW_RESET 0xFF

#define CCS_811_ADDRESS 0x5A        // CCS811 ADDR pin is logic zero otherwise address would be 0x5B
#define GPIO_WAKE 0x5
#define DRIVE_MODE_IDLE 0x0
#define DRIVE_MODE_1SEC 0x10
#define DRIVE_MODE_10SEC 0x20
#define DRIVE_MODE_60SEC 0x30
#define INTERRUPT_DRIVEN 0x8
#define THRESHOLDS_ENABLED 0x4

u8 i2c_buff[8];
bool wake_gpio_enabled = true;
void i2c_write(u8 address, u8 register, u8 *tx_data_ptr, u8 length);
void i2c_read(u8 address, u8 *rx_data_ptr, u8 length);
void gpio_write(u8 gpio_id, u8 level);
```

The pseudo code utilizes a global array, i2c_buff, to hold the data to be transmitted and the data received. For illustrative purposes it is assumed that read transactions do not automatically perform a set-up write before the read. Some environments abstract this in their API. The I²C function prototypes are fairly self-explanatory:

- u8 address: this argument is the value of the CCS811 address (CCS_811_ADDRESS)
- u8 register: this argument is the mailbox ID
- u8 length: this argument is the number of bytes to write/read not including address and register
- u8 rx_data_ptr/tx_data_ptr: pointer to a buffer where the called function will access the data to be written to the CCS811 or store the data read from the CCS811, i2c_buff in most of the following examples

The gpio_write function is used by the host to write GPIO outputs (input to CCS811) to a logic high or logic low level. Its arguments are described below:

- u8 gpio_id: GPIO number to be written to logic high or low
- u8 level: zero for logic low, and one for logic high

See 13 Handling nWAKE Using a GPIO for more information.

For simplicity, the examples do not show any error handling of the physical layer such as $I^2C$ aborts and timeouts.

# 5 CCS811 I²C Data Byte Ordering

When reading or writing more than one byte the user must be aware of the order of the bytes used by the CCS811. The CCS811 assumes that the most significant byte of the scalar is present on the I²C before bytes of lesser significance. For example reading a 16-bit scalar with the value 0x11AA, the byte with value 0x11 would appear on the bus before the byte with value 0xAA.

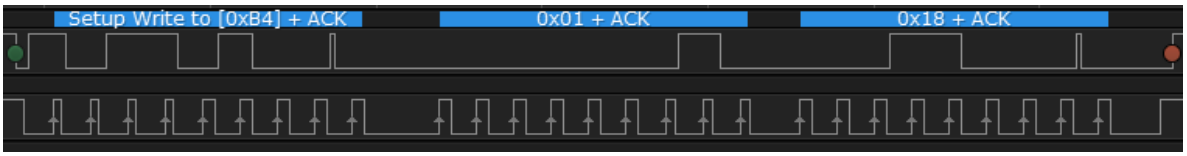
# 6 I²C Write Transactions

The host must perform I²C write transactions to enable and configure the sensor in the intended operation mode. When writing to a mailbox, for example the MEAS_MODE register with a 1s drive mode and interrupts enabled, the user software would run code similar to the following code example:

```
i2c_buff[0] = DRIVE_MODE_1SEC | INTERRUPT_DRIVEN;
i2c_write(CCS_811_ADDRESS, MEAS_MODE_REG, i2c_buff, 1);
```

If this transaction is viewed on a protocol analyzer it will look something similar to Figure 5

**Figure 5: I²C Write Transaction to MEAS_MODE, 1 Second Drive Mode, Interrupts Enabled**




Note that the I²C address occupies the most significant 7 bits of the first byte transmitted. The least significant bit indicates write or read with a logic 0 or a logic 1 value respectively. This is a write hence why the value of this byte is 0xB4 (i.e. 0x5A << 1 = 0xB4). The next byte on the line is the mailbox ID for MEAS_MODE, and finally the data, 0x18 (DRIVE_MODE = 1s, Interrupt enable), that is written to the MEAS_MODE mailbox's register.

# 7 I²C Read Transactions

I²C read transactions must be preceded with a set-up write to enable the target mailbox for the read. This is best illustrated by an example. Let's assume that the user is required to read the STATUS register, the code flow would look like the following:

```
i2c_write(CCS_811_ADDRESS, STATUS_REG, i2c_buff, 0);
i2c_read(CCS_811_ADDRESS, i2c_buff, 1);
```

Viewing this on a protocol analyzer is illustrated in Figure 6.

**Figure 6: I²C Read Transaction To STATUS Register**

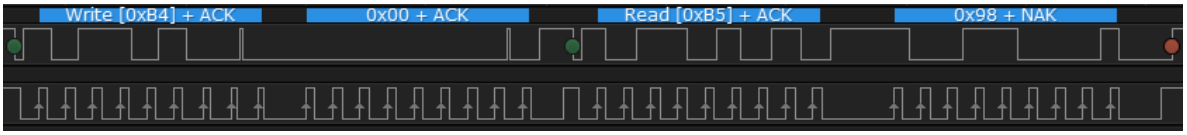



Figure 6 shows a setup write to mailbox 0x00. After the write to read transactions target the status register. Note the size of the write data in this instance is zero bytes, and as such the i2c_buff array does not need to be written with any data. The read command contains the size of the data in this mailbox's register, in this case 1 byte. The i2c_read function will store to i2c_buff[0] the value of the status register i.e. 0x98. User software can then react accordingly to the values of the fields in the STATUS register.

# 8 Hardware ID (0x20) Register Handling

The hardware ID register, sometimes referred to as the 'Who Am I' register, can be read during CCS811 initialization to ensure that the device is indeed a CCS811. When this mailbox is the target of a read its proxy register will return the value 0x81. To read this register the host software would perform the following:

