



---

Application Note: AZD062  
IQ Switch<sup>®</sup> - ProxSense<sup>®</sup> Series  
IQS253 Communication and Interface Guideline

---

## Contents

<b>1</b>	<b>INTRODUCTION</b> .....	<b>2</b>
<b>2</b>	<b>COMMUNICATION PROTOCOL</b> .....	<b>3</b>
2.1	BUS CHARACTERISTICS .....	4
2.2	CONTROL BYTE AND DEVICE ADDRESS .....	6
<b>3</b>	<b>IQS253 COMMUNICATION WINDOW</b> .....	<b>7</b>
3.1	USING THE RDY LINE .....	7
3.2	ACKNOWLEDGE POLLING .....	7
3.3	INITIAL WINDOW .....	8
<b>4</b>	<b>WRITING TO OR READING FROM IQS253</b> .....	<b>9</b>
4.1	WRITE OPERATION .....	11
4.2	READ OPERATION .....	14
<b>5</b>	<b>ADJUSTING SETTING FOR IQS253</b> .....	<b>17</b>

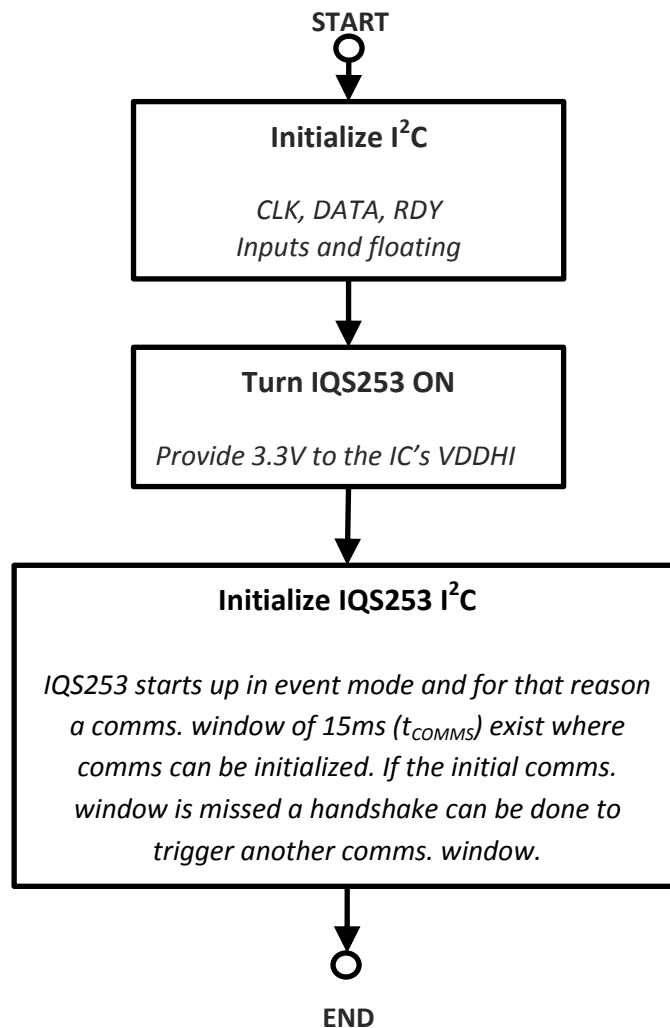


## 1 Introduction

This application note is designed to guide the reader through the process of setting up the communication interface between the ProxSense<sup>®</sup> IQS253 IC and any MCU capable of I<sup>2</sup>C (400kBit/s) communication. This is done through easy to understand flow diagrams as well as providing the source code in listings throughout the document.

In Figure 1 below an overview flow diagram is shown to provide the reader with an overview of what is discussed within this document.

The complete source code is available on request.



**Figure 1: Initialize I<sup>2</sup>C Flow Diagram**

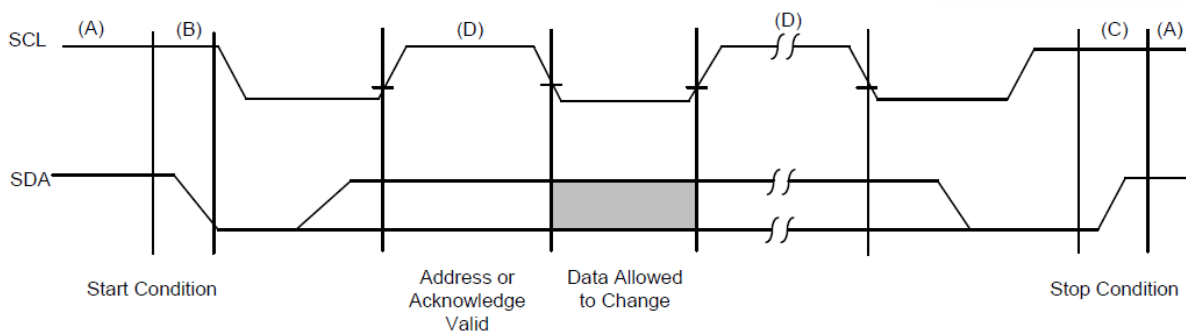


## 2 Communication Protocol

The IQS253 uses a bi-directional 2-wire bus and data transmission protocol. The serial protocol is I2C™ compatible. The IQS253 has an optional ready (RDY) pin which indicates when the device enters its communication window period. Communication with the device can only take place in this state, this can be determined by monitoring the RDY line or by using ACK polling. The IQS253 only functions as a slave device on the bus. The bus is controlled by a master device which generates the serial clock (SCL), controls bus access, and generates the START and STOP conditions. The serial clock (SCL) and serial data (SDA) lines are open-drain and therefore must be pulled high to the operating voltage with a pull-up resistor (4.7kΩ recommended). The RDY pin functions as an open-drain pin and should always be pulled to the operating voltage of the master device via a resistor (100kΩ recommended).

