TELAIRE
A Division of Edwards Systems Technology

# 6000 Series Module $CO_2$ Sensor

# UART and SPI

# Communications Protocols

(Document Revision 02)

## Document Revisions

08/11/05 – Renamed document "UART_SPI_6004_X04_Protocol_02.doc",

10/03/02 – Renamed document "UART_SPI_6004_X04_Protocol_01.doc", and noted that CMD_READ and CMD_UPDATE for SNGPT_PPM available on Release 04 or later

09/25/02 – First Draft, adapted from "6000_Comm_01.doc" and "SPI_Comm_Protocol_09C.doc"

TELAIRE
A Division of Edwards Systems Technology

# 6000 Series Module $CO_2$ Sensor<br>UART and SPI<br>Communications Protocols

## Table of Contents

# 6000 Series Module $CO_2$ Sensor
# UART and SPI
# Communications Protocols

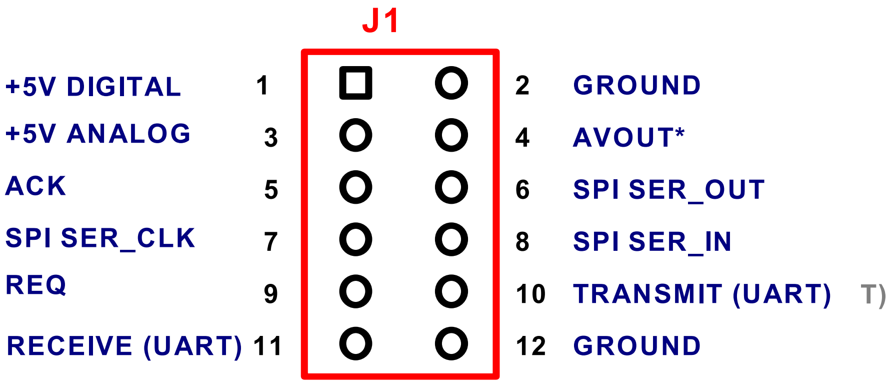
## 1. Communications Interfaces

The 6000 Series Module $CO_2$ Sensor possesses two different communications interfaces with which to communicate with an external host. The first is an asynchronous, RS-232, UART serial communications port. All communications over this UART serial interface must be wrapped in the proprietary Telaire Tsunami Communications Protocol.

The second method of communication is the Synchronous Peripheral Interface, commonly referred to as an SPI Bus communications interface. This interface in earlier documentation is referred to as a MICROWIRE[1] Bus Interface. All communications over the SPI bus require a protocol different from the Tsunami Protocol.

A physical diagram of the interface connector is shown below: (Note that these signals are received/transmitted with a 74HC244 CMOS line driver)

Fig 1.1



Both UART and SPI communications share a common command syntax and repertoire of commands. In fact, both communication interfaces share the same microcontroller resources for pointers, variables, and message buffers. Hence, the Module can respond to only one type of communication, UART or SPI, at a time.

[1] MICROWIRE™ is a registered trademark of National Semiconductor. For more information on MICROWIRE see National Semiconductor's COP8™ Microcontroller Databook or COP8 Feature Family User's Manual, (March 1998, Literature Number 620897-004.) On the web, see http://www.national.com.

# 2. UART and SPI Communications Logic and Timing

Although protocols for the two types of communications interfaces are different, both have initialization logic and timing constrains. Timing information for the Module may be considered at three levels:

- General system timing and initialization
- Timing related to byte/command transfers
- Hardware related timing (max clock rates, etc..)

## 2.1. General System Timing

When sensor is operating, the internal cycle of the data acquisition and signal processing is 2 seconds. The host could interrogate the sensor more often for this information, but generally it makes no sense and is not recommended. Therefor, it is advised to keep the communications cycle for $CO_2$ concentration requests to a multiple of 2 seconds: 2, 4, 6, …etc.

The time interval between other commands is less restricted. In general, with the exception of a Status command following a Calibration command, a subsequent command can be issued as soon as the reply from the previous command has been received. For a Status command following a Calibration command, the host should wait at least 2 to 4 seconds before issuing the first Status command. This allows time for the Module to begin the calibration process.

## 2.2. Initializing Communications at Power-Up

When Module power is first applied, or in case of power brown-outs and other forms of power failure, the Module will respond to host commands after 5 to 7 seconds. This communications delay time is necessary for the sensor to achieve full power and initialize.

After initialization, the Module stays in a Warm-up mode. The duration of the Warm-up period is configurable. Depending on the Module model, the warm-up time can be set anywhere from 6 to 60 seconds. The difference between Warm-Up and normal operating mode is that in the Warm-Up the module may not yet report accurate readings, and hence cannot execute any calibration commands. All other commands can be executed during Warm-up.

The Status of the sensor can be checked by using the Status command (see the commands description below). This command returns the status byte with a number of flags, including the Warm-Up status flag.

The Warm-up mode can be terminated by using the Skip-Warm-Up command.

The gas ppm concentration can be read while the sensor is in Warm-Up mode; however, the data may not be accurate.

The recommended sequence for the host microcontroller communications, for both UART and SPI, is:

- Power-Up
- Wait 5-7 seconds
- Start polling Status Byte every 2 seconds
- Wait for the Status Byte equal to 0
- Start polling the $CO_2$ ppm data every 2 seconds.

NOTE: If for any reason the sensor does not respond to a request, either UART or SPI, simply re-send the command.

### 2.3. Timing related to byte/command transfers

The Module's UART communications interface expects a frame size of 8 bits, no parity, one stop bit, and a baud rate of 9600. The SPI communications protocol utilizes a hand-shaking paradigm to transfer bytes between the Module and the host, which will be described in detail below.

### 2.4. Hardware related timing

The RC Oscillator, set to 2 MHz during sensor initialization, maintains the UART baud rate of 9600. The clock rate for the SPI communications interface is determined by the host. Requirements for the SPI clock rate are discussed below.

## 3. UART Serial Communications Interface

The 6000 Series Module $CO_2$ Sensor communicates over an asynchronous, UART interface at 9600 baud, no parity, 8 data bits, and 1 stop bit. When a host computer or PC communicates with the Sensor, the host computer sends a request to the Sensor, and the Sensor returns a response. The host computer acts as a master, initiating all communications, and the Sensor acts as a slave, responding with a reply.

All Sensor commands and replies are wrapped in the proprietary Telaire Tsunami Communications Protocol to insure the integrity and reliability of the data exchange. The Communications Protocol for the serial interface and the Command Set for the 6000 Series Module $CO_2$ Sensor are described in detail in the sections that follow.

### 3.1. UART Tsunami Communications Protocol

Each command to the Sensor consists of a length byte, a command byte, and any additional data required by the command. Each response from the Sensor consists of a length byte and the response data if any. Both the command to the sensor and the response from the Sensor are wrapped in the Tsunami communications protocol layer.

| | |
|---|---|
| Command: | <length><command><additional_data> |
| Response: | <length><response_data> |

The communications protocol consists of two flag bytes (0xFF) and an address byte as a header, and a two-byte CRC as a trailer. In addition, if the byte 0xFF occurs anywhere in the message body or CRC trailer, the protocol inserts a null (0x00) byte immediately following the 0xFF byte. The inserted 0x00 byte is for transmission purposes only, and is not included in the determination of the message length or the calculation of the CRC.

| Header | Message Body | Trailer |
|---|---|---|
| <flag><flag><address> | <Command/Response> | <crc_lsb><crc_msb> |

When receiving a command or response, the flags and any inserted 0x00 bytes must be stripped from the message before calculating the verification CRC. A verification CRC should be computed on all received messages from the sensor and compared with the CRC in the message trailer. If the verification CRC matches the trailer CRC, then the data from the Sensor was transmitted correctly with a high degree of certainty.

The CRC is computed only on the address and the Message Body. That is, the CRC calculation is performed on the address byte, the length byte, and all bytes of the Command (including any additional_data bytes) and the Response.

## 3.2. UART Commands from PC to Sensor

Commands sent from a host computer or PC to the Sensor have the following format:

<flag><flag><address><length><command><additional_data>< crc_lsb><crc_msb >

where:

<flag> the hex value 0xFF

<address> one byte hex value. The byte 0xFE is an address to which all sensors respond.

<length> total length in bytes of the command and additional data

<command> one byte hex command, values explained below

<additional_data> may or may not be applicable, depending upon the command

< crc_lsb><crc_msb > two byte binary CRC (algorithm given below). The CRC is little-endian, meaning that the least significant byte is given first.

For example, to request Sensor identification, the following command is used:

```
          0xFF      0x FF     0xFE      0x02      0x02      0x01      0x34      0x25
        <flag>     <flag>    <address>    |        |         |          |      <crc_msb>
                                       <length>    |         |      <crc_lsb>
                                                   |      <additional data> = SERIAL_NUMBER
                                              <command> = CMD_READ
```

The length of the command is 0x02, since the command CMD_READ, SERIAL_NUMBER consists of the two bytes “0x02 0x01”.

