



AZD072 IQS333 Communication and Interface Guideline

1 Introduction

This application note is designed to guide the reader through the process of setting up the communication interface between the ProxSense[®] IQS333 IC and any MCU capable of I²C (400kBit/s) communication.

This is done through flow diagrams as well as providing the source code in listings throughout the document.

In Figure 1.1 below an overview flow diagram is shown to provide the reader

with an overview of what is discussed within this document.

A Microchip PIC[®] was used in this document, thus the code that is relevant is listed in this document. The complete source code is available from http://www.azoteq.com/images/stories/software/iqs333_example_code.zip.

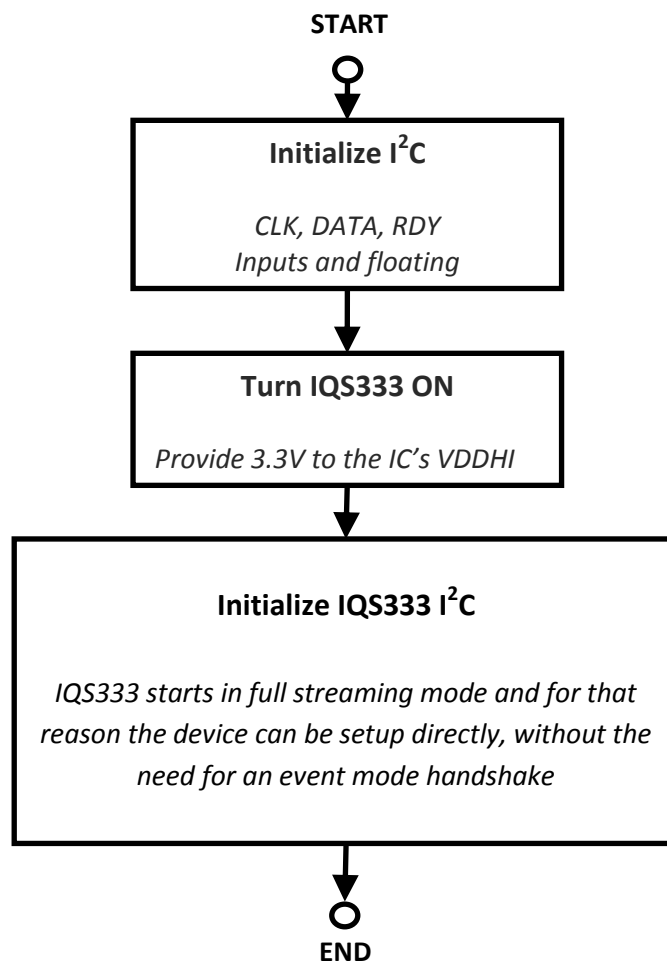


Figure 1.1 Initialize I²C Flow Diagram.



2 Communication Protocol

The IQS333 uses a bi-directional 3-wire (SDA, SCL, and RDY) serial interface bus. The serial protocol is I²C™ compatible. The IQS333 has a ready (RDY) pin which indicates when the device enters its communication window period. Communication with the device can only take place in this state and can be determined by monitoring the RDY line or by using ACK polling, which is discouraged. The IQS333 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 for a selectable duration of always (Timeout Disabled) or approximately 14ms (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, with the data being lost. The RDY line will remain low for the duration of the communication window period, or until a stop command is sent.

In Figure 2.1 the data transfer sequence for the communication protocol is shown as an overview of what is explained within this section.

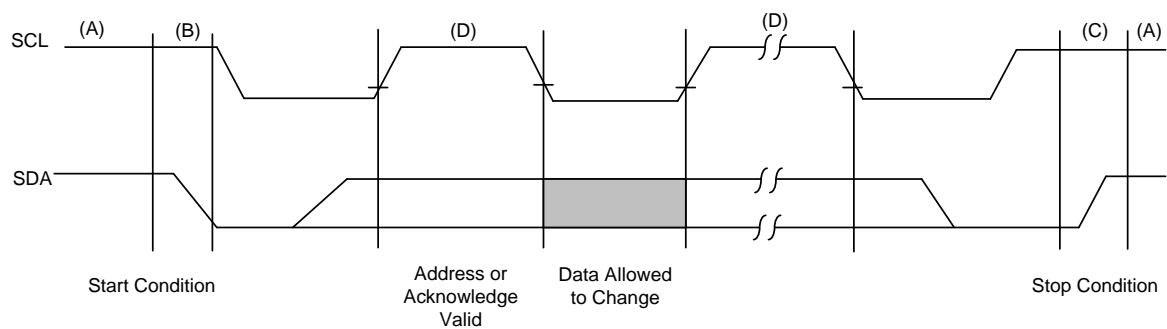


Figure 2.1 Data Transfer Sequence on the Serial Bus.