During the communication window period the RDY line will remain low (high for pre-production engineering versions of the IC) for a selectable duration of always/2ms (See datasheet for selection options). If the master does not initiate a data transfer during this time, the device will exit the communication window and continue doing conversions. During the communication window the address pointer will default to the value specified in the DEFAULT\_ADDR register. Using this method the user can simply start reading without having to set the address pointer first. The RDY line will remain low for the duration of the communication window period.

In the figure below (Figure 2) the data transfer sequence for the communication protocol is shown as an overview of what is explained within this section.



**Figure 2: Data Transfer Sequence on the Serial Bus.**



## 2.1 Bus Characteristics

The following bus protocol has been defined:

- Data transfer may only be initiated when the bus is not busy
- During data transfer, the data line must remain stable whenever the clock line is HIGH. Changes in the data line while the clock is HIGH will be interpreted as START and STOP conditions.

The following conditions have been defined for the bus: (refer to Figure 2)

- **Bus Idle (A)** - The SCL and SDA lines are both HIGH.
- **START Condition (B)** - A HIGH to LOW transition of the SDA while the SCL is HIGH. All serial communication must be preceded by a START condition.

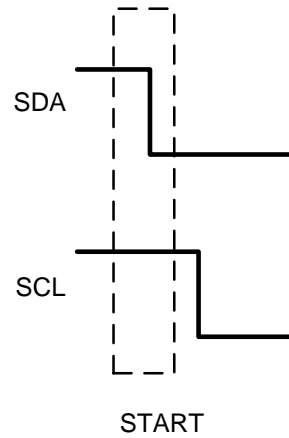


Figure 3: Start Condition.

### Listing 1. START Condition.

```
/*  
    Generate Start Condition  
*/  
void i2c_start(void)  
{  
    I2C_CLK_IN_FLOAT; // Set I2C CLK pin as input and floating  
    while(!I2C_CLK_IN); // Wait for I2C CLK line low (clock stretching)  
    wait(2); // Wait two I2C clock cycles  
    I2C_DATA_IN_FLOAT; // Set I2C DATA pin as input and floating  
    wait(2); // Wait two I2C clock cycles  
    while(I2C_RDY_IN); // Wait while ready high (This could take long in event mode)  
    // RDY checks could also be done before generating a start condition  
    wait(100); // Delay 100 clock cycles  
    I2C_DATA_OUT_PP_LOW; // Set I2C DATA pin as output (push-pull) and floating  
    wait(2); // Wait two I2C clock cycles  
    I2C_CLK_OUT_PP_LOW; // Set I2C CLK pin as output (push-pull) and floating  
    wait(2); // Wait two I2C clock cycles  
}
```



- **STOP Condition (C)** - A LOW to HIGH transition of the SDA while the SCL is HIGH. All serial communication must be ended by a STOP condition. NOTE: When a STOP condition is sent the device will exit the communications window and continue with conversions.

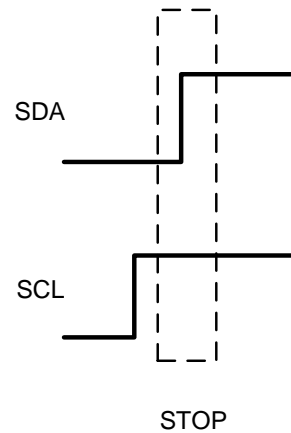


Figure 4: Stop Condition.

Listing 2. STOP Condition.

```
/*  
    Generate Stop Condition  
*/  
void i2c_stop(void)  
{  
    I2C_CLK_OUT_PP_LOW; // Set I2C CLK pin as output (push-pull) and floating  
    wait(2); // Wait two I2C clock cycles  
    I2C_DATA_OUT_PP_LOW; // Set I2C DATA pin as output (push-pull) and floating  
    wait(2); // Wait two I2C clock cycles  
    I2C_CLK_IN_FLOAT; // Set I2C CLK pin as input and floating  
    while(!I2C_CLK_IN); // Wait for I2C CLK line low (clock stretching)  
    wait(2); // Wait two I2C clock cycles  
    I2C_DATA_IN_FLOAT; // Set I2C DATA pin as input and floating  
    wait(2); // Wait two I2C clock cycles  
}
```

- **Data Valid (D)** - The state of the SDA line represents valid data when, after a START condition, the SDA is stable for the duration of the HIGH period of the clock signal. The data on the line must be changed during the LOW period of the clock signal. There is one clock pulse per bit of data. Each data transfer is initiated with a START condition and terminated with a STOP condition.
- **Acknowledge** - The slave device must generate an acknowledge after the reception of each byte. The master device must generate an extra (9th) clock pulse which is associated with this acknowledge bit. The device that acknowledges, has to pull down the SDA line during the acknowledge clock pulse. NOTE: The IQS253 does not generate any acknowledge bits while it is not in its communication window.