## 3.3. UART Response from Sensor to PC

Responses returned from the Sensor to the host computer or PC have the following format:

<flag><flag><address><length><response_data>< crc_lsb><crc_msb >

where:

<flag> the hex value 0xFF.

<address> one byte hex value. The byte 0xFA signifies “to master” in a master/slave communication.

<length> total length in bytes of the response data

<response_data> may or may not be applicable, depending upon the command

< crc_lsb><crc_msb > two byte binary CRC (algorithm given below). The CRC is little-endian, that is, the least significant byte is given first.

In response to the above identification command CMD_READ SERIAL_NUMBER, one Sensor replied with the following byte stream:

```
0xFF   0xFF   0xFA    0x09   0x4E   0x4F    0x42   0x30   0x30   0x3    0x32   0x34   0x00   0x13   0xB0
<flag>  <flag> <address>  |    <response_data>----------------------------------------------------------| <crc_lsb>  |
                          <length>                                                                            <crc_msb>
```

The nine byte response_data, “4E 4F 42 30 30 31 32 34 00”, translates to the null terminated ASCII string “NOB00124”, the serial number for that particular sensor.

### 3.4. UART Acknowledgement or <ACK> Reply

Some commands require that a Sensor only confirm that the command was received and the appropriate action was taken. In this case, when a Sensor does not need to return data in response to a command, it will instead reply with an Acknowledgement response, called an <ACK>. This is a response packet formatted as shown above, but with the <length> equal to 0x00, and no response data present:

```
0xFF    0xFF      0xFA     0x00       0x0A       0xFC
<flag>  <flag>  <address>   |       <crc_lsb>     |
                         <length>              <crc_msb>
```

Examples of commands that expect an Acknowledgement response are Update Commands, Calibrate Commands, and the Skip Warmup Command. Detailed descriptions of these commands are given below.

### 3.5. UART 0xFF Bytes and Zero Insertion

If any field other than the flag field contains the byte 0xFF, the communications protocol inserts a trailing 0x00 byte immediately following the 0xFF byte. The inserted 0x00 byte is for transmission purposes only, and is not included in the determination of the message length or the calculation of the CRC. In fact, the 0x00 byte insertion is done after the CRC is appended to the packet. Hence, if one of the CRC bytes is 0xFF, then the protocol will insert a 0x00 byte after the 0xFF CRC byte. The following table gives several examples (albeit contrived) of Zero Insertion.

| | |
|---|---|
| Req> FF FF FE 02 00 FF 00 87 4D<br>Resp> FF FF FA 01 FF 00 52 09 | The request is CMD_LOOPBACK, with data 0xFF. The <length> is 0x02, the <command> is 0x00, and the <additional_data> is 0xFF. The protocol inserts a 0x00 following the 0xFF in the <additional_data>.<br>The 0x00 <command> requests the Sensor to echo back the <additional_data> from the request packet. So the <response_data> in the response packet is the single 0xFF byte.<br>In the response the <length> is 0x01, and the <response_data> is 0xFF. The protocol inserts a 0x00 following the 0xFF in <response_data>. |

| | |
|---|---|
| Req> FF FF FE 02 00 F2 2A 9C<br>Resp> FF FF FA 01 F2 FF 00 D8 | The request is CMD_LOOPBACK, with data 0xF2. The16-bit CRC in the response is 0xD8FF, but it is written with it's least significant byte first. Since the CRC in the response contains an 0xFF the protocol inserts a 0x00 following the 0xFF. |
| Req> FF FF FE 02 00 80 FF 00 C2<br>Resp> FF FF FA 01 80 2A 86 | The request is CMD_LOOPBACK, with data 0x80. In this example the CRC in the request contains a 0xFF and so the protocol inserts a 0x00 following the 0xFF. |

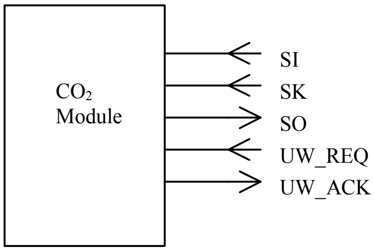
# 4. SPI Communications Interface Overview

The SPI is a serial synchronous communications interface consisting of an 8-bit serial shift register with serial data input (SI), serial data output (SO) and serial shift Clock (SK.)

The Module works as a slave on the SPI bus. The external processor is the master. This has two important consequences. First, the external processor provides the SK clock signal for both sending and receiving data across the bus. Secondly, all communications are initiated by the external processor, with the Module merely responding. From the Module's point of view, during communications with an external processor, its SI (serial in) and SK (serial clock) are inputs, and its SO (serial out) is an output.

Additionally there are two digital handshake lines that an external processor uses to communicate with the Module: UB_REQ and UB_ACK. The acronym UB stands for "MICROWIRE Bus", with the "U" being similar to the Greek letter, micron. UB_REQ is an input to the Module. UB_ACK is an output from the Module. A conceptual diagram of the input and output lines is given below:

Fig 4.1



## 4.1. SPI Handshaking – Starting To Communicate

Normally, the external processor keeps UB_REQ high and the Module keeps UB_ACK high. When the external processor wants to communicate with the Module, it lowers UB_REQ then waits until the Module lowers UB_ACK. This lowering of UB_ACK indicates that the Module is now in SPI bus slave mode and is prepared to communicate with the external processor. The external processor, as an SPI bus master, can then begin sending the bytes in the request sequence.

### 4.2. SPI Sending and Receiving Data

Every data exchange between an external processor and the Module starts with the external processor sending a request data-packet – several bytes – to the Module. The Module then responds by returning a response data-packet to the external processor. The request data packet contains a command byte, and perhaps one or more parameter bytes. Details of the commands and the data associated with each command are shown below in section "Commands." Additionally, request and response data-packets are wrapped in a packet protocol described in the section "SPI Packet Protocol."

### 4.3. SPI Handshaking – Between Bytes

After receiving each byte in a request data packet, the Module raises the UB_ACK handshaking line. When it is ready to receive the next byte it lowers UB_ACK. The external processor may send the next byte to the Module any time within 10 milliseconds of the time UB_ACK goes low. This handshaking between bytes provides flow control and insures that the external processor does not overrun the Module's input buffer and that the Module does not wait indefinitely for the external processor to send the next byte. After receiving the final byte of the request data-packet, the Module again raises UB_ACK.

The UB_REQ line remains low during the whole of the request packet operation, and during the response packet operation, and between the request and response.

When the Module has processed the request and is ready to send the first byte of the response data-packet, the Module lowers UB_ACK. The external processor has 10 milliseconds from the time the UB_ACK line goes low in order to start the clock and receive the byte. After transmitting the byte, the Module raises UB_ACK, and lowers it again when it is ready to transmit the next byte. The process continues until all bytes of the response data-packet have been transmitted to the external processor. The 10-millisecond time limit insures that the Module does not wait indefinitely for the external processor to start the clock to receive the byte.

### 4.4. SPI Handshaking – Ending a Data Exchange

After sending the final byte in a response packet, the Module raises UB_ACK and leaves it high. The external processor then raises UB_REQ, concluding the data interchange. UB_REQ must stay high longer than a specified minimum before the external processor lowers it to start any subsequent data exchange (see Timing below).

### 4.5. Aborting an SPI Data Exchange

If the external processor needs to terminate an incomplete data exchange it raises the UB_REQ line. When the Module detects this, it discards the contents of its communication buffers and responds by raising UB_ACK.

If the Module needs to terminate an incomplete data exchange, it raises UB_ACK. If UB_ACK remains high longer than the maximum time specified for UB_ACK High Between Bytes (see Timing below) then the external processor must recognize this as termination of an incomplete data exchange. For example, if the Module receives bytes that do not correspond to a valid request data-packet then it raises UB_ACK and holds it high, signaling the termination of an incomplete data exchange.

The Module starts a 10-millisecond timeout timer each time it lowers UB_ACK. The external processor must respond by starting the serial shift clock within this interval so that the module can transmit or receive the pending byte. If the external processor fails to start the clock, the Module presumes that the communication has been aborted and will raise UB_ACK.

If either the external processor or the Module terminates a data exchange, no new communication can be initiated until both UB_ACK and UB_REQ have returned to the high state. The new command then starts with the external processor lowering UB_REQ as described above.

# 5. SPI Bus Timing

During SPI communications, the external processor supplies the clock pulse for both sending and receiving data across the bus. Thus, to an external processor, the Module appears as a slave on the SPI bus.

## 5.1. SPI SK Shift Clock

To assure compatibility with the previous revisions of the product, the rev X04 modules use hardware implementation of the MICROWIRE interface emulating the interface of National Semiconductor COP8 microcontroller. Although the user is not expected to be familiar with the COP8, nevertheless some of the explanations mention COP8 implementation details for the benefit of those who are familiar.