3 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 Figure 2.1)

- **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, which is shown in Figure 3.1.
- **REPEAT-START Condition** - A HIGH to LOW transition of the SDA while the SCL is HIGH. This is used for sending multiple bytes to the I2C device, without terminating the communication window, like in a register read operation. An example of a repeat-start is shown in Figure 3.3
- **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 and is shown in Figure 3.2. NOTE: When a STOP condition is sent the device will exit the communications window and continue with conversions.
- **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 IQS333 does not generate any acknowledge bits while it is not in its communication window.

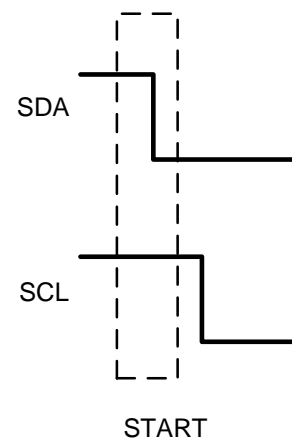


Figure 3.1 Start Condition.

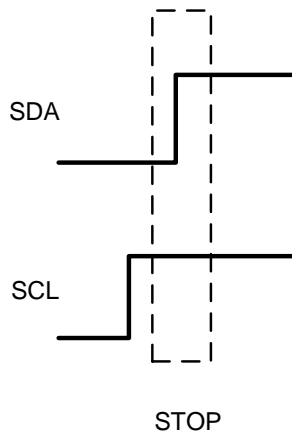


Figure 3.2 Stop Condition.

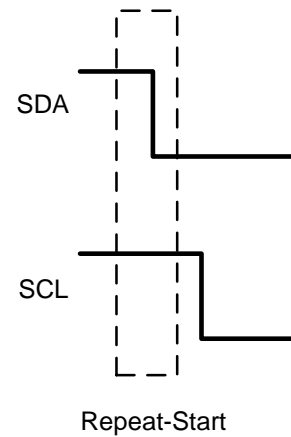


Figure 3.3 Repeat-Start Condition.

Listing 1: START Condition

```
/*
Function name:      CommsIQS_start
Parameters:        none
Return:            none
Description:        A start condition is created on the I2C. Use a
                    timeout sequence, to ensure that the i2c does not time out
*/
unsigned char CommsIQS_start(void)
{
    unsigned int timeout = 1000;                // create a timeout for communication

    while (PORTBbits.RB0 == 1 && timeout > 0)    // wait for ready - pin RB0 active low
    {
        timeout--;
    }

    SSP1CON2bits.SEN = 1;                      // start condition

    timeout = 1000;                            // Reload timer
    while ((PIR1bits.SSP1IF == 0) && (timeout > 0)) //wait for start condition to be generated
    {
        timeout--;
    }

    if (timeout == 0)
    {
        if ((SSP1CON2 & 0x08) == 0x08)
        {
            //re-init I2C
            SSP1CON1 = 0x00;
            SSP1CON2 = 0x00;
            SSP1CON1 = 0x28;
        }
        return IC_ERROR_SER_TIMEOUT;
    }

    PIR1bits.SSP1IF = 0;                       // clear flag

    timeout = 1000;                            // Reload timer
    while (SSP1CON2bits.SEN == 1 && timeout > 0); // verify start is complete
    {
        timeout--;
    }

    return RETURN_OK;                          // start was generated
}
```



Listing 2: STOP Condition

```
/*  
Function name:      CommsIQS_stop  
Parameters:        none  
Return:            none  
Description:       A stop condition is created on the I2C.  
                   A timeout is incorporated to insure a stuck condition  
                   does not occur on the i2c bus, holding the MCU stuck  
*/  
void CommsIQS_stop(void)  
{  
    unsigned int timeout = 1000; // create a timeout for communication  
  
    SSP1CON2bits.PEN = 1; // stop condition  
  
    while (PIR1bits.SSP1IF == 0) // wait for stop condition to be generated  
    {  
        timeout--;  
    }  
  
    PIR1bits.SSP1IF = 0; // clear flag  
  
    timeout = 1000; // Reload timer  
    while (SSP1CON2bits.PEN == 1)  
    {  
        timeout--;  
    }  
  
    delay_ms(1);  
    //wait for the IQS device to become ready  
    while(PORTBbits.RB0 == 0) // wait for the IQS to change from ready to not ready  
    {}  
    while(PORTBbits.RB0 == 1) // wait for the IQS to change from not ready to ready  
    {}  
}
```