Listing 3. Check for Acknowledge.

```
/*  
    Check for acknowledge  
*/  
unsigned char i2c_ack_check(void)  
{  
    I2C_DATA_IN_FLOAT;           // Set I2C DATA pin as input and floating  
    wait(2);                     // Wait two I2C clock cycles  
    I2C_CLK_IN_FLOAT;           // Set I2C CLK pin as input and floating  
    wait(2);                     // Wait two I2C clock cycles  
    while(!I2C_CLK_IN);         // Wait for I2C CLK line low (clock stretching)  
    wait(2);                     // Wait two I2C clock cycles  
    if (I2C_DATA_IN) return 1;   // Return 1 if no acknowledge received  
    else return 0;              // Return 0 if acknowledge received  
}
```

## 2.2 Control byte and Device Address

The Control byte indicates the 7-bit device address and the Read/Write indicator bit. The structure of the control byte is shown in Figure 5.

Azoteq distributor for devices with preconfigured I2C addresses. The two sub-address bits allow 4 IQS253 slave devices to be used on the same I2C bus, as well as to prevent address conflict.

The I2C device has a 7 bit Slave Address in the control byte as shown in Figure 5. To confirm the address, the software compares the received address with the device address. Please contact your local

If more than one IQS253 are on the I<sup>2</sup>C bus then sub-address bits and self/mutual settings must be preconfigured.

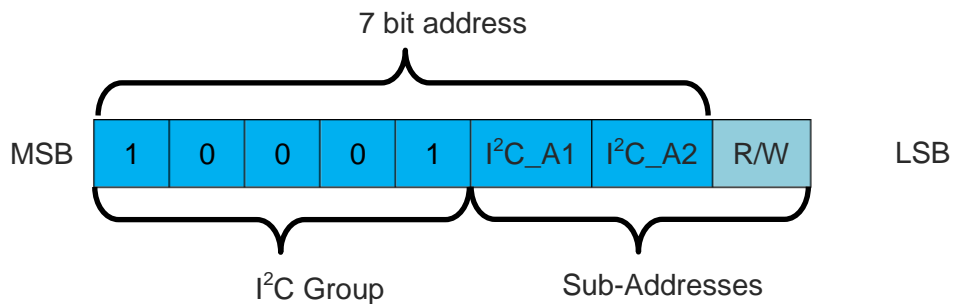


Figure 5: Control Byte Format.



### 3 IQS253 communication window

There are only two methods of entering the I<sup>2</sup>C communication window namely “Using the RDY Line” and “Acknowledge Polling”. However a word of caution: if the “Acknowledge Polling” is used and the first communication window is missed there exists no further method of invoking another communication window.

The first communication of the ProxSense® IQS253 IC after it is powered is special in the sense that there are settings that can only be adjusted within this specific communication window.

#### 3.1 Using the RDY Line

When polling is not selected the MCU can simply wait for the ready line to go low or a communication window can be invoked by a handshake. The handshake is done by setting the ready line as an output, pulling it low for 10ms and then setting it to a floating input again. The IC will respond by pulling ready low from its side if the handshake was successful. This is done until an acknowledge can be obtained.

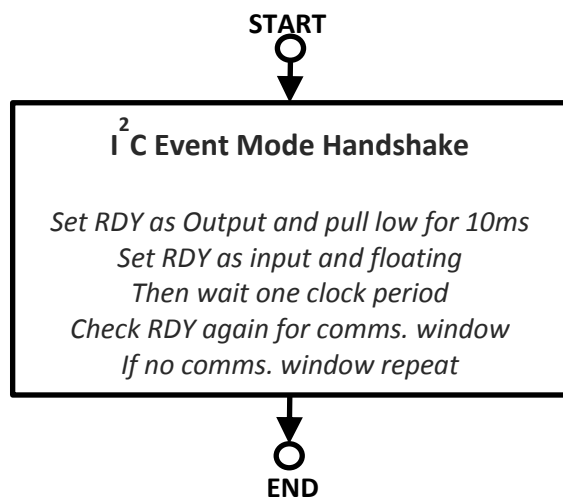


Figure 6: Flow diagram block for Event Mode Handshake.

#### Listing 4. Event Mode Handshake

```

/*
    Invokes Communication Window
*/
void i2c_event_mode_handshake(void)
{
    unsigned int i;           // Counter for stuck check

    do
    {
        I2C_RDY_OUT_PP_LOW;
        delay_ms(10);
        I2C_RDY_IN_FLOAT;
        i2c_wait();
        delay_ms(1);
        i++;
    }while(!I2C_RDY_IN || i == 15); // Test for Comms. Window
}
  
```

#### 3.2 Acknowledge Polling

If the Master device does not have an I/O available for the RDY pin, ACK polling can be used to determine when the device is ready for communication. The device will not acknowledge during a conversion cycle, this can be used to determine when a cycle is complete and whether the device has entered the communication window. Once a STOP condition is sent by the Master the device will perform the next conversion cycle. ACK polling can be initiated at any time during the conversion cycle to determine if the device has entered its communication window. The RDY pin will function normally even if it is not connected to a master device, or being used during communication.

To perform ACK polling the master sends a START condition followed by the control byte. If the device is still busy then no ACK will be returned. If the device has completed its cycle the device will return



an ACK and the master can proceed with the next read or write operation. To summarise, when polling the following procedures are executed:

1. The device master (MCU) generates a START condition.
2. The device master (MCU) sends the control byte.
3. The device master (MCU) checks if an acknowledge was received.
4. If not received the procedure is repeated from step 1.
5. The device master (MCU) reads from or writes to the IQS253.