```
i2c_write(CCS_811_ADDRESS, HW_ID_REG, i2c_buff, 0);
i2c_read(CCS_811_ADDRESS, i2c_buff, 1);
if(i2c_buff[0] != 0x81)
return ERROR_NOT_A_CCS811;
```

**Figure 7 I2C Read Transaction to HW_ID**


The setup write to the mailbox HW_ID precedes the actual read of its proxy register. The read is 1 byte in length and it must return 0x81 to the i2c_buff array, otherwise the device will not be configured by the host's software. The resultant transactions on $I^2C$ would appear as shown in the protocol analyzer screenshot in Figure 7.

# 9 APP_START (0xF4) Handling

The mailbox ID 0xF4, APP_START, is used to transition the CCS811 state from boot to application mode. Note that the size of the proxy register for this mailbox is zero bytes. Therefore to trigger this state transition a setup write with is all that is required as shown in Figure 8.

**Figure 8: APP_START Interface**


Note the byte 0xF4 terminates the I²C transaction, no data follows this byte. The CCS811 treats this as a command to transition from boot to application mode. If any data appears after the 0xF4 byte then the CCS811 will not enter application mode.

A typical flow can be viewed below:

```
i2c_write(CCS_811_ADDRESS, STATUS_REG, i2c_buff, 0);
i2c_read(CCS_811_ADDRESS, i2c_buff, 1);
if(!(i2c_buff[0] & 0x10))
return ERROR_NO_VALID_APP;
i2c_write(CCS_811_ADDRESS, APP_START_REG, i2c_buff, 0);
```

Note that before mailbox 0xF4 is written to the code should check that there is a valid application, STATUS[APP_VALID] = 1. If this is not set then an error condition is returned and valid application firmware should be downloaded to the CCS811. If there is valid firmware present then the setup write to the APP_START mailbox will transition the CCS811 state from boot to application mode. The logic analyzer capture illustrates the state of the sensor before the application has been started:

**Figure 9: STATUS Register Prior To APP_START Setup Write**

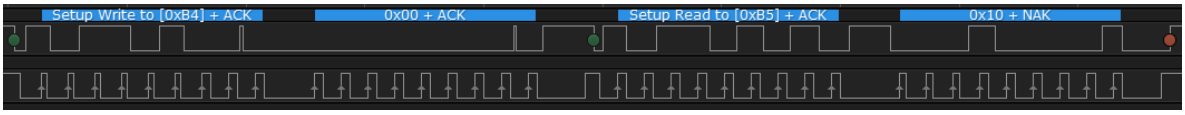



Figure 9 shows the status register has returned the value 0x10, which indicates that a valid app is present but the sensor is still in boot mode. The next transaction on the I²C bus is a setup write to mailbox 0xF4. Subsequent reads of the status register then return the value 0x90, i.e. there is a valid application and the sensor is in application mode. The DATA_READY flag will remain zero until a valid drive mode is written to the MEAS_MODE register.

## 10 Host Software Polling Mode

In this mode the host software is required to cyclically read data from the sensor at the same period as the programmed drive mode. A timer interrupt on the host application can be used to read the sensor's $eCO_2$ and TVOC values. A typical flow for this can be seen below:

```
u8 *timer_routine_read_data(){
i2c_write(CCS_811_ADDRESS, STATUS_REG, i2c_buff, 0);
i2c_read(CCS_811_ADDRESS, i2c_buff, 1);
if(i2c_buff & 0x8)                                          // check if data ready
{
i2c_write(CCS_811_ADDRESS, ALG_RESULT_DATA, i2c_buff, 0);
i2c_read(CCS_811_ADDRESS, i2c_buff, 4);
}
return i2c_buff;
}
```

The status register is read to see if data is ready. If data is ready the $eCO_2$ and TVOC results are read by performing a 4 byte data read to ALG_RESULT_DATA. The software can then process the data according to the $eCO_2$ and TVOC results in the i2c_buff location. The protocol analyzer shows the following:

**Figure 10: Polling Mode for CCS811 Data Samples**


Basically a set-up write and a subsequent read of 1byte to mailbox 0x0 to check the status for data ready. This is followed by a setup write to mailbox 0x2 and a 4 byte read. The data would be stored in i2c_buff as follows:

- i2c_buff[0] = 0x01
- i2c_buff[1] = 0x90
- i2c_buff[2] = 0x00
- i2c_buff[3] = 0x32

Note that i2c_buff[0] and i2c_buff[1] values should be converted to a 16 bit type field with the value 0x0190 (decimal 400), this is the current $eCO_2$ reading from the sensor. Similarly the bytes values in i2c_buff[2] and i2c_buff[3] should be converted to a 16 bit value 0x0032 that is the TVOC, in this case decimal 50.

This example above assumes 4 bytes of data is read from ALG_RESULT_DATA. It is possible to read up to eight bytes from ALG_RESULT_DATA. The format of this is shown in Figure 11.

**Figure 11: ALG_RESULT_DATA Format**

| Byte: | 0:1 | 2:3 | 4 | 5 | 6:7 |
|---|---|---|---|---|---|
| Parameter: | $eCO_2$ | TVOC | STATUS | ERROR_ID | RAW_DATA |

The user can define a structure to hold this data as follows:

```
typedef struct {
u16 eco2;
u16 tvoc;
u8 status;
u8 error_id;
u16 raw_data;
} ccs811_measurement_t;
```

The base address of this structure can be cast to a char pointer and supplied as the address that the i2c_read routine deposits the data read from the sensor. Alternatively data can be copied to this structure from the current i2c_buff. Care must always be taken with processor endianness when accessing data written to the i2c_buff by the i2c_read function.

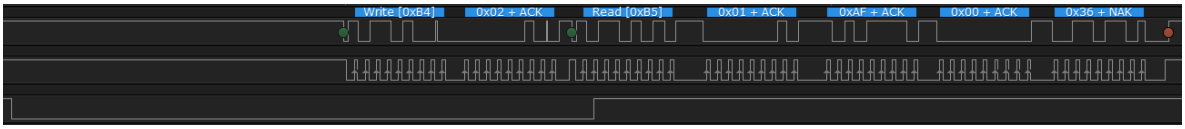
## 11 Reading Data On Interrupt Thresholds Disabled

The nINT signal is driven low by the CCS811 whenever new data is ready and the MEAS_MODE[INTERRUPT] bit is set. The host's CCS811 driver can provide a callback, to the host application's O/S, which is registered to the GPIO signal connected to the nINT output signal. When the nINT is asserted the callback is invoked which then reads the ALG_RESULT_DATA from the CCS811:

```
u8 *ccs811_read_data_callback(){
i2c_write(CCS_811_ADDRESS, ALG_RESULT_DATA, i2c_buff, 0);
i2c_read(CCS_811_ADDRESS, i2c_buff, 4);
return i2c_buff;
}
```

There is no need to check data ready in this case. The issuing of the interrupt indicates that new data samples are ready. The protocol analyzer would show a similar $I^2C$ transaction flow to Figure 12.

**Figure 12: Interrupt Triggering a Read of CCS811 Data Samples**




The nINT line is asserted by the CCS811, a short time thereafter the host application calls the callback which performs a setup write to mailbox 0x2, ALG_RESULT_DATA. Next the callback routine performs a 4 byte read, which results in the $eCO_2$ and TVOC readings to be stored into the i2c_buff location.

## 12 Reading Data On Threshold Interrupt

It is possible to program CCS811 such that the nINT signal generates an interrupt when an $eCO_2$ reading crosses the low or high threshold in either direction. The firmware will only trigger the interrupt if the threshold is breached by more than a programmable hysteresis value.

```
typedef struct __packed{
u16 thresh_low;
u16 thresh_high;
u8 thresh_hyst;
}threshold_reg;
…
threshold_reg thresh_reg = {1000, 2200, 50};
temp_ptr = (uint8_t *)&thresh_reg->thresh_low;
i2c_write(CCS_811_ADDRESS, THRESHOLDS, temp_ptr, 5);
…
i2c_buff[0] = DRIVE_MODE_1SEC | INTERRUPT_DRIVEN | THRESHOLDS_ENABLED;
i2c_write(CCS_811_ADDRESS, MEAS_MODE_REG, i2c_buff, 1);
```

Before enabling the drive mode the user must program the desired low and high threshold value in the THRESHOLDS' registers. This can be achieved by defining and declaring the structure threshold_reg shown above. It can be used to hold the values of the threshold registers. The structure's base can be cast to a pointer to char, it then can be the tx_data_ptr argument in the i2c_write function call.

The default values in the CCS811 threshold registers are as follows, if the application always uses these values there is no requirement to write to the threshold registers:

- LOW_THRESHOLD = 1500ppm
- HIGH_THRESHOLD = 2500ppm
- HYSTERESIS = 50ppm

If only one threshold is required the low and high threshold can be configured to the same value. It is possible to change the threshold registers on the fly.

## 13 Handling nWAKE Using a GPIO

There are 2 methods for handling the nWAKE:

1. Pull low and therefore always asserted
2. Use a GPIO and control assertion and deassertion dynamically

When the active low nWAKE signal is asserted the CCS811's integrated processor is active and will handle requests on the $I^2C$ interface. When this pin is logic high the CCS811 enters sleep mode and all $I^2C$ requests are ignored. Therefore when this signal is dynamically controlled by a GPIO it can efficiently control CCS811 power consumption. Pulling nWAKE permanently low is not recommended in power sensitive applications.