The Module expects the SK serial clock line to be low when it is not being used to clock data over the bus. The Module samples input data on SI at the rising edge of the SK clock, and clocks (shifts) its SPI shift register on the falling edge of the SK clock. Thus, the Module's output data is available on SO for the external processor to sample at the rising edge of the clock. See the diagram below titled "MICROWIRE Timing, Standard SK Mode." In the COP8 this is described as standard SK mode with SK clock idle state low.

## 5.2. SPI Min/Max Timing Issues

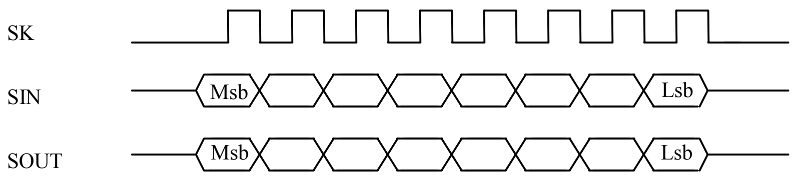
Refer to the diagrams below, Fig 5.2 "SPI Write to Module," Fig 5.3 "SPI Writing to, Then Reading from Module," and Fig 5.4 "End of SPI Reading or Writing."

Table 5.1

| Parameter | Limit | Units | Condition / Comments |
|---|---|---|---|
| $t_1$ | 780* | u-Sec | Typical delay UB_REQ low to UB_ACK low |
| $t_2$ | 0 | u-Sec Min | Minimum elapsed UB_ACK low to first SK rising edge |
| $t_3$ | 150* | u-Sec | Typical delay Byte sent/received to UB_ACK Hi. |
| $t_4$ | 200* | u-Sec Min | Typical minimum UB_ACK high between bytes |
| $t_4$ | 440* | u-Sec Max | Typical maximum UB_ACK high between bytes |
| $f_{SK\ MAX}$ | 500 | KHz Max | Maximum SK clock frequency |
| $t_5$ | 1 | u-Sec Min | Minimum SK hi/low pulse width |
| $t_7$ | 20 | n-Sec Min | Data Valid to SK rising edge Setup |
| $t_8$ | 56 | n-Sec Min | Data Valid after SK rising edge Hold |
| $t_{10}$ | 0 | u-Sec Min | UB_ACK High to UB_REQ High |
| $t_{11}$ | 680 | u-Sec Min | UB_REQ High between communications exchanges |

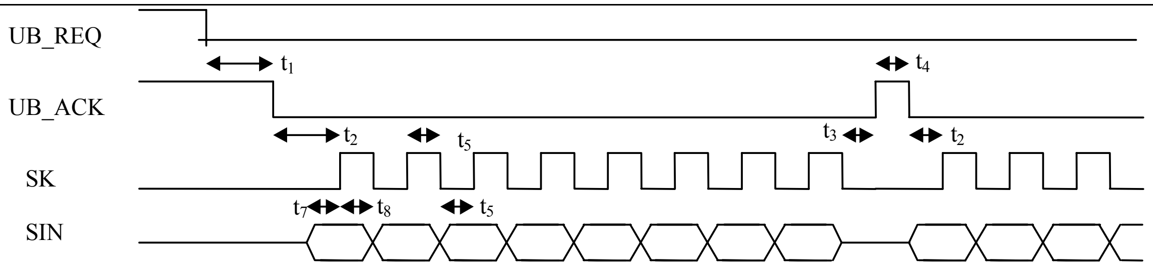
*Observed measurement - typical, not limiting.

Fig 5.1



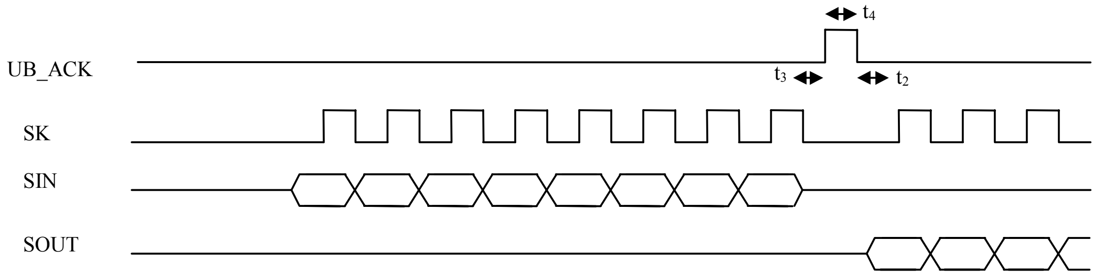
MICROWIRE Timing, Standard SK Mode

Fig 5.2



SPI Write to Module

Fig 5.3



SPI Writing to, Then Reading From Module

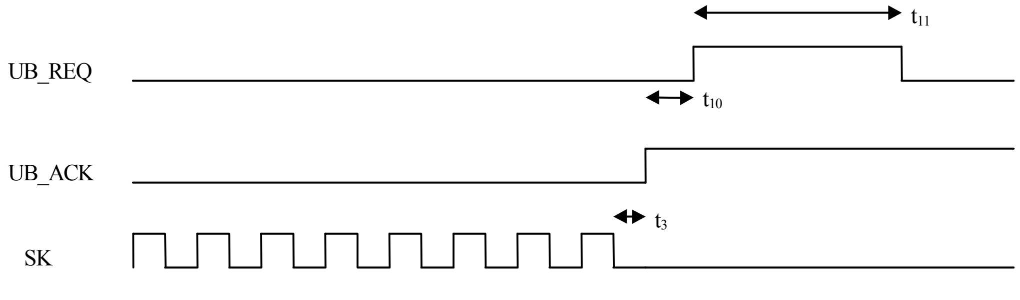


Fig 5.4 -End of SPI Reading or

# 6. SPI Packet Protocol

Every SPI exchange of data between an external processor and the Module starts with a request data-packet sent to the Module, followed by a response data-packet returned from the Module. The request and response data-packets are formatted as follows.

## 6.1. SPI Request Data-Packet

<flag> <length> <command> [<additional data> …]

<flag> – A single byte, value 0xFE. This signals the start of a request or response data-packet. If the Module receives a value other than 0xFE as the first byte in a request data-packet then it raises UB_ACK and leaves it up. This signals that it is terminating an incomplete data exchange.

<length> – A one-byte binary value from 0x01 to 0xFF. Length counts the number of bytes in the command plus any additional data. If a command has no additional data then length is 0x01. Length of 0x00 is not valid for a request data-packet. In a response data-packet a length of 0x00 indicates that the packet is an acknowledgement, "<ACK>", and no additional data bytes follow the length byte.

<command> -- A one-byte value. See Commands, below. This specifies the nature of the request and also establishes the meaning of any additional data in the request.

<additional data> -- Zero or more bytes of data, depending on the command. See Commands, below.

## 6.2. SPI Response Data-Packet

<flag> <length> [<response data> …]

<flag> – A single byte, value 0xFE. This signals the start of a request or response data-packet. If the Module receives a value other than 0xFE as the first byte in a request data-packet then it raises UB_ACK and leaves it up. This signals that it is terminating an incomplete data exchange.

<length> – A one-byte binary value from 0x00 to 0xFF. Length counts the number of bytes in response data. In a response data-packet a length of 0x00 indicates that the packet is an acknowledgement, "<ACK>", and no response data bytes follow the length byte.

<response data> -- Zero or more bytes of data, depending on the command sent in the request data-packet. See Commands, above.

## 6.3. Example SPI Data-Packets

A comprehensive list of available commands and their syntax is given in following sections. The following examples illustrate the use of the SPI packet protocol.

### 6.3.1. Request to Read Module's PPM measurement.

```
    <flag> <length> <command> [<additional data> …]
                              CMD_READ     CO2_PPM
      0xFE    0x02          0x02              0x03
```

Response indicating 419 PPM $CO_2$. (Note that 419 = 0x01A3.)

```
      <flag> <length> [<response data> …]
      0xFE     0x02          0xA3   0x01
```

### 6.3.2. Request to set  Module's elevation to 1000 feet.  (Note:  1000 = 0x03E8.)

```
      <flag> <length> <command>         [<additional data> …          ]
             CMD_UPDATE                ELEVATION       1000
      0xFE     0x04          0x03             0x0F            0xE8   0x03
```

Response indicating acknowledgement, "<ACK>."

```
      <flag> <length>
       0xFE     0x00
```

### 6.3.3. Request Module to perform "Skip Warm-up"

```
      <flag> <length> <command>
                            CMD_SKIP_WARMUP
      0xFE      0x01         0x91
```

Response indicating acknowledgement, "<ACK>."

```
      <flag> <length>
       0xFE     0x00
```

# 7. Command Reference for the 6000 Sensor

Every common exchange of data between a host processor (or PC) and the Sensor starts with a request data-packet sent to the Sensor, followed by a response data-packet returned from the Sensor. The request data-packet contains a command byte telling what data or sensor action is required. The command byte also determines what additional data is included in the request packet.