Note that polling should only be done a fix number of times to insure that the master does not get stuck waiting for the slave. Especially in event mode it could take some time for the master to get hold of a communication window. It is also recommended to place a pull up resistor on the RDY line even though it is not used to ensure that communication windows are not randomly forced.

### 3.3 Initial Window

The initial communication window or otherwise called the 'Setup Window' gives the user an option to write start-up settings before any conversions have been done.

Most settings can be updated at any time on the IC, except switching between self and mutual capacitance technology, which can only be done in the 'Setup Window'. This enables the designer to use the device in the state that fits the operation best at a given at any given time; provided that he has control over the IC's VDDHI line. The figure below (Figure 7) shows a timing diagram that illustrates when the initial communication window occurs.

$T_{START\_UP}$  (approx.15ms) after VDDHI is set to a logic high (in this case 3.3V) the ready line will drop to a logic low for the 'Setup Window'. After addressing the IC, the required settings should be updated (Section 5) and only thereafter should a STOP bit be issued. If the 'Setup Window' is not serviced within  $t_{COMMS}$  (22ms), the ready line will go HIGH again, the IC will then start with its conversions and remain in event mode.

The IQS253 can also be requested to be preconfigured as a self or mutual capacitance device (unless application requires switching between the two) which would then not require setting up this function via I<sup>2</sup>C commands.