Listing 3: Check for Acknowledge

```
/*
Function name: CommsIQS_read_ack
Parameters:   none
Return:      unsigned char - the data received via the I2C
Description: Enables the Master Receive Mode of the I2C module on the PIC16LF1827. The data received is
              returned and a ACK acknowledge is sent to the IQS259 to indicate that another read command
will follow this one.
              A timeout is incorporated to insure a stuck condition
              does not occur on the i2c bus, holding the MCU stuck
*/
unsigned char CommsIQS_read_ack(void)
{
    unsigned char temp;
    unsigned int timeout = 1000;           // create a timeout for communication

    SSP1CON2bits.RCEN = 1;                // enable master receiver mode
    while (PIR1bits.SSP1IF == 0 && timeout > 0) // wait for byte received flag
    {
        timeout--;
    }

    PIR1bits.SSP1IF = 0;                  //clear flag

    timeout = 1000;                        // Reload timer
    while (SSP1STATbits.BF == 0 && timeout > 0) // wait for buffer full flag (receive complete)
    {
        timeout--;
    }

    temp = SSP1BUF;                        // store received byte

    SSP1CON2bits.ACKDT = 0;                // enable ACK
    SSP1CON2bits.ACKEN = 1;                // execute ACK sequence

    timeout = 1000;                        // Reload timer
    while (PIR1bits.SSP1IF == 0 && timeout > 0) // Wait for ACK transmission complete
    {
        timeout--;
    }

    PIR1bits.SSP1IF = 0;                  // clear flag

    timeout = 1000;                        // Reload timer
    while (SSP1CON2bits.ACKEN == 1 && timeout > 0) // verify acknowledge sequence is complete
    {
        timeout--;
    }

    return temp;                           // return received data
}
```



4 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 4.1.

The I²C device has a 7 bit Slave Address (default 64H) in the control byte as shown in Figure 4.1. The control bytes for Read and Write with default sub-addresses are as follows:

- Write to the IQS333 – 0xC8
- Read from the IQS333 – 0xC9

To confirm the address, the software compares the received address with the device address. Please contact your local

Azoteq distributor for devices with preconfigured I²C addresses. The two sub-address bits allow 4 IQS333 slave devices to be used on the same I²C bus, as well as to prevent address conflict.

If more than one IQS333 are on the I²C bus then sub-address bits must be preconfigured.

Figure 4.2 shows an example of a Write Control byte (C8H) for the default I²C address of the IQS333, followed by an acknowledge.

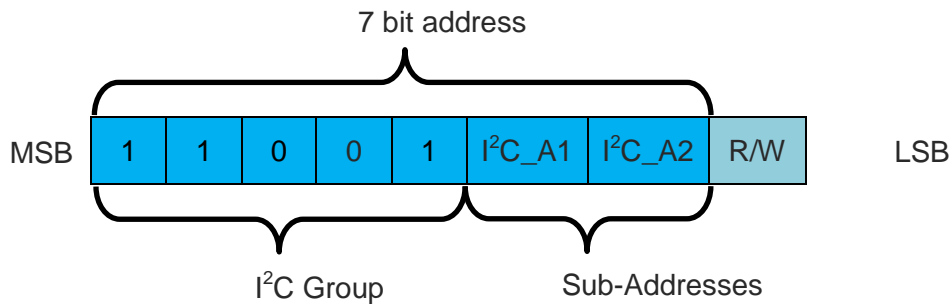


Figure 4.1 Control Byte Format.