Both UART and SPI communications share a common command syntax and repertoire of commands. In fact, both communication interfaces share the same microcontroller resources for pointers, variables, and message buffers. Hence, the Module can respond to only one type of communication, UART or SPI, at a time.

*NOTE: Each request and response must be wrapped in the appropriate communications interface protocol, either UART Tsunami or SPI, as described above. The following Command Reference gives only the command syntax and response only, and omits the protocol wrapping.*

In the following Commands Tables, hex bytes are represented as '0x12' for clarity. However, when sending the byte string in a message, the '0x' notation must be omitted.

## 7.1. CMD_READ Commands

| | |
|---|---|
| CMD_READ = 0x02<br>Req: 0x02 <data ID><br>Resp: <data> [… <data>] | Read a data value or parameter from the Module<br>Request is the command byte, 0x02, followed by a 1-byte number that identifies which data value or parameter to read.<br>Response is one or more bytes of data.<br>Details of useful values that can be read from the 6000 Sensor follow. |
| CMD_READ CO2_PPM<br>Req: 0x02 0x03<br>Resp: <ppm_lsb>, <ppm_msb> | Read the CO2 PPM as measured by the Sensor.<br>Response is a 2-byte binary value, least significant byte first, giving the $CO_2$ PPM value between 0 and 65,535. |
| CMD_READ SERIAL_NUMBER<br>Req: 0x02 0x01<br>Resp: [ASCII string, null terminated, up to 16 bytes] | Read the serial number from the Sensor.<br>Response is an ASCII string of printable characters, for example "074177". The last byte of the response is a null character, 0x00. |
| CMD_READ COMPILE_SUBVOL<br>Req: 0x02 0x0D<br>Resp: [ASCII string, null terminated, up to 16 bytes] | Read the compilation subvolume for the Sensor control software. COMPILE_DATE and COMPILE_SUBVOL together identify the software version.<br>Response is an ASCII string representing, for example |

| | |
|---|---|
| | “\S53\000306”. The last byte of the response is a null character, 0x00. |
| CMD_READ COMPILE_DATE<br>Req: 0x02 0x0C<br>Resp [7-byte ASCII string, null terminated] | Read the compilation date for the $CO_2$ module control software. COMPILE_DATE and COMPILE_SUBVOL together identify the software version.<br>Response is an ASCII string representing a date, for example “000302” for March 2, 2000. The 7th byte of the response is a null character, 0x00. |
| CMD_READ ELEVATION<br>Req: 0x02 0x0F<br>Resp <elevation_lsb> <elevation_msb> | Read the elevation in feet above sea level, a required operating parameter for the Sensor. The Sensor’s elevation setting is used to estimate air pressure and is factored into the calculation of $CO_2$ PPM.<br>Response is a 2-byte binary value, least significant byte first, giving the elevation value between 0 and 65,535 feet. |
| CMD_READ SPAN_CAL_PPM<br>Req: 0x02 0x10<br>Resp: <span_lsb> <span_msb> | Read the $CO_2$ PPM used in the most recent Span Calibration of the Sensor.<br>Response is a 2-byte binary value, least significant byte first, giving the Span Gas concentration between 0 and 65,535 PPM. |
| CMD_READ SNGPT_CAL_PPM<br>Req: 0x02 0x11<br>Resp: <sngpt_lsb> < sngpt _msb> | Read the $CO_2$ PPM used in the most recent Single Point Calibration of the Sensor.<br>Response is a 2-byte binary value, least significant byte first, giving the Single Point Gas concentration between 0 and 65,535 PPM.<br>****Command available on Release 04 or later** |

## 7.2. CMD_UPDATE Commands

| | |
|---|---|
| CMD_UPDATE ELEVATION<br>Req: 0x03 0x0F <elevation_lsb> <elevation_msb><br>Resp <ACK> | Set/Write the elevation in feet above sea level, a required operating parameter for the Sensor. Elevation is expressed as a 2-byte binary value, least significant byte first.<br>Response is an “acknowledgement” or <ACK> , a response data-packet with the length byte set to zero and no data bytes.<br>The CMD_UPDATE command should be followed by the corresponding CMD_READ command to |

| | |
|---|---|
| | verify that the expected value was written. |
| CMD_UPDATE SPAN_CAL_PPM<br>Req: 0x03 0x10 <span_lsb> <span_msb><br>Resp: <ACK> | Set/Write the CO$_2$ PPM to be used in the Span Calibration of the Sensor. The Span value is a 2-byte binary value, least significant byte first, giving the Span Gas concentration between 0 and 65,535 PPM.<br>The CMD_UPDATE command should be followed by the corresponding CMD_READ command to verify that the expected value was written. |
| CMD_UPDATE SNGPT_CAL_PPM<br>Req: 0x03 0x11 < sngpt _lsb> < sngpt _msb><br>Resp: <ACK> | Set/Write the CO$_2$ PPM to be used in the Single Point Calibration of the Sensor. The Single Point value is a 2-byte binary value, least significant byte first, giving the Single Point Gas concentration between 0 and 65,535 PPM.<br>The CMD_UPDATE command should be followed by the corresponding CMD_READ command to verify that the expected value was written.<br>****Command available on Release 04 or later** |

## 7.3. RESET and WARMUP Commands

| | |
|---|---|
| CMD_WARM<br>Req: 0x84<br>Resp: <ACK> or <no response> | Reset the sensor, which puts it in a known state, similar to power up. The Sensor initializes itself, waits a period of time in warm-up mode, and then starts to measure CO$_2$ PPM. The Sensor attempts to send an <ACK>, but transmission may be aborted by the reset.<br>The Sensor experiences the same communications delay as at power-up. |
| CMD_HARD<br>Req: 0xB5<br>Resp: <ACK> or <no response> | Same as CMD_WARM |
| CMD_SKIP_WARMUP<br>Req: 0x91<br>Resp: <ACK> | When the sensor is powered up or reset, it waits for a period of time in warm-up mode before starting to measure CO$_2$ PPM. When the Sensor is in warm-up mode, CMD_SKIP_WARMUP tells the Sensor to end warm-up mode and start measuring CO$_2$ PPM.<br>The command CMD_STATUS can be used to determine if a sensor is in Warmup Mode.<br>See Communication Examples, below. |

## 7.4. CALIBRATION Commands

| | |
|---|---|
| CMD_ZERO_CALIBRATE<br>Req: 0x97<br>Resp: <ACK> | This command tells the Sensor to start zero calibration. Before sending this command, the zero gas (such as nitrogen) should be flowing to the sensor.<br>The <ACK> response indicates that the calibration request has been received.<br>To verify that calibration has started, wait 2 to 4 seconds and then send command CMD_STATUS to see if the calibration bit is set. When calibration is finished, the calibration bit in the status byte is cleared.<br>A zero calibration will not start if a Sensor is in warm-up mode or in error condition.<br>See Communication Examples, below. |
| CMD_SPAN_CALIBRATE<br>Req: 0x9A<br>Resp: <ACK> | This command tells the Sensor to start a Span calibration. (See “Span Calibration” in “Communication Examples” below.)<br>Before sending this command, the Span Gas with a known $CO_2$ PPM should be flowing to the sensor. The command CMD_UPDATE SPAN_CAL_PPM (see above), must be sent to inform the sensor about the PPM of the calibration gas.<br>The <ACK> response indicates that the calibration request has been received.<br>To verify that calibration has started, wait 2 to 4 seconds and then send command CMD_STATUS to see if the calibration bit is set. When calibration is finished, the calibration bit in the status byte is cleared.<br>A span calibration will not start if a Sensor is in warm-up mode or in error condition.<br>See Communication Examples, below. |
| CMD_SNGPT_CALIBRATE<br>Req: 0x9D<br>Resp: <ACK> | This command tells the Sensor to start a Single Point calibration.<br>Before sending this command, the Single Point Gas with a known PPM should be flowing to the sensor.<br>The command CMD_UPDATE SNGPT_CAL_PPM |

| | |
|---|---|
| | (see above), must be sent to inform the sensor about the PPM of the calibration gas.<br>The <ACK> response indicates that the calibration request has been received.<br>To verify that calibration has started, wait 2 to 4 seconds and then send command CMD_STATUS to see if the calibration bit is set. When calibration is finished, the calibration bit in the status byte is cleared.<br>A Single Point Calibration will not start if the Sensor is in warm-up mode or in error condition. |