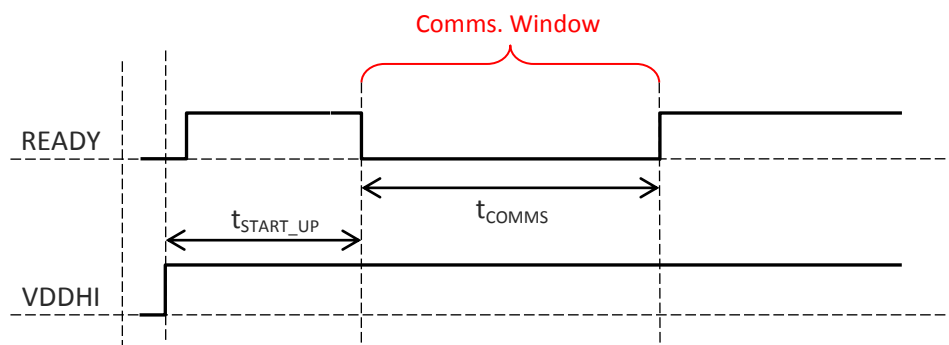


Figure 7: Timing Diagram showing initial window

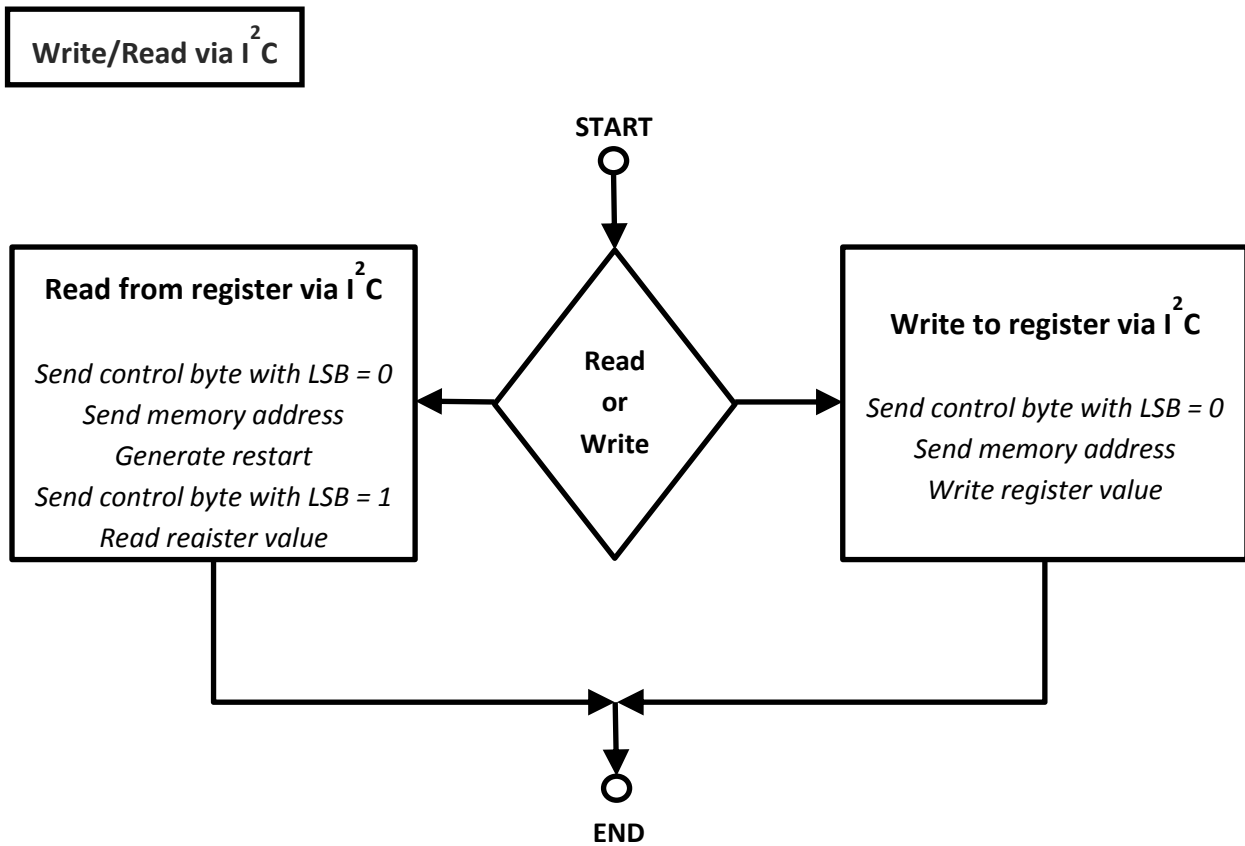




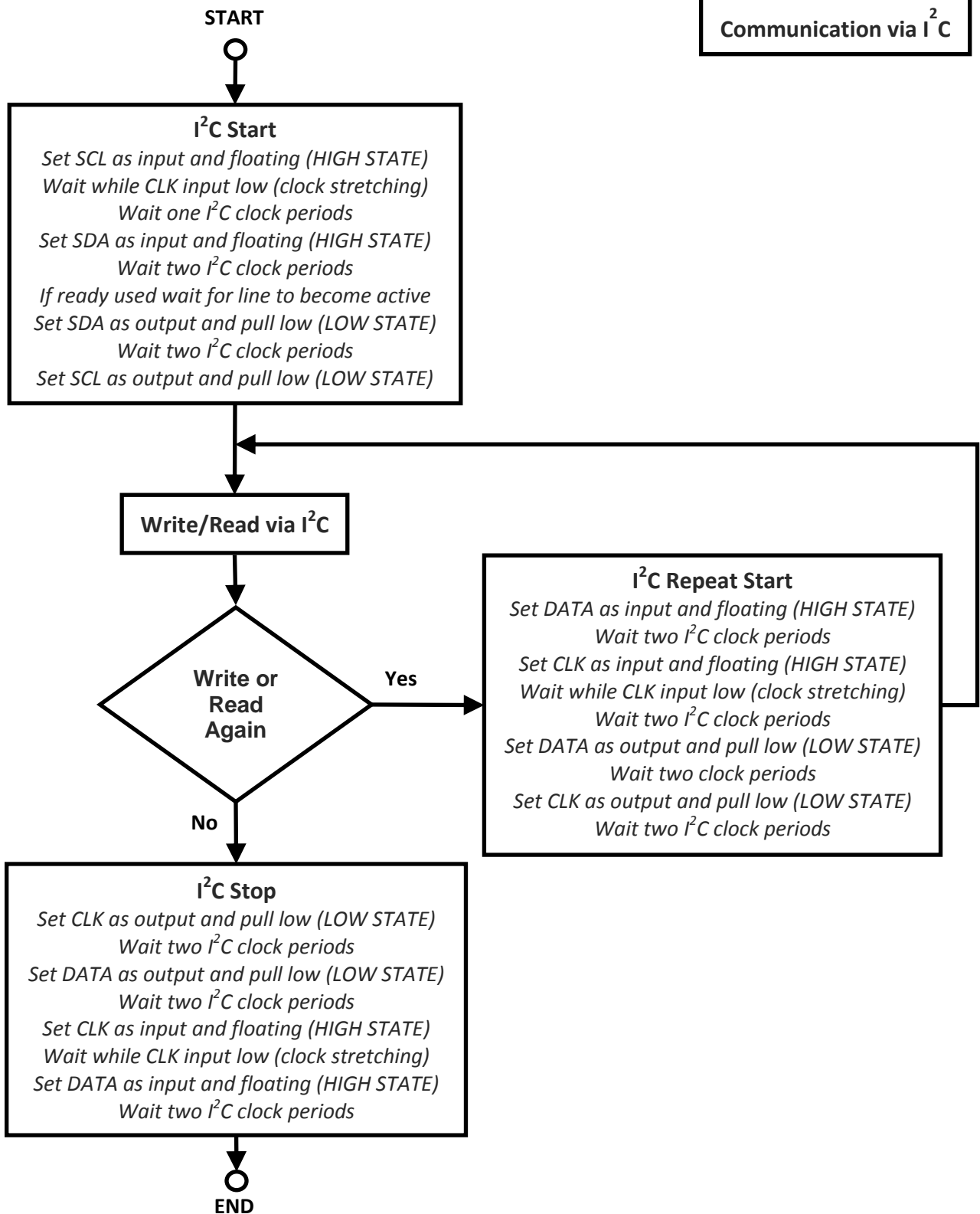
## 4 Writing to or Reading from IQS253

Once the communication window is entered and a data transfer is initiated, a write or a read operation can be executed. Write and read operations are in the format shown in Listings 5 to 8. Once the Master is finished writing/reading, the Master can then either generate another start condition (repeat-start) or it could

generate a stop condition. Another start condition will allow the Master to perform another read or write operation. A stop condition will exit the communications window and the IQS253 will continue with conversions.