```
if(wake_gpio_enabled)
gpio_write(GPIO_WAKE, 0);                                   // enable wake
i2c_write(CCS_811_ADDRESS, STATUS_REG, i2c_buff, 0);
i2c_read(CCS_811_ADDRESS, i2c_buff, 1);
if(i2c_buff & 0x8)                                          // check if data ready
{
i2c_write(CCS_811_ADDRESS, ALG_RESULT_DATA, i2c_buff, 0);
i2c_read(CCS_811_ADDRESS, i2c_buff, 4);
}
if(wake_gpio_enabled)
gpio_write(GPIO_WAKE, 1);                                   // disable wake
```

nWAKE is enabled by writing logic level zero to the GPIO, note the code checks if this signal is currently controlled by a GPIO. If no GPIO is available then nWAKE must be pulled low. The status register is then read to see if data is ready.

When data is ready the $eCO_2$ and TVOC results are read by performing a read to ALG_RESULT_DATA. Finally nWAKE is disabled. The software can then process the data according to the $eCO_2$ and TVOC results in the i2c_buff location.

## 14 Disabling the CCS811

In order to disable the sensor and place in the lowest possible power consumption mode (while still connected to the supply voltage $V_{DD}$), the following can used:

```
if(wake_gpio_enabled)
gpio_write(GPIO_WAKE, 0);                                   // enable wake
i2c_buff[0] = DRIVE_MODE_IDLE;
i2c_write(CCS_811_ADDRESS, MEAS_MODE_REG, i2c_buff, 1);
if(wake_gpio_enabled)
gpio_write(GPIO_WAKE, 1);                                   // disable wake
```

The driver can disable the sensor by programming the drive mode in MEAS_MODE to the idle state. The nWAKE signal can then be placed in its inactive state (logic high), thus no $I^2C$ commands will be processed. To ‘re-awaken’ the sensor nWAKE should be asserted and the appropriate drive mode including interrupt, if used, should be written to the MEAS_MODE register.

Another method of disabling the sensor is by writing the unlock sequence to the SW_RESET mailbox. This sequence is shown in the code example below (nWAKE handling not shown for simplicity):

```
u8 soft_reset_code[] = {0x11, 0xE5, 0x72, 0x8A};
i2c_write(CCS_811_ADDRESS, SW_RESET, soft_reset_code, 4);
```

This will reset the CCS811 and place it in boot mode, ready to be reprogrammed.

## 15 CCS811 Timing Considerations

The CCS811 data sheet details a number of timing parameters that the programmer must adhere to. Failure to meet these timings may result in initialization failure. The CCS811 may also return NAK on I²C transactions.

A typical flow in a system is to perform an I²C setup write followed by very quickly thereafter an I²C read. The code for this will similar to the following:

```
i2c_write(CCS_811_ADDRESS, STATUS_REG, i2c_buff, 0);
i2c_read(CCS_811_ADDRESS, i2c_buff, 1);

void i2c_write(ARGS){
gpio_write(GPIO_WAKE, 0);                    // enable wake
...
I2C_WRITE_HW_REG();
...
gpio_write(GPIO_WAKE, 1);                    // disable wake
}

void i2c_read(ARGS){

gpio_write(GPIO_WAKE, 0);                    // enable wake
...
I2C_READ_HW_REG();
...
gpio_write(GPIO_WAKE, 1);                    // disable wake
}
```

The above shows a setup write to the STATUS register followed by a single byte read. Note that the I²C read and write functions each enable and disable nWAKE. This is done as close as possible to the physical I²C transaction in order achieve the least power consumption. However the timing in the data sheet needs to be adhered to.

For example TAWAKE is defined as 50µs, to ensure this is met then after the enabling nWAKE a delay of at least 50µs must be provided by the software. Let's assume we have a routine, wait(DELAY_US), capable of providing delays in increments of microseconds:

```
void i2c_XXXXX(ARGS){                        // XXXXX = read or write
gpio_write(GPIO_WAKE, 0);                    // enable wake
wait(50);                                    // ensure TAWAKE
...
I2C_XXXXX_HW_REG();
...
gpio_write(GPIO_WAKE, 1);                    // disable wake
```

TDWAKE is 20µs, this is the minimum deassertion time of the nWAKE signal (if it is deasserted) between back to back I²C transactions to the CCS811. This timing could be violated if the i2c_write routine desserts nWAKE and subsequently the i2c_read routine asserts wake too quickly.

To ensure this timing parameter is not violated a delay is required after de-asserting nWAKE as follows:

```
void i2c_XXXXX(ARGS){                    // XXXXX = read or write
wait(20);                                // ensure TDWAKE
gpio_write(GPIO_WAKE, 0);                // enable wake
wait(50); // ensure TAWAKE
...
I2C_XXXXX_HW_REG();
...
gpio_write(GPIO_WAKE, 1);                // disable wake
```

The delay can either be placed before the nWAKE assertion or directly after the deassertion on nWAKE in the I²C routines. In a similar manner after pulsing nRESET or writing to the SW_RESET or after a power on, the timing parameters $T_{START}$ and $T_{RESET}$ must not be violated by the driver.

# 16 Handling Environment Parameters in ENV_DATA (0x05)

The CCS811 supports compensation for relative humidity and ambient temperature. The ENV_DATA registers can be updated with the temperature and humidity (TH) values on each cycle.

To avoid floating point operations a typical TH sensor will represent a value a few orders of magnitude greater. For example a humidity reading of 42.348% will be stored in a sensor's H result register as 42348. The CCS811 supports a 7 bit field for humidity and a 9 bit fraction followed by a 7 bit temperature field and corresponding 9 bit fraction.

**Figure 13: Environmental Data Register Fields and Byte Order**

| Byte 0 | | | | | | | | Byte 1 | | | | | | | | Byte 2 | | | | | | | | Byte 3 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Humidity High Byte | | | | | | | | Humidity Low Byte | | | | | | | | Temperature High Byte | | | | | | | | Temperature Low Byte | | | | | | | |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Humidity % | | | | | | | | Humidity % Fraction | | | | | | | | Temp + 25ºC | | | | | | | | Temp + 25ºC Fraction | | | | | | | |

The data format of the TH sensor must be mapped to the ENV_DATA format shown in Figure 13. To place humidity value at the start of the I²C buffer, i2c_buff, the user can create a routine similar to the following:

```
i2c_buff[0] = ((RH % 1000) / 100) > 7 ? (RH/1000 + 1)<<1 : (RH/1000)<<1;
i2c_buff[1] = 0;
if(((RH % 1000) / 100) > 2 && (((RH % 1000) / 100) < 8))
{
i2c_buff[0] |= 1;
}
```

This routine accepts one argument: the humidity reading from the TH sensor. In this case it is 3 orders of magnitude greater than the actual reading expressed as a percentage. Currently the CCS811 supports fractional values in increments of 0.5, resultantly only the most significant bit of the fraction field needs to be set.

Firstly the 7 bit humidity value is written to i2c_buff[0]. This is achieved using the ternary operator to check if the remainder (modulo 1000) is 0.8 or 0.9 in which case the RH value is rounded up, otherwise RH/1000 is written to the 7 bit humidity field represented in i2c_buff[0].

The driver then sets the fractional part. It does this by checking if the remainder (most significant digit after the decimal point) yields a value between 0.3 and 0.7 inclusive. If so then the most significant bit of the fraction is set.

In a similar manner a routine for placing the temperature value into i2c_buff can be generated as follows:

```
TEMP += 25000;
i2c_buff[2] = ((TEMP % 1000) / 100) > 7 ? (TEMP/1000 + 1)<<1 : (TEMP/1000)<<1;
i2c_buff[3] = 0;
if(((TEMP % 1000) / 100) > 2 && (((TEMP % 1000) / 100) < 8))
{
i2c_buff[2] |= 1;
}
```

The bolded code takes care of the data sheet requirement that current temp value plus 25 is written to the ENV_DATA's temperature register. To program ENV_DATA The user code would therefore look similar to the following:

```
u32 TEMP, RH;