### 7.5. STATUS and OPERATING Commands

| | |
|---|---|
| CMD_STATUS<br>Req: 0xB6<br>Resp: <status> | Read a status byte from the Sensor. The status byte indicates whether the sensor is functioning and is measuring PPM concentration.<br>The response is a single byte, <status>, of bit flags. (Note, bit 0 is the least significant bit.)<br>Bit 0: Error<br>Bit 1: Warmup Mode<br>Bit 2: Calibration<br>Bit 3: Idle Mode<br>Bits 4 - 7: (internal)<br>If a given status bit is "1", the sensor is in that state or mode. If a status bit is "0", the sensor is not in that mode. |
| CMD_IDLE_ON<br>Req: 0xB9 0x01<br>Resp: <ACK> | This command tells the Sensor to go into Idle Mode. In Idle Mode, the Lamp is turned off and no data collection takes place.<br>Issuing this command causes the Sensor to reset in order to enter Idle Mode. Hence, the Sensor experiences the same communications delay as at power-up.<br>Send the CMD_STATUS command to verify that the Sensor has entered Idle Mode (status bit 3 = 1). |
| CMD_IDLE_OFF<br>Req: 0xB9 0x02<br>Resp: <ACK> | This command tells the Sensor to exit Idle Mode and resume data collection.<br>Issuing this command causes the Sensor to reset in order to exit Idle Mode. Hence, the Sensor experiences the same communications delay as at power-up.<br>The Sensor goes through Warm-up Mode prior to resuming data collection.<br>Send the CMD_STATUS command to verify that the Sensor has come out of Idle Mode (status bit 3 = 0). |
| CMD_ABC_LOGIC<br>Req: 0xB7 0x00<br>Resp: <abc_state> | This command queries the Sensor for its ABC_LOGIC state.<br>If ABC_LOGIC is ON, <abc_state> = 0x01.<br>If ABC_LOGIC is OFF, <abc_state> = 0x02. |

| | |
|---|---|
| CMD_ ABC_LOGIC _ON<br>Req: 0xB7 0x01<br>Resp: <0x01> | This command turns the ABC_LOGIC ON. The reply <0x01> indicates that the ABC_LOGIC has been turned on. |
| CMD_ ABC_LOGIC _RESET<br>Req: 0xB7 0x03<br>Resp: <0x01> | This command turns the ABC_LOGIC ON and resets the ABC_LOGIC to its startup state. The reply <0x01> indicates that the ABC_LOGIC has been turned on. |
| CMD_ ABC_LOGIC _OFF<br>Req: 0xB7 0x02<br>Resp: <0x02> | This command turns the ABC_LOGIC OFF. The reply <0x02> indicates that the ABC_LOGIC has been turned off. |

### 7.6. Test Commands

| | |
|---|---|
| CMD_HALT<br>Req: 0x95<br>Resp: <no response> | This command is used strictly for testing. It tells the Sensor to put itself into error mode and act as though a fatal error has occurred. The sensor should automatically reset itself and go into Warmup Mode.<br>See Communication Examples, below. |
| CMD_LOOPBACK<br>Req: 0x00 <data_bytes><br>Resp: <data_bytes> | This command is used strictly for testing. The data_bytes (up to 16 bytes) following the 0x00 command are echoed back in the response packet. |

## 7.7. CMD_PEEK Commands (For Completeness Only)

The CMD_PEEK Command is included in the Command Set for completeness only. CMD_PEEK permits the inspection of a Sensor's entire RAM and ee-prom memory. Use of the command requires a detailed knowledge of the Sensors memory map and expertise in interpreting values in IEEE little-endian Floating Point format.

*It is strongly recommended that this command* ***not*** *be used unless under the specific direction of the manufacturer.*

| | |
|---|---|
| CMD_PEEK = 0x06<br>Req: 0x06 <page> <addr lsb> <count><br>Resp: <data> [… <data>] | Read bytes from the Sensor's memory, (RAM, or EE_Prom). Read <count> consecutive bytes starting at address <addr lsb> in memory page <page>. <count> must be from 1 to16. Sensor's memory is organized in 256-byte pages. Pages 0,1, & 2 are RAM. Pages 10, 11, … 17 are EE_Prom.<br>Response is <count> bytes of data ( 1 up to 16 bytes.)<br>Three examples of values that can be PEEKed are ELEVATION, SPAN_CAL_PPM, and SNGPT_CAL_PPM. |
| CMD_PEEK ELEVATION<br>Req: 0x06 0x11 0x1C 0x04<br>Resp: <elevation_ieee> | Read the elevation above sea level, an operating parameter for the Sensor.<br>Response is the elevation represented as a 4-byte, single precision, IEEE floating point, least significant byte first, (little endian.) See Appendix 3 on IEEE Floating Point.<br>This command is equivalent in function to the command CMD_READ ELEVATION, which returns the elevation as an unsigned long. |
| CMD_PEEK SPAN_CAL_PPM<br>Req: 0x06 0x11 0xA0 0x04<br>Resp: <span_cal_ppm_ieee> | Read the $CO_2$ PPM used in the most recent Span Calibration of the Sensor.<br>Response is the $CO_2$ ppm represented as a 4-byte, single precision, IEEE floating point, least significant byte first, (little endian.).<br>This command is equivalent in function to the command CMD_READ SPAN_CAL_PPM, which returns the span gas PPM as an unsigned long. |
| CMD_PEEK SNGPT_CAL_PPM<br>Req: 0x06 0x11 0xA8 0x04<br>Resp: <sngpt_cal_ppm_ieee> | Read the $CO_2$ PPM used in the most recent Single Point Calibration of the Sensor.<br>Response is the $CO_2$ ppm represented as a 4-byte, single precision, IEEE floating point, least significant byte first, (little endian.). |

| | |
|---|---|
| | This command is equivalent in function to the command CMD_READ  SNGPT_CAL_PPM, which returns the single point gas PPM as an unsigned long. |

## 7.8. CMD_POKE Commands (For Completeness Only)

The CMD_POKE Command is included in the Command Set for completeness only.  A CMD_POKE **modifies** a Sensor's RAM or ee-prom memory and ***can render a Sensor non-functional if misused***.  Use of the command requires a detailed knowledge of the Sensors memory map and expertise in representing values in IEEE little-endian Floating Point format.

***This command must not be used*** *unless under the direct specification of the manufacturer.*

| | |
|---|---|
| CMD_POKE = 0x07<br>Req: 0x07  <page> <addr lsb>  <data>  [… <data>]<br>Resp: <ACK> | Write bytes to the sensor's memory, (RAM, or EE_Prom).  Write <count> consecutive bytes starting at address <addr lsb> in memory page <page>.  <count> must be from 1 to16.  Sensor's memory is organized in 256-byte pages.  Pages 0,1, & 3 are RAM.  Pages 10, 11, … 17 are EE_Prom.<br>Response is an "acknowledgement" or <ACK> , a response data-packet with the length byte set to zero and no data bytes.<br>The CMD_POKE command should be followed by the corresponding CMD_PEEK command to verify that the expected value was written.<br>Three examples of values that can be POKEd are ELEVATION, SPAN_CAL_PPM, and SNGPT_CAL_PPM. |
| CMD_POKE   ELEVATION<br>Req: 0x07  0x11  0x1C  <elevation_ieee><br>Resp: <ACK> | Write an elevation value to the sensor's memory.  <elevation_ieee> is expressed as a 4-byte, single precision, IEEE floating point number, least significant byte first, (little endian.)  Elevation is measured in feet above sea level.  The Sensor's elevation setting is used to estimate air pressure and is factored into the calculation of $CO_2$ PPM.<br>Response is an "acknowledgement" or <ACK> , a response data-packet with the length byte set to zero and no data bytes.<br>The CMD_POKE command should be followed by the corresponding CMD_PEEK command to verify that the expected value was written.<br>This command is equivalent in function to the |

| | |
|---|---|
| | command CMD_UPDATE ELEVATION discussed above, which writes the elevation as an unsigned long. |
| CMD_POKE SPAN_CAL_PPM<br>Req: 0x07 0x11 0xA0 <span_cal_ppm_ieee><br>Resp: <ACK> | Write a $CO_2$ PPM value for the Sensor to use in Span calibration. <span_cal_ppm_ieee> is expressed as a 4-byte, single precision, IEEE floating point number, least significant byte first, (little endian.).<br>Response is an "acknowledgement" or <ACK> , a response data-packet with the length byte set to zero and no data bytes.<br>The CMD_POKE command should be followed by the corresponding CMD_PEEK command to verify that the expected value was written.<br>This command is equivalent in function to the command CMD_UPDATE SPAN_CAL_PPM discussed above, which writes the span gas PPM as an unsigned long. |
| CMD_POKE SNGPT_CAL_PPM<br>Req: 0x07 0x11 0xA8 <sngpt_cal_ppm_ieee><br>Resp: <ACK> | Write a $CO_2$ PPM value for the Sensor to use in Single Point calibration. <sngpt_cal_ppm_ieee> is expressed as a 4-byte, single precision, IEEE floating point number, least significant byte first, (little endian.).<br>Response is an "acknowledgement" or <ACK> , a response data-packet with the length byte set to zero and no data bytes.<br>The CMD_POKE command should be followed by the corresponding CMD_PEEK command to verify that the expected value was written.<br>This command is equivalent in function to the command CMD_UPDATE SNGPT_CAL_PPM discussed above, which writes the single point gas PPM as an unsigned long. |

# 8. UART Communication Examples

The following examples illustrate request and response packets with the full UART Tsunami communication protocol. Requests and responses are expressed in hexadecimal bytes. The <command> portion of a request and the <response_data> are in bold type.