**Figure 8: Flow Diagram for a Read or Write Operation**



**Figure 9: Flow Diagram for (repeated) communication**



### 4.1 Write Operation

With the R/W bit cleared in the control byte, a write is initiated. An I<sup>2</sup>C write is performed by sending the address, followed by the data. The Address is only sent once, followed by data bytes. A block of data can be written by sending the address followed by multiple blocks of data. The internal address pointer is incremented automatically for each consecutive write. If the pointer increments to an address which doesn't

exist in the memory map, no write will take place.

Note that the pointer doesn't automatically jump from the end of the LT average block to the settings block. An example of the write process is given in Figure 10.



**Figure 10: I<sup>2</sup>C Data Write**



## Listing 5. Write Operation

```
/*
    Send a given byte via I2C

    @param send_byte – byte that has to be send via I2C
    @return unsigned char – Boolean value that signifies a acknowledge returned or not
*/
unsigned char i2c_send_byte (unsigned char send_byte)
{
    unsigned char ack; // Variable to store acknowledge boolean in
    unsigned char i; // Counter variable to count off bits send

    for ( i = 0; i < 8; i++) //Send 8 bits to I2C Bus
    {
        wait(1); // Wait one I2C clock cycle
        if ( send_byte & 0x08 ) // If most significant bit equal to 1
        {
            I2C_DATA_IN_FLOAT; // Set DATA pin HIGH to clock out a 1
        }
        else
        {
            I2C_DATA_OUT_PP_LOW; // Set DATA pin LOW to clock out a 0
        }

        send_byte <<= 1; // Shift 'send_byte' left with one bit in order to send the next bit

        wait(1); // Wait one I2C clock cycle
        I2C_CLK_IN_FLOAT; // Set CLK pin HIGH
        wait(1); // Wait one I2C clock cycle
        while(!I2C_CLK_IN); // Wait for I2C CLK pin low (clock stretching)
        wait(1); // Wait one I2C clock cycle
        I2C_CLK_OUT_PP_LOW; // Set CLK pin LOW
    }

    ack = i2c_ack_check (); // Check for an acknowledge bit
    I2C_CLK_OUT_PP_LOW; // Set CLK pin LOW
    wait(1); // Wait one I2C clock cycle

    return ack; // Return acknowledge boolean
}
```



---

## Listing 6. Register Write Operation

```
/*  
    Send a given byte via I2C to specific register  
  
    Note: This function is called once already in comms window, thus start() or repeat_start() called prior calling i2c_write_register  
    After the function call the slave will still be in a comms window, waiting for either a stop or a repeat start  
  
    @param control_byte – I2C control byte  
    @param mem_address – Address or register that has to be written to  
    @param mem_value – Byte that has to be written to register  
    @return unsigned char – Boolean value that signifies a acknowledge returned or not  
*/  
unsigned char i2c_write_register(unsigned char device_address, unsigned char mem_address, unsigned char mem_value)  
{  
    unsigned char ack; // Variable to store acknowledge boolean in  
    unsigned char polling_attempt = 0; // Counter for polling attempts  
  
    ack = i2c_send_byte(device_address); // Send device address to I2C Bus  
  
    #ifdef POLLING // Include code segment if polling enabled  
    while (ack && (polling_attempt < POLLING_ATTEMPTS))  
    {  
        wait(2);  
        i2c_start();  
        ack = i2c_send_byte (device_address); // Send control byte to I2C Bus  
        polling_attempt++; // Increase polling attempts counter  
    }  
    #endif  
  
    if (!ack)  
    {  
        ack = i2c_send_byte (mem_address);  
        ack = i2c_send_byte (mem_value);  
        i2c_wait();  
    }  
    return ack;  
}
```

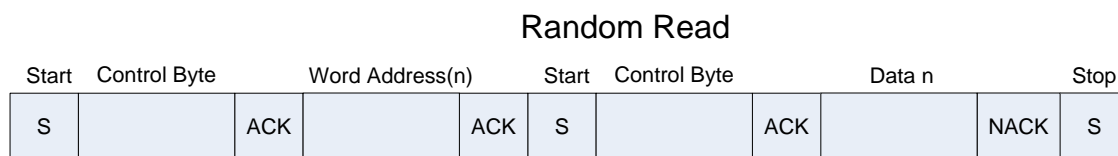


## 4.2 Read Operation

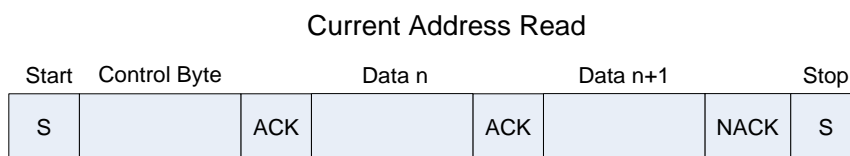
With the R/W bit SET in the control byte, a read is initiated. Data will be read from the address specified by the internal address pointer (Figure 12). This pointer will be automatically incremented (to next available address in memory map) to read through the memory map data blocks. If a random address is to be read, a Random Read must be performed. The process for a Random Read is as follows: write to the pointer (Word Address in Figure 11),

initiate a repeated-Start, read from the address.

In read mode it is the master's responsibility to acknowledge data read. The slave will send the next byte (clock stretch) if an acknowledge is give after the master has read a byte. The slave then waits for a repeat start or a stop condition from the master.