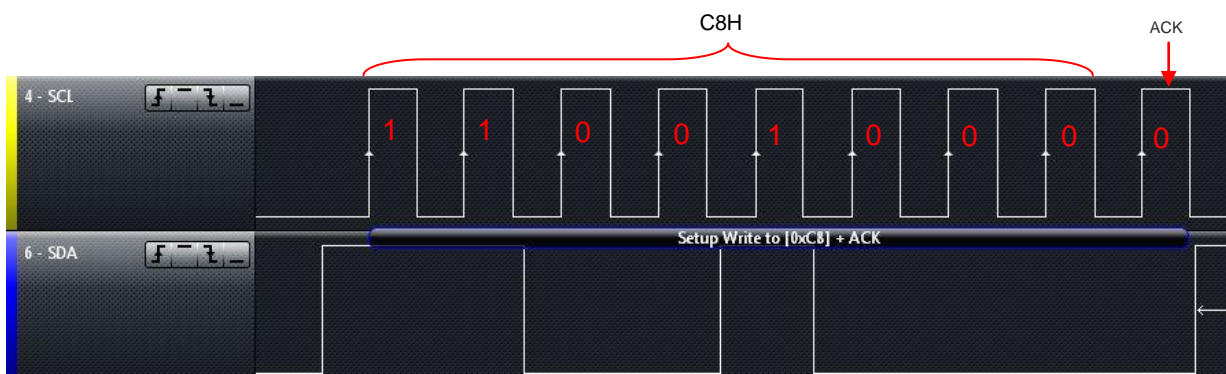


Figure 4.2 Example of the Control Byte Format for a Write Operation with the Default Device Address

5 IQS333 communication window

There are only two methods of entering the I²C communication window namely “Using the RDY Line” and “Acknowledge Polling”. However, “Acknowledge Polling” is discouraged, as it reduces speed and could introduce noise on communication lines during conversions.

5.1 Using the RDY Line

The MCU can simply wait for the ready line to go low know when a communication window is open. A communication window can be invoked by a handshake during event mode. 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. The process is shown in Figure 5.1.

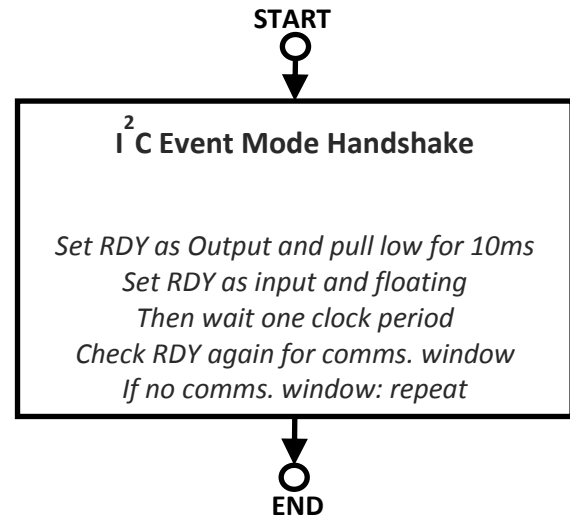


Figure 5.1 Flow diagram block for Event Mode Handshake.

Listing 4: Event Mode Handshake

```
/*
Function name: CommsIQS_event_mode_handshake
Parameters:   none
Return:       none
Description:  This function establishes a connection with
              the IQS333, while the IQS333 is in event mode.
              See description of event mode in datasheet of
              the IQS333. A timeout is incorporated to
              insure a stuck condition does not occur on the
              i2c bus, holding the MCU stuck
*/
void CommsIQS_event_mode_handshake(void)
{
    TRISBbits.TRISB0 = 0;    // RDY output
    PORTBbits.RB0 = 0;      // Force a communication window
    // create a delay
    delay_ms(10);
    // reset the RDY line as an input to indicate if communication was established
    TRISBbits.TRISB0 = 1;    // RDY input
    delay_ms(2);
}
```




5.2 Initial Window

The initial communication window is the first communication window after start-up of the IQS333.

Settings can be updated at any time on the IC and does not have to happen in the first communication window. Figure 5.2 shows a timing diagram that illustrates when the initial communication window occurs.

T_{START_UP} (approx.10ms) after VDDHI is set to logic high (in this case 3.3V) the ready line will drop to a logic low for the

initial communication window. After addressing the IC, the required settings should be updated (Section 6.4) and only thereafter should a STOP bit be issued. The initial communication window remains open for approximately t_{COMMS} (10ms), after which the ready line will go HIGH again, the IC will then start with its conversions.

After the conversions of all channels are finished, another communication window is given.