read_temp_hum_sensor(TEMP, RH);
ccs811_temp_hum_convert(TEMP, RH);
i2c_write(CCS_811_ADDRESS, ENV_DATA, i2c_buff, 4);
```

The ccs811_temp_hum_convert combines the 2 examples shown above to render the values extracted from the RH into the format required by the CCS811's ENV_DATA registers (stored in i2c_buff). After this the user can program the ENV_DATA to the sensor in the normal manner using the i2c_write routine.

If the application supports temperature or humidity compensation, but not both, the data sheet's default value must be written to the register corresponding to the unsupported environment parameter. The user must not write zero to the unsupported temperature or humidity field of the ENV_DATA register.

## 16.1 Compensation Using ENS210

The recommended method for environmental compensation is to use the data read from the ENS210 temperature and relative humidity sensor (T+RH). It has the fastest response/recovery time and is the most accurate T+RH sensor of any device in its class. It will therefore yield the most accurate gas sensor compensation data and therefore the most accurate system/product.

**Figure 14: CCS811 with ENS210 for Compensation**

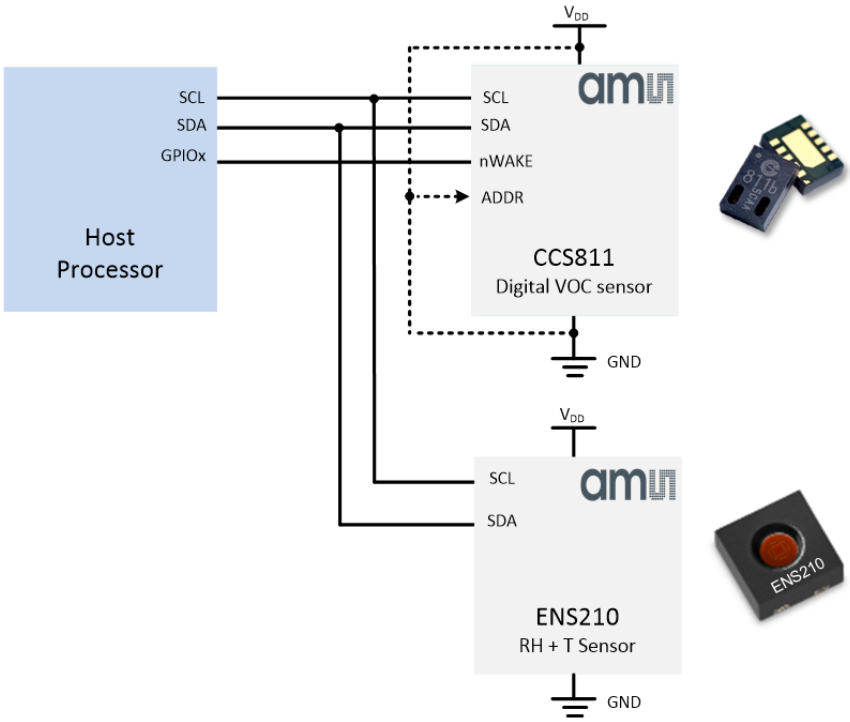



Figure 14 shows a block diagram of the CCS811 and ENS210. The CCS811 is an I²C slave, as is the ENS210.

1. Read T+RH data from the ENS210

Reformat this data to the format required by the CCS811. See

2. Handling Environment Parameters in ENV_DATA (0x05) for details on this format and code for conversion
3. Write the reformatted T+RH data to the ENV_DATA mailbox of the CCS811

The CCS811 will use the most recent T+RH data written to the ENV_DATA mailbox to compensate the $eCO_2$ and TVOC outputs to the current ambient conditions.

## 16.2 Compensation Data Timing Requirements

It is recommended that the T+RH data is read from the ENS210 at the same frequency as the CCS811 generates $eCO_2$ and TVOC outputs. For example in mode 3, 60 second mode, the application software should read the ENS210 data once every minute.

Additionally the flow of reading from the ENS210 and then writing to the CCS811 on the host's software should be uninterrupted. The read of the ENS210 and the write to the CCS811 should be as close in time as physically possible on the end customer system and $I^2C$ bus.