## 8.1. UART Read $CO_2$ PPM

| | |
|---|---|
| Req> FF FF FE 02 **02 03** 76 05 | In the request "02 03" is CMD_READ CO2_PPM (see Command Reference, above.) |
| Resp> FF FF FA 02 **50 02** 7B B7 | In the response "50 02" is a 2-byte binary value, least significant byte first., giving the $CO_2$ PPM as 592 PPM (592 = 0x0250) |

## 8.2. UART CMD_STATUS to Verify Normal Operation

| | |
|---|---|
| Req> FF FF FE 01 **B6** 7F 0C | In the request, "B6" is CMD_STATUS (see Command Reference, above.) |
| Resp> FF FF FA 01 **00** A2 17 | In the response, "00" is the status byte. The zero value indicates that the Sensor is in normal mode where it is measuring $CO_2$ PPM. It is not in warm-up mode, it is not in calibration mode, and it is not in an error condition.<br>Further examples of CMD_STATUS are given in the examples below. |

## 8.3. UART Read and Update Elevation

In this set of interchanges we first read the Sensor's elevation parameter and find it is set at 1000 ft. Then we change the elevation setting to 2500 ft. Then we read back the new elevation setting and verify that it is set to 2500 ft.

| | |
|---|---|
| Req 1> FF FF FE 02 **02 0F** FA C4 | In request 1, "02 0F" is CMD_READ, ELEVATION (see Command Reference, above.) |
| Resp1>.FF FF FA 02 **E8 03** FE 30 | In the first response, "E8 03" is the elevation, 1000 ft (1000 = 0x03E8). |
| Req 2> FF FF FE 04 **03 0F C4 09** 4D 64 | In request 2, "03 0F" is CMD_UPDATE, ELEVATION, and "C4 09" is the elevation, 2500 ft (2500 = 0x09C4). |
| Resp2> FF FF FA **00** 0A FC | The second response is an <ACK>, since the length is 0x00. |
| Req 3> FF FF FE 02 **02 0F** FA C4 | The third request and response are formatted just like the |

| | |
|---|---|
| Resp3> FF FF FA 02 **C4** **09** 3F D2 | first, reading back the new elevation setting, 2500 ft. |

### 8.4. UART Error Simulation with Recovery

In this set of interchanges we first verify that the Sensor is operating normally. Then we send a command that forces the sensor into an error state. The Sensor automatically recovers by resetting itself, and going into Warmup Mode. We then send the command to skip warm-up, thus putting the sensor back into the normal state.

| | |
|---|---|
| Req 1> FF FF FE 01 **B6** 7F 0C | Req 1: CMD_STATUS. |
| Resp1> FF FF FA 01 **00** A2 17 | Resp1: status byte is 0x00. Sensor is in normal mode, measuring CO2 PPM. |
| Req 2> FF FF FE 01 **95** 7E 18 | Req 2: CMD_HALT. Puts sensor in error mode. No response |
| Req 3> FF FF FE 01 **B6** 7F 0C | Req 3: CMD_STATUS. |
| Resp3> FF FF FA 01 **02** E0 37 | Resp3: status byte is 0x02. Bit 1 high indicates Sensor is in warm-up mode.<br>(If CMD_STATUS is sent quickly enough, the sensor may respond with 0x01, indicating the brief error state prior to reset.) |
| Req 4> FF FF FE 01 **91** FA 58 | Req 4: CMD_SKIP_WARMUP. |
| Resp4> FF FF FA **00** 0A FC | Resp4: <ACK> |
| Req 5> FF FF FE 01 **B6** 7F 0C | Req 5: CMD_STATUS |
| Resp5> FF FF FA 01 **00** A2 17 | Resp5: status byte is 0x00. Sensor is in normal mode, measuring CO2 PPM. |

## 8.5. UART Zero Calibration

In this set of interchanges we run a zero calibration on the Sensor. Before sending any commands we start flowing a zero gas, like nitrogen, to the Sensor. Then we verify that the sensor is in normal operating mode, since calibration will not work if the sensor is not in normal operating mode. Then we send the zero calibration command to start the calibration process. We check the Sensor's status and see that it is in calibration mode. Later we check the status again and see that the Sensor has finished calibration and returned to normal operating mode.

| | |
|---|---|
| Req 1> FF FF FE 01 **B6** 7F 0C | Req 1: CMD_STATUS. |
| Resp1> FF FF FA 01 **00** A2 17 | Resp1: status byte is 0x00. Sensor is in normal mode, measuring CO2 PPM. |
| Req 2> FF FF FE 01 **97** 3C 38 | Req 2: CMD_ZERO_CALIBRATE. Starts the calibration process. |
| Resp2> FF FF FA **00** 0A FC | Resp2: <ACK> |
| | Wait 2 – 4 seconds. |
| Req 3> FF FF FE 01 **B6** 7F 0C | Req 3: CMD_STATUS. |
| Resp3> FF FF FA 01 **04** 26 57 | Resp3: status byte is 0x04. Sensor is in calibration mode. |
| Req 4> FF FF FE 01 **B6** 7F 0C | Req 4: CMD_STATUS. |
| Resp4> FF FF FA 01 **00** A2 17 | Resp4: status byte is 0x00. Sensor is in normal mode, measuring CO2 PPM. |

### 8.6. UART Span Calibration

In this set of interchanges we run a span calibration on the Sensor. The span calibration adjusts the Sensor’s zero calibration settings in such a way as to make the Sensor’s $CO_2$ PPM measurement match the span PPM value

In this example, we are flowing a 2000 PPM $CO_2$ gas to the Sensor. We update SPAN_CAL_PPM to 2000 so this value will be used as the span gas concentration for span calibration. Then we start the span calibration and check the Sensor’s status until the calibration is completed.

| | |
|---|---|
| Req 1> FF FF FE 04 **03 10 D0 07** 66 25 | Req 1: “03 10” is CMD_UPDATE SPAN_CAL_PPM, and “D0 07” is 2000 PPM (2000 = 0x07D0). |
| Resp1> FF FF FA **00** 0A FC | Resp1: <ACK> |
| Req 2> FF FF FE 01 **9A** 91 E9 | Req 2: CMD_SPAN_CALIBRATE. Starts the calibration process. |
| Resp2> FF FF FA **00** 0A FC | Resp2: <ACK> |
| | Wait 2 – 4 seconds. |
| Req 3> FF FF FE 01 **B6** 7F 0C | Req 3: CMD_STATUS. |
| Resp3> FF FF FA 01 **04** 26 57 | Resp3: status byte is 0x04. Sensor is in calibration mode. |
| Req 4> FF FF FE 01 **B6** 7F 0C | Req 4: CMD_STATUS. |
| Resp4> FF FF FA 01 **00** A2 17 | Resp4: status byte is 0x00. Sensor is in normal mode, measuring CO2 PPM. |

## 9. SPI Communication Examples

The following examples illustrate request and response packets with the SPI communications protocol. These are the same commands that were listed above for the UART communications protocol. Requests and responses are expressed in hexadecimal bytes. The <command> portion of a request and the <response_data> are in bold type.

### 9.1. SPI Read $CO_2$ PPM

| | |
|---|---|
| Req> FE 02 **02 03** | In the request “02 03” is CMD_READ CO2_PPM |
| Resp> FE 02 **50 02** | In the response “50 02” is a 2-byte binary value, least significant byte first., giving the $CO_2$ PPM as 592 PPM (592 = 0x0250) |

### 9.2. SPI CMD_STATUS to Verify Normal Operation

| | |
|---|---|
| Req> FE 01 **B6** | In the request, “B6” is CMD_STATUS |
| Resp> FE 01 **00** | In the response, “00” is the status byte. The zero value indicates that the Sensor is in normal mode where it is measuring $CO_2$ PPM. It is not in warm-up mode, it is not in calibration mode, and it is not in an error condition.<br>Further examples of CMD_STATUS are given in the examples below. |

### 9.3. SPI Read and Update Elevation

In this set of interchanges we first read the Sensor’s elevation parameter and find it is set at 1000 ft. Then we change the elevation setting to 2500 ft. Then we read back the new elevation setting and verify that it is set to 2500 ft.

| | |
|---|---|
| Req 1> FE 02 **02 0F** | In request 1, “02 0F” is CMD_READ ELEVATION |
| Resp1>.FE 02 **E8 03** | In the first response, “E8 03” is the elevation, 1000 ft (1000 = 0x03E8). |
| Req 2> FE 04 **03 0F C4 09** | In request 2, “03 0F” is CMD_UPDATE ELEVATION, and “C4 09” is the elevation, 2500 ft (2500 = 0x09C4). |
| Resp2> FE **00** | The second response is an <ACK>, since the length is 0x00. |
| Req 3> FE 02 **02 0F** | The third request and response are formatted just like the first, reading back the new elevation setting, 2500 ft. |
| Resp3> FE 02 **C4 09** | |