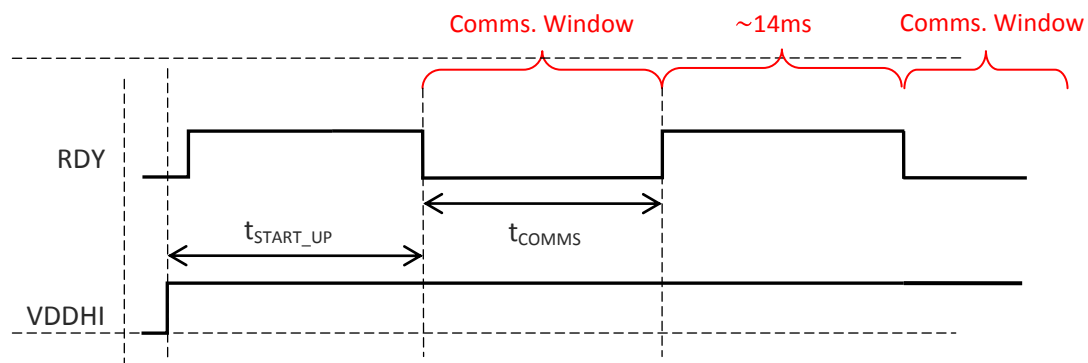


Figure 5.2 Timing Diagram showing initial window

5.3 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 IQS333.

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.

6 Writing to or Reading from IQS333

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. A repeat-start condition will allow the Master to perform another read or write operation, without having to wait for a conversion cycle to finish. A stop condition will exit the

communications window and the IQS333 will continue with conversions.

The IQS333 uses a command/address structure, which means that multiple bytes are sent to or read from the same register address. For example, if Proxsettings2 needs to be changed, 3 bytes need to be sent to the IQS333. Refer to Figure 6.1 for a flow diagram on the read/write operation of the IQS333. Figure 6.2 show the sequence for multiple read/writes.