This is required to generate the most accurate compensation and CCS811 outputs. The following flow explains why this is required:

**Figure 15: Compensation Using the Wrong Timing**

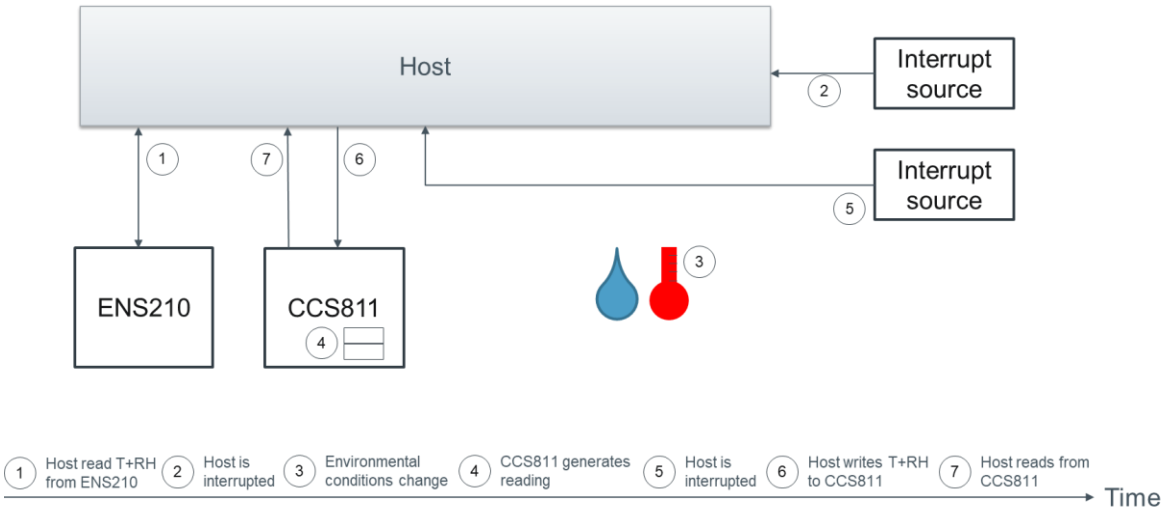



Figure 15 shows an application using the wrong timing. Note that the ENS210 is read too early. Before this data can be written to the CCS811 the host is interrupted from another source and the ambient conditions change. Resultantly at step 4 the CCS811 generates a gas reading using old environmental data that is not representative of the current ambient conditions. The most recent ambient data should be used for compensation.

**Figure 16: Compensation Using Recommended Timing**

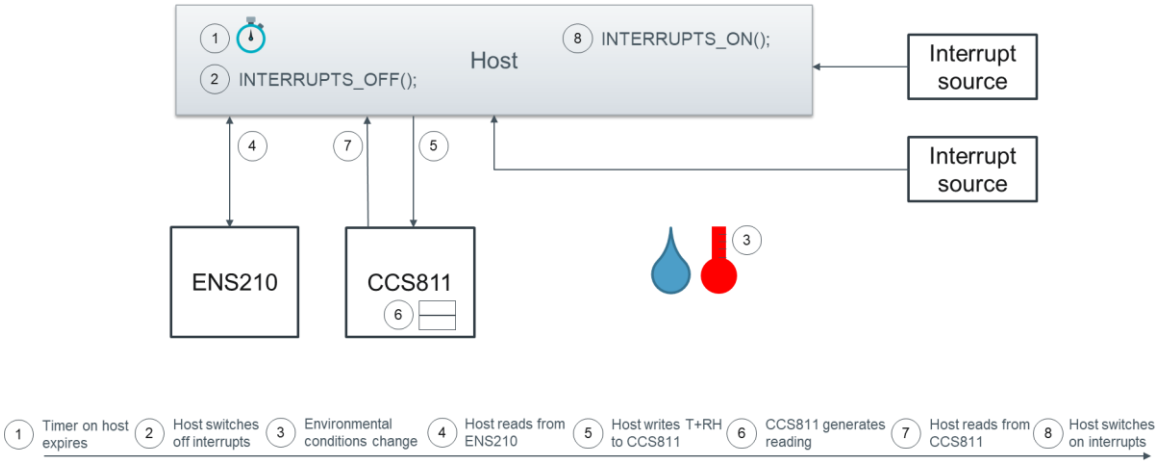



Figure 16 details the recommended timing and handling of environmental data.

1. A timer on the host expires, it has frequency configured for the same duration as the CCS811 generates gas readings
2. Host switches off all interrupts and handles this section in a high priority task
3. Conditions change, this has no impact as the ENS210 has not been accessed
4. Host read T+RH from the ENS210
5. Host immediately writes this to the CCS811 ENV_DATA
6. CCS811 generates a reading soon after
7. Host reads $eCO_2$ and TVOC from CCS811
8. Host switches on interrupts

Note that the interrupt sources are effectively masked until the ENS210 data is written to the CCS811. This is ensures that the section of code that reads from the ENS210 and writes the ENS210 values to the CCS811 takes the minimal amount of time.

# 17 Handling the BASELINE register (0x11)

The BASELINE register can be used to ensure that the CCS811 operates in a stable and consistent manner after each power on, regardless of the ambient air quality (e.g. if the sensor is started in ‘dirty’ air). To use this mechanism the application software must read and store the BASELINE register when the ambient air is fresh.

The BASELINE register value is meaningless to the host application as it uses proprietary encoding. The application software need only read this value and store it locally. It is not mandatory to write to this register, the CCS811 will configure and manage its baseline autonomously. The BASELINE register can be read and its value stored as follows:

```
u8 baseline_reg[2];                                    // storage for baseline value in clean air

if(air_is_clean(eCO2, CO2)){
        i2c_write(CCS_811_ADDRESS, BASELINE, baseline_reg, 0);
        i2c_read(CCS_811_ADDRESS, baseline_reg, 2);
}
```

The application determines if the air is fresh. If so it performs a setup write to BASELINE then a read of 2 bytes into an array storage parameter baseline_reg. The baseline_reg parameter is therefore available to the user to write back to the BASELINE register:

```
i2c_write(CCS_811_ADDRESS, BASELINE, baseline_reg, 2);
```

This writes the 2 bytes stored in baseline_reg to the BASELINE register on the CCS811, setting the CCS811 baseline to its resistance in clean air. It will typically be written after each power on of the CCS811 after the sensor has stabilized. The BASELINE register can only be written when the CCS811 MEAS_MODE[DRIVE_MODE] is not in the idle state.

Due to the long term drift of MOX sensors it is recommended that new clean baselines are stored periodically.

## 18 Host processor Endianness

The endianness of the host processor needs to be known in order to process the $eCO_2$ and TVOC values. Assuming we read the following from ALG_RESULT_DATA i.e. i2c_buff is loaded with:

- i2c_buff[0] = 0x01
- i2c_buff[1] = 0x90
- i2c_buff[2] = 0x00
- i2c_buff[3] = 0x32

Therefore $eCO_2$ = 0x0190 and TVOC = 0x0032.

**Figure 17: Big Endian Storage of $I^2C$ Data**

| Address | 0x1000 | 0x1001 | 0x1002 | 0x1003 |
|---|---|---|---|---|
| Data | 0x01 | 0x90 | 0x00 | 0x32 |

A CPU with a little endian memory storage policy for data will not be able to access these values directly. A conversion/accessor function must be used that handles the endianness conversion from big endian to little endian. Some operating systems, e.g. Linux, support these endianness aware accessor functions and will apply the appropriate endianness formatting based on the size of the scalar being accessed.

If no such functions are available in an environment, the software driver must perform this by either using c code or inline assembly directives that utilize endianness accelerated swap opcodes. For example to ensure that a CPU register is given the correct value of the $eCO_2$ reading the following can be used:

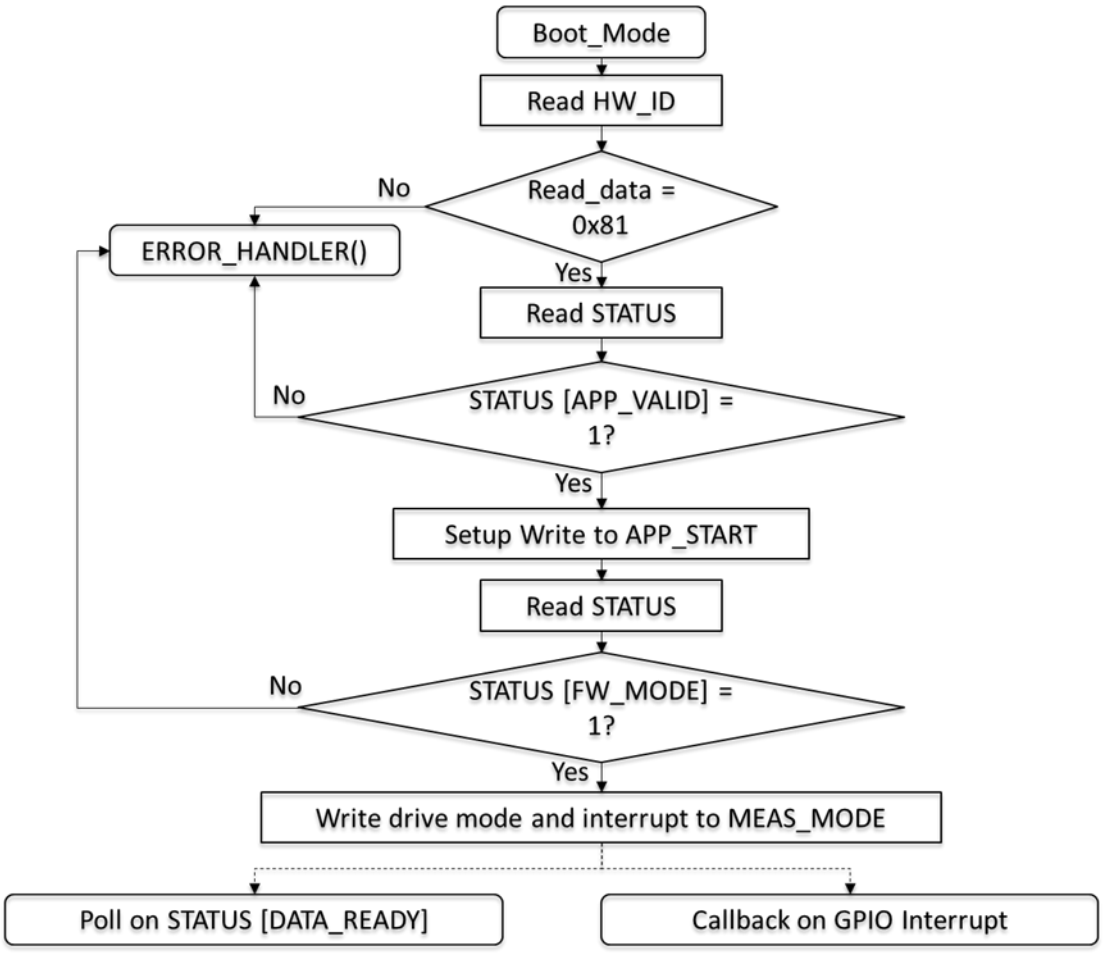
```
eco2_data = ((uint16_t)i2c_buf[0] << 8) | ((uint32_t)i2c_buf[1] << 0));
```

If the user's CPU supports endianness swapping assembly instructions then it is more efficient to use these.

# 19 Initialization Flow

The initialization flow is detailed below:

**Figure 18 CCS811 Initialization Flow**




The CCS811 powers-up in boot mode. The user software should check if a read of the HW_ID returns the expected response 0x81. If so then the STATUS should be checked to ensure there is a valid firmware image present. Then a setup write to APP_START will transition the sensor from boot mode to firmware mode. After this the STATUS register should be checked to ensure that the firmware is valid and that the sensor is now in firmware and therefore ready to start taking and processing samples from the ADC. If any checks fail then an error handle routine can be called to place the sensor and sensor's driver state to a non-operational mode.

Next the user software can initialize the desired drive mode and interrupt signal (nINT) configuration. Thereafter the user software can move it state to the process data state where it reads and processes samples of $eCO_2$/TVOC from the sensor. Not shown above is that in some applications a timeout should be used on STATUS[DATA_READY] and/or nINT. If these are not asserted within a specific time window then an error state can be declared and the user software can react in an application specific manner.

# 20 References

| Document Reference | Description |
|---|---|
| CCS811_DS_000459 | CCS811 Datasheet |
| CCS811_AN000371 | CCS811 Performing an Application code binary file download application note |

# 21 Contact Information

**Technical Support is available at:**
www.ams.com/Technical-Support

**Provide feedback about this document at:**
www.ams.com/Document-Feedback

**For further information and requests, e-mail us at:**
ams_sales@ams.com

**For sales offices, distributors and representatives, please visit:**
www.ams.com/contact

**Headquarters**
ams AG
Tobelbader Strasse 30
8141 Premstaetten
Austria, Europe

Tel: +43 (0) 3136 500 0
Website: www.ams.com

## 22 Copyrights & Disclaimer



Information in this document is believed to be accurate and reliable. However, ams AG does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information.

Applications that are described herein are for illustrative purposes only. ams AG makes no representation or warranty that such applications will be appropriate for the specified use without further testing or modification. ams AG takes no responsibility for the design, operation and testing of the applications and end-products as well as assistance with the applications or end-product designs when using ams AG products. ams AG is not liable for the suitability and fit of ams AG products in applications and end-products planned.

ams AG shall not be liable to recipient or any third party for any damages, including but not limited to personal injury, property damage, loss of profits, loss of use, interruption of business or indirect, special, incidental or consequential damages, of any kind, in connection with or arising out of the furnishing, performance or use of the technical data or applications described herein. No obligation or liability to recipient or any third party shall arise or flow out of ams AG rendering of technical or other services.

ams AG reserves the right to change information in this document at any time and without notice.

# 23 Revision Information

| Changes from previous version to current revision 2-00 (2017-Sep-26) | Page |
|---|---|
| Initial Cambridge CMOS Sensors version 1-00 | |
| Latest version 2-00 | |

**Note:** Page numbers for the previous version may differ from page numbers in the current revision.
Correction of typographical errors is not explicitly mentioned.