### 9.4. SPI Error Simulation with Recovery

In this set of interchanges we first verify that the Sensor is operating normally. Then we send a command that forces the sensor into an error state. The Sensor automatically recovers by resetting itself, and going into Warmup Mode. We then send the command to skip warm-up, thus putting the sensor back into the normal state.

| | |
|---|---|
| Req 1> FE 01 **B6** | Req 1: CMD_STATUS. |
| Resp1> FE 01 **00** | Resp1: status byte is 0x00. Sensor is in normal mode, measuring CO2 PPM. |
| Req 2> FE 01 **95** | Req 2: CMD_HALT. Puts sensor in error mode. No |

| | |
|---|---|
| | response |
| Req 3> FE 01 **B6** | Req 3: CMD_STATUS. |
| Resp3> FE 01 **02** | Resp3: status byte is 0x02.  Bit 1 high indicates Sensor is in warm-up mode.<br>(If CMD_STATUS is sent quickly enough, the sensor may respond with 0x01, indicating the brief error state prior to reset.) |
| Req 4> FE 01 **91** | Req 4: CMD_SKIP_WARMUP. |
| Resp4> FE **00** | Resp4: <ACK> |
| Req 5> FE 01 **B6** | Req 5: CMD_STATUS |
| Resp5> FE 01 **00** | Resp5: status byte is 0x00.  Sensor is in normal mode, measuring CO2 PPM. |

### 9.5. SPI Zero Calibration

In this set of interchanges we run a zero calibration on the Sensor.  Before sending any commands we start flowing a zero gas, like nitrogen, to the Sensor.   Then we verify that the sensor is in normal operating mode, since calibration will not work if the sensor is not in normal operating mode.  Then we send the zero calibration command to start the calibration process.  We check the Sensor's status and see that it is in calibration mode.  Later we check the status again and see that the Sensor has finished calibration and returned to normal operating mode.

| | |
|---|---|
| Req 1> FE 01 **B6** | Req 1: CMD_STATUS. |
| Resp1> FE 01 **00** | Resp1: status byte is 0x00.  Sensor is in normal mode, measuring CO2 PPM. |
| Req 2> FE 01 **97** | Req 2: CMD_ZERO_CALIBRATE.  Starts the calibration process. |
| Resp2> FE **00** | Resp2: <ACK> |
| | Wait 2 – 4 seconds. |
| Req 3> FE 01 **B6** | Req 3: CMD_STATUS. |
| Resp3> FE 01 **04** | Resp3: status byte is 0x04.  Sensor is in calibration mode. |
| Req 4> FE 01 **B6** | Req 4: CMD_STATUS. |
| Resp4> FE 01 **00** | Resp4: status byte is 0x00.  Sensor is in normal mode, measuring CO2 PPM. |

### 9.6. SPI Span Calibration

In this set of interchanges we run a span calibration on the Sensor. The span calibration adjusts the Sensor's zero calibration settings in such a way as to make the Sensor's $CO_2$ PPM measurement match the span PPM value

In this example, we are flowing a 2000 PPM $CO_2$ gas to the Sensor. We update SPAN_CAL_PPM to 2000 so this value will be used as the span gas concentration for span calibration. Then we start the span calibration and check the Sensor's status until the calibration is completed.

| | |
|---|---|
| Req 1> FE 04 **03 10 D0 07** | Req 1: "03 10" is CMD_UPDATE SPAN_CAL_PPM, and "D0 07" is 2000 PPM (2000 = 0x07D0). |
| Resp1> FE **00** | Resp1: <ACK> |
| Req 2> FE 01 **9A** | Req 2: CMD_SPAN_CALIBRATE. Starts the calibration process. |
| Resp2> FE **00** | Resp2: <ACK> |
| | Wait 2 – 4 seconds. |
| Req 3> FE 01 **B6** | Req 3: CMD_STATUS. |
| Resp3> FE 01 **04** | Resp3: status byte is 0x04. Sensor is in calibration mode. |
| Req 4> FE 01 **B6** | Req 4: CMD_STATUS. |
| Resp4> FE 01 **00** | Resp4: status byte is 0x00. Sensor is in normal mode, measuring CO2 PPM. |

# Appendix 1. CRC Calculation

## A1.1 CalcCRC

Below is a sample ‘C’ Subroutine for calculating the 2 byte CRC used in the communications protocol:

```
const unsigned int crc_tab[256] =
{
        0x0000,  0x1021,  0x2042,  0x3063,  0x4084,  0x50A5,  0x60C6,  0x70E7,
        0x8108,  0x9129,  0xA14A,  0xB16B,  0xC18C,  0xD1AD,  0xE1CE,  0xF1EF,
        0x1231,  0x0210,  0x3273,  0x2252,  0x52B5,  0x4294,  0x72F7,  0x62D6,
        0x9339,  0x8318,  0xB37B,  0xA35A,  0xD3BD,  0xC39C,  0xF3FF,  0xE3DE,
        0x2462,  0x3443,  0x0420,  0x1401,  0x64E6,  0x74C7,  0x44A4,  0x5485,
        0xA56A,  0xB54B,  0x8528,  0x9509,  0xE5EE,  0xF5CF,  0xC5AC,  0xD58D,
        0x3653,  0x2672,  0x1611,  0x0630,  0x76D7,  0x66F6,  0x5695,  0x46B4,
        0xB75B,  0xA77A,  0x9719,  0x8738,  0xF7DF,  0xE7FE,  0xD79D,  0xC7BC,
        0x48C4,  0x58E5,  0x6886,  0x78A7,  0x0840,  0x1861,  0x2802,  0x3823,
        0xC9CC,  0xD9ED,  0xE98E,  0xF9AF,  0x8948,  0x9969,  0xA90A,  0xB92B,
        0x5AF5,  0x4AD4,  0x7AB7,  0x6A96,  0x1A71,  0x0A50,  0x3A33,  0x2A12,
        0xDBFD,  0xCBDC,  0xFBBF,  0xEB9E,  0x9B79,  0x8B58,  0xBB3B,  0xAB1A,
        0x6CA6,  0x7C87,  0x4CE4,  0x5CC5,  0x2C22,  0x3C03,  0x0C60,  0x1C41,
        0xEDAE,  0xFD8F,  0xCDEC,  0xDDCD,  0xAD2A,  0xBD0B,  0x8D68,  0x9D49,
        0x7E97,  0x6EB6,  0x5ED5,  0x4EF4,  0x3E13,  0x2E32,  0x1E51,  0x0E70,
        0xFF9F,  0xEFBE,  0xDFDD,  0xCFFC,  0xBF1B,  0xAF3A,  0x9F59,  0x8F78,
        0x9188,  0x81A9,  0xB1CA,  0xA1EB,  0xD10C,  0xC12D,  0xF14E,  0xE16F,
        0x1080,  0x00A1,  0x30C2,  0x20E3,  0x5004,  0x4025,  0x7046,  0x6067,
        0x83B9,  0x9398,  0xA3FB,  0xB3DA,  0xC33D,  0xD31C,  0xE37F,  0xF35E,
        0x02B1,  0x1290,  0x22F3,  0x32D2,  0x4235,  0x5214,  0x6277,  0x7256,
        0xB5EA,  0xA5CB,  0x95A8,  0x8589,  0xF56E,  0xE54F,  0xD52C,  0xC50D,
        0x34E2,  0x24C3,  0x14A0,  0x0481,  0x7466,  0x6447,  0x5424,  0x4405,
        0xA7DB,  0xB7FA,  0x8799,  0x97B8,  0xE75F,  0xF77E,  0xC71D,  0xD73C,
        0x26D3,  0x36F2,  0x0691,  0x16B0,  0x6657,  0x7676,  0x4615,  0x5634,
        0xD94C,  0xC96D,  0xF90E,  0xE92F,  0x99C8,  0x89E9,  0xB98A,  0xA9AB,
        0x5844,  0x4865,  0x7806,  0x6827,  0x18C0,  0x08E1,  0x3882,  0x28A3,
        0xCB7D,  0xDB5C,  0xEB3F,  0xFB1E,  0x8BF9,  0x9BD8,  0xABBB,  0xBB9A,
        0x4A75,  0x5A54,  0x6A37,  0x7A16,  0x0AF1,  0x1AD0,  0x2AB3,  0x3A92,
        0xFD2E,  0xED0F,  0xDD6C,  0xCD4D,  0xBDAA,  0xAD8B,  0x9DE8,  0x8DC9,
        0x7C26,  0x6C07,  0x5C64,  0x4C45,  0x3CA2,  0x2C83,  0x1CE0,  0x0CC1,
        0xEF1F,  0xFF3E,  0xCF5D,  0xDF7C,  0xAF9B,  0xBFBA,  0x8FD9,  0x9FF8,
        0x6E17,  0x7E36,  0x4E55,  0x5E74,  0x2E93,  0x3EB2,  0x0ED1,  0x1EF0
};

WORD CalcCRC(WORD wAccum, BYTE byte)
{
      return (wAccum << 8) ^ crc_tab[((BYTE)(wAccum>>8))^byte];
}
```