**Figure 11: I<sup>2</sup>C Random Read**



**Figure 12: I<sup>2</sup>C Current Address Read**



---

**Listing 7. Read Operation.**

```
/*
  Read byte via I2C
  @param ack if 1 send acknowledge bit else don't send acknowledge bit.
  @return byte received
*/
unsigned char i2c_read_byte(unsigned char ack)
{
    unsigned char i, receive_byte = 0;

    I2C_DATA_IN_FLOAT; // Set I2C DATA pin as input and floating
    wait(2); // Wait two I2C clock cycles

    for (i = 8; i > 0; i--) // Loop and read 8 bits from I2C DATA pin
    {
        wait(2); // Wait two I2C clock cycles
        I2C_CLK_IN_FLOAT; // Set I2C CLK pin as input and floating
        while(!I2C_CLK_IN); // Wait for I2C CLK line low (clock stretching)

        if (I2C_DATA_IN) receive_byte |= (1 << (i - 1)); //Read data from I2C DATA pin

        wait(1); // Wait one I2C clock cycle
        I2C_CLK_OUT_PP_LOW; // Set I2C CLK pin as output (push-pull) and floating
    }

    wait(1); // Wait one I2C clock cycle

    if (ack == 0) I2C_DATA_IN_FLOAT; //
    else I2C_DATA_OUT_PP_LOW; //Send acknowledge if required

    wait(2); // Wait two I2C clock cycles
    I2C_CLK_IN_FLOAT; // Set I2C CLK pin as input and floating
    while(!I2C_CLK_IN); // Wait for I2C CLK pin low (clock stretching)
    wait(2); // Wait two I2C clock cycles
    I2C_CLK_OUT_PP_LOW; // Set I2C CLK pin as output (push-pull) and floating
    wait(2); // Wait two I2C clock cycles
    I2C_DATA_IN_FLOAT; // Set I2C DATA pin as input and floating

    return receive_byte;
}
```



---

**Listing 8. Read I2C Data.**

```
/*
    Read byte from specified address via I2C

    Note: This function is called once already in comms window, thus start() or repeat_start() called prior calling i2c_write_register
    After the function call the slave will still be in a comms window, waiting for either a stop or a repeat start

    @param mem_address
    @param
    @return acknowledge status
*/
unsigned char i2c_read_register(unsigned char device_address, unsigned char mem_address, unsigned char *data_read)
{
    unsigned char temp = 0;
    unsigned char ack = 0;
    unsigned char control_byte = (device_address << 1);
    unsigned char polling_attempt = 0; //Counter for polling attempts

    ack = i2c_send_byte (control_byte); // Send device address

    #ifdef POLLING //If polling enabled
    while (ack && (polling_attempt < POLLING_ATTEMPTS) )
    {
        wait(2);
        I2Cstart();
        ack = i2c_send_byte (control_byte);
        polling_attempt++; //Increase polling attempts counter
    }
    #endif

    if (ack == 0)
    {
        i2c_send_byte (mem_address); // Write mem_address to internal pointer
        i2c_repeat_start();
        control_byte = (device_address << 1) | 0x01;
        ack = i2c_send_byte (control_byte); // Send controlbyte with r/w = 1
        temp = i2c_read_byte (1); //Read byte and don't acknowledge to indicate a repeat start or stop will follow
        (*data_read) = temp;
    }

    return ack;
}
```





## 5 Adjusting Setting for IQS253

Refer to the IQS253 memory map in its datasheet for specific addresses of registers.