## A1.2 Example Calling CalcCRC

Following is ‘C’ code that wraps the communications protocol around a request packet. The <command> and <additional_data> are initially in the array `bIssue[ ]`. ESCAPE is a literal for the 0xFF <flag> character. The length of the <command> and <additional_data> is in `wLen`. The <address> is in `lpTSU->ComTarget`. And the fully wrapped request packet is placed in the array `pbPacket[ ]`.

```
j = 0;
pbPacket[j++] = ESCAPE;
pbPacket[j++] = ESCAPE;
pbPacket[j++] = (BYTE)lpTSU->ComTarget;
wCrc          = CalcCRC(0,(BYTE)lpTSU->ComTarget);
pbPacket[j++] = (BYTE)wLen;
if (wLen == ESCAPE) {
        pbPacket[j++] = 0;
}
wCrc = CalcCRC(wCrc,(BYTE)wLen);
for (i = 0; i < wLen; i++) {
        pbPacket[j++] = bIssue[i];
        wCrc = CalcCRC(wCrc,bIssue[i]);
        if (bIssue[i] == ESCAPE) {
                pbPacket[j++] = 0;      // No CRC on transport material
        }
}
pbPacket[j++] = (BYTE)wCrc;
if (pbPacket[j-1] == ESCAPE) {
        pbPacket[j++] = 0;
}
pbPacket[j++] = (BYTE)((wCrc & 0xFF00)>>8);
if (pbPacket[j-1] == ESCAPE) {
        pbPacket[j++] = 0;
}
```

# Appendix 2.  Summary of Commands

### CMD_READ Commands

| Command | Request | Response |
|---|---|---|
| CMD_READ | 0x02  <data ID> | <data>, [… <data>] |
| CMD_READ  CO2_PPM | 0x02  0x03 | <ppm_lsb>  <ppm_msb> |
| CMD_READ  SERIAL_NUMBER | 0x02  0x01 | [ASCII string, null terminated, up to 16 bytes] |
| CMD_READ  COMPILE_SUBVOL | 0x02  0x0D | [12-byte ASCII string, null terminated] |
| CMD_READ  COMPILE_DATE | 0x02  0x0C | [7-byte ASCII string, null terminated] |
| CMD_READ  ELEVATION | 0x02  0x0F | <elev_lsb>  <elev_msb> |
| CMD_READ  SPAN_CAL_PPM | 0x02  0x10 | <span_lsb>  <span_msb> |
| CMD_READ  SNGPT_CAL_PPM<br>**Command available on Release 04 or later | 0x02  0x11 | <sngpt_lsb>  <sngpt_msb> |

### CMD_UPDATE Commands

| Command | Request | Response |
|---|---|---|
| CMD_UPDATE   ELEVATION | 0x03  0x0F   <elev_lsb>  <elev_msb> | <ACK> |
| CMD_UPDATE   SPAN_CAL_PPM | 0x03  0x10   <span_lsb>  <span_msb> | <ACK> |
| CMD_UPDATE   _CAL_PPM<br>**Command available on Release 04 or later | 0x03  0x11   <sngpt_lsb>  <sngpt_msb> | <ACK> |

### RESET and WARMUP Commands

| Command | Request | Response |
|---|---|---|
| CMD_WARM | 0x84 | <ACK> or <no response> |
| CMD_HARD | 0xB5 | <ACK> or <no response> |
| CMD_SKIP_WARMUP | 0x91 | <ACK> |

### CALIBRATION Commands

| Command | Request | Response |
|---|---|---|
| CMD_ZERO_CALIBRATE | 0x97 | <ACK> |
| CMD_SPAN_CALIBRATE | 0x9A | <ACK> |
| CMD_SNGPT_CALIBRATE | 0x9D | <ACK> |

### STATUS and OPERATING Commands

| Command | Request | Response |
|---|---|---|
| CMD_STATUS | 0xB6 | <status> |
| CMD_IDLE_ON | 0xB9  0x01 | <ACK> |
| CMD_IDLE_OFF | 0xB9  0x02 | <ACK> |
| CMD_ABC_LOGIC | 0xB7  0x00 | <abc_state> |
| CMD_ABC_LOGIC_ON | 0xB7  0x01 | <0x01> |
| CMD_ABC_LOGIC_RESET | 0xB7  0x03 | <0x01> |
| CMD_ABC_LOGIC_OFF | 0xB7  0x02 | <0x02> |

### TEST Commands

| Command | Request | Response |
|---|---|---|
| CMD_HALT | 0x95 | <no response> |
| CMD_LOOPBACK | 0x00  <data_bytes> | <data_bytes> |

### CMD_PEEK Commands (For Completeness Only)

*It is strongly recommended that this command* ***not*** *be used unless under the specific direction of the manufacturer*

| Command | Request | Response |
|---|---|---|
| CMD_PEEK | 0x06  <page>  <addr lsb>   <count> | <data>  [… <data>] |
| CMD_PEEK  ELEVATION | 0x06  0x11  0x1C  0x04 | <elevation, ieee little-endian> |
| CMD_PEEK  SPAN_CAL_PPM | 0x06  0x11  0xA0  0x04 | <span_cal_ppm, ieee> |
| CMD_PEEK  SNGPT_CAL_PPM | 0x06  0x11  0xA8  0x04 | <sngpt_cal_ppm, ieee> |

### CMD_POKE Commands (For Completeness Only)

**This command must not be used** unless under the direct specification of the manufacturer.

| Command | Request | Response |
|---|---|---|
| CMD_POKE | 0x07 <page> <addr lsb> <data>[…<data>] | <ACK> |
| CMD_POKE  ELEVATION | 0x07  0x11  0x1C  <elevation,ieee> | <ACK> |
| CMD_POKE   SPAN_CAL_PPM | 0x07  0x11  0xA0  <span_cal_ppm, ieee> | <ACK> |
| CMD_POKE   SNGPT_CAL_PPM | 0x07  0x11  0xA8  <sngpt_cal_ppm, ieee> | <ACK> |

## Appendix 3. IEEE Floating Point

Some Sensor commands use data formatted as 4-byte, single precision, IEEE floating point, least significant byte first, (little endian.) Following is a description of that numerical format. Although this description depicts the “big endian” implementation, the 6000 Module $CO_2$ Sensor uses a “little endian” implementation. That is, the order of the bytes in the Sensor is reversed, so that byte #0 is stored at the next higher address from byte #1, which is stored at the next higher address from byte #2, etc.

```
//***************************************************************************//
//                                                                           //
//            IEEE 754 4-BYTE FLOATING POINT FORMAT (BIG ENDIAN)             //
//                                                                           //
//      <-- Byte #0 --> <-- Byte #1 --> <-- Byte #2 --> <-- Byte #3 -->      //
//      ---------------------------------------------------------------      //
//     |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|     //
//      ---------------------------------------------------------------      //
//     |S|   Exponent    |                   Mantisa                   |     //
//      ---------------------------------------------------------------      //
//    > 1 <----- 8 -----> <------------------- 23 -------------------->      //
//                                                                           //
//      Byte 0 is the most significant byte.                                 //
//      Byte 3 is the least significant byte.                                //
//                                                                           //
//      The S bit (located at byte 0 bit 7) flags the sign of the            //
//      floating point number. This format does NOT use two's                //
//      complement encoding. The S bit is defined as follows:                //
//          0 => Positive                                                    //
//          1 => Negative                                                    //
//                                                                           //
//      The exponent (located in bytes 0 and 1) is 8 bits long and           //
//      is positive biased. In the special case of a zero exponent,          //
//      the entire value of the floating point number is said to be          //
//      zero and all bits should be cleared.                                 //
//                                                                           //
//      The mantissa (located in bytes 1 through 3) is 23 bits long,         //
//      but carries 24 bits of information. The implied bit is               //
//      located in the most significant (bit 23) position. If the            //
//      exponent is zero, bit 23 (and all other bits) are clear. If          //
//      the exponent is non-zero, bit 23 is set.                             //
//                                                                           //
//      When calculating the value of a floating point number that           //
//      has been stored in this format, one assigns the value 0.5            //
//      (1/2) to bit 23, 0.25 (1/4) to bit 22, 0.125 (1/8) to bit            //
//      21, and so on. If the exponent is non-zero, and so the               //
//      implied bit 23 is set, the value will fall between one half          //
//      and below one.                                                       //
//                                                                           //
//      This number (between 0.5 and 1.0) is then multiplied by two          //
//      raised to the (exponent-126) power. For example, if the              //
//      exponent contains the binary value 127 and the mantissa is           //
//      all zeros (except for the implied bit 23), the value this            //
//      number is 0.5*2^(127-126) = 1.0 (-1.0 if the sign bit is set).       //
//                                                                           //
//***************************************************************************//
```