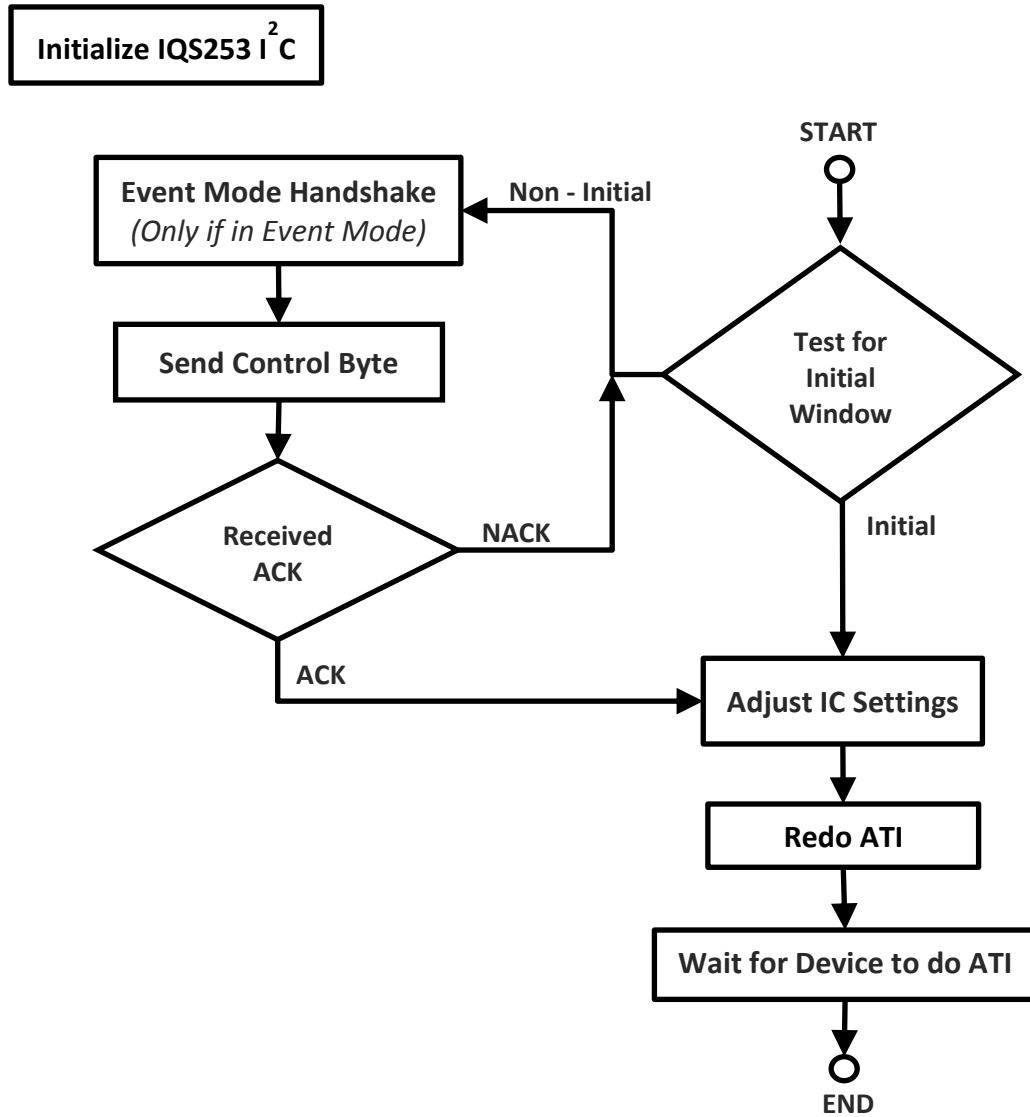


Figure 13: Initialize I2C Flow Diagram



---

**Listing 9. Adjusting Setting for IQS253**

```
/*
    Initialize IQS253
*/
void Init_IQS253(void)
{
    unsigned char result;

    do {
        i2c_event_mode_handshake();
        i2c_start();
        result = i2c_write_register(PROX_SETTINGS2, 0x14);           //set to streaming mode for setup
        i2c_stop();
    } while(result);

    do {
        i2c_start();
        i2c_write_register(CHAN_ACTIVE, 0x07);                     // Enable Channels (0-2)
        i2c_repeat_start();
        i2c_write_register(PROX_SETTINGS0, 0x40);                 // ATI OFF, ATI partial OFF
        i2c_repeat_start();
        i2c_write_register(PROX_SETTINGS2, 0x40);                 // WDT Off
        i2c_stop();


        /*Set Touch Thresholds*/
        i2c_start();
        result |= i2c_write_register(CH0_TTH, 0x04);
        i2c_repeat_start();
        result |= i2c_write_register(CH1_TTH, 0x20);
        i2c_repeat_start();
        result |= i2c_write_register(CH2_TTH, 0x20);
        /* Set Proximity Thresholds */
        i2c_repeat_start();
        result |= i2c_write_register(CH0_PTH, 0x04);
        i2c_repeat_start();
        result |= i2c_write_register(CH1_PTH, 0x04);
        i2c_repeat_start();
        result |= i2c_write_register(CH2_PTH, 0x04);
        i2c_repeat_start();
        i2c_write_register(TARGET, 0x40);                         //Set Target Current Count = 512
        i2c_repeat_start();
        result |= i2c_write_register(CHAN_ENABLE, 0x07);         //Disable distributed PROX CH0
        i2c_stop();
    }while (result);

    delay_ms(200);

    do {
        delay_ms(10);
        i2c_start();
        result = i2c_read_register(STATUS, 1);
        i2c_stop();
    } while ( (result & 0x04) != 0 );
}
```



The following patents relate to the device or usage of the device: US 6,249,089 B1, US 6,621,225 B2, US 6,650,066 B2, US 6,952,084 B2, US 6,984,900 B1, US 7,084,526 B2, US 7,084,531 B2, US 7,119,459 B2, US 7,265,494 B2, US 7,291,940 B2, US 7,329,970 B2, US 7,336,037 B2, US 7,443,101 B2, US 7,466,040 B2, US 7,498,749 B2, US 7,528,508 B2, US 7,755,219 B2, US 7,772,781, US 7,781,980 B2, US 7,915,765 B2, EP 1 120 018 B1, EP 1 206 168 B1, EP 1 308 913 B1, EP 1 530 178 B1, ZL 99 8 14357.X, AUS 761094

IQ Switch<sup>®</sup>, ProxSense<sup>®</sup>, LightSense<sup>™</sup>, AirButton<sup>®</sup> and the  logo are trademarks of Azoteq.

The information in this Datasheet is believed to be accurate at the time of publication. Azoteq assumes no liability arising from the use of the information or the product. The applications mentioned herein are used solely for the purpose of illustration and Azoteq makes no warranty or representation that such applications will be suitable without further modification, nor recommends the use of its products for application that may present a risk to human life due to malfunction or otherwise. Azoteq products are not authorized for use as critical components in life support devices or systems. No licenses to patents are granted, implicitly or otherwise, under any intellectual property rights. Azoteq reserves the right to alter its products without prior notification. For the most up-to-date information, please refer to [www.azoteq.com](http://www.azoteq.com).

[WWW.AZOTEQ.COM](http://WWW.AZOTEQ.COM)

[ProxSenseSupport@azoteq.com](mailto:ProxSenseSupport@azoteq.com)