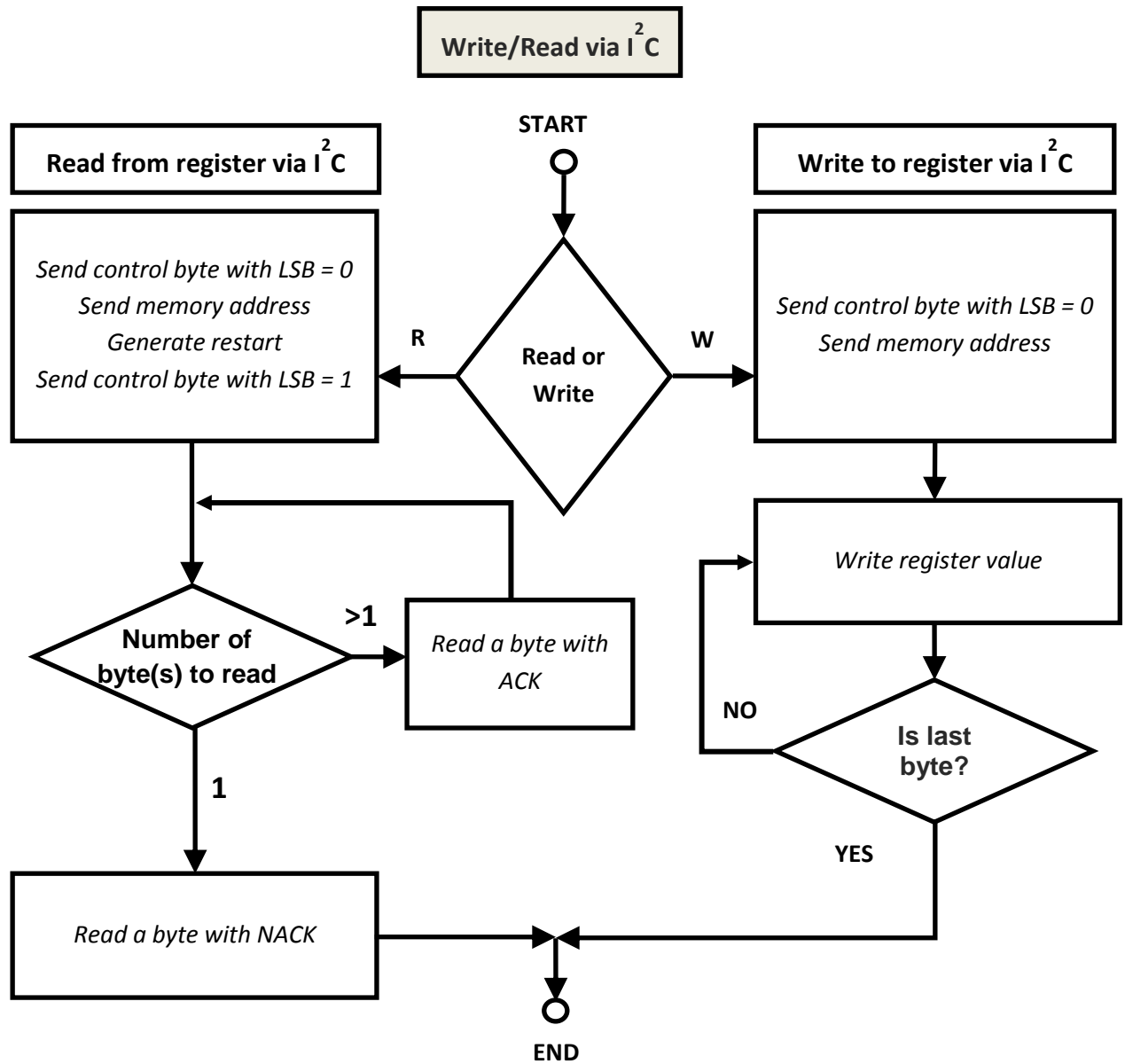


Figure 6.1 Flow Diagram for a Read or Write Operation

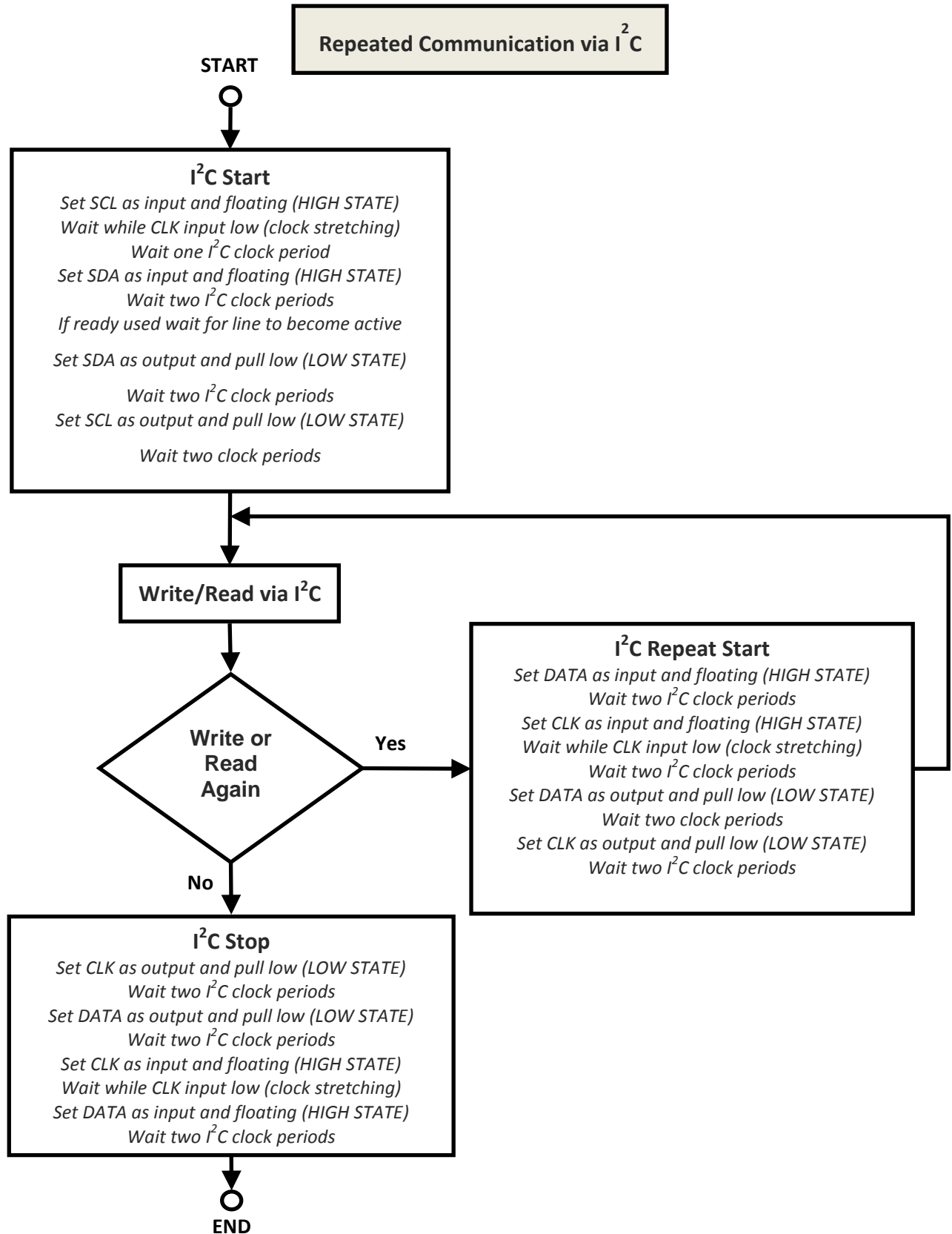


Figure 6.2 Flow Diagram for (repeated) communication



6.2 Write Operation

With the R/W bit cleared in the control byte, a write is initiated. An I²C 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. No write will take place if data is written to a register that does not exist. Note that the pointer doesn't automatically jump from the end of, for example the wheel coordinates block to the touch bytes, therefore a new address must be

sent to write (or read) to a new register. An example of the write process is given in Figure 6.3.

An example of the write operation timing diagram can be seen in Figure 6.4, which shows the start sequence, the write control byte (C8H), address to write to, data bytes and stop command on the I²C Clock line as well as the Data line. In this example the ATI Targets register is setup with byte 1 (80H) and byte 2 (40H) – 1024 counts for CH0 and 512 counts for the active channels.

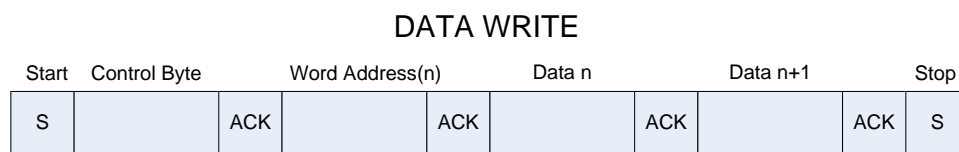


Figure 6.3 I²C Data Write

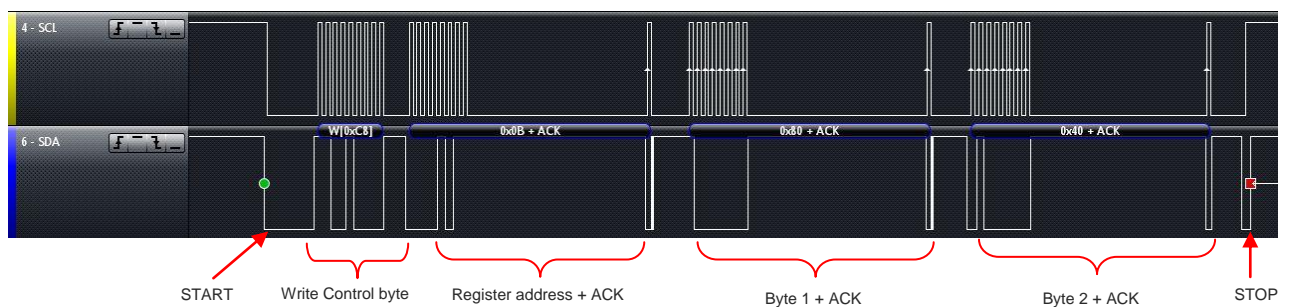


Figure 6.4 Example of a Register Write Operation setting up the ATI Targets Register



Listing 5: Register Write Operation

```
/*
Description:      --- CommsIQS Write() ---
                  Writes data to IQS device

Input:           write_addr - This is the IQS ADDRESS-COMMAND
                  data - Pointer to an array of chars, which contain the data to write to
                  the IQS
                  NoOfBytes - The number of bytes to write to the IQS

Limitations:     This function must be preceded by one of the following function calls:
                  CommsIQS_start(), CommsIQS_repeat_start(),

                  This function must be followed by one of the following function calls:
                  CommsIQS_repeat_start(), CommsIQS_stop

*/
void CommsIQS_Write(unsigned char i2c_addr, unsigned char write_addr,
                   unsigned char *data, unsigned char NoOfBytes)
{
    unsigned char i;

    CommsIQS_send((i2c_addr << 1) + 0x00);    // device address + write
    CommsIQS_send(write_addr);               // IQS address-command
    for (i = 0 ; i < NoOfBytes ; i++)        // Send more than one byte if necessary
        CommsIQS_send(data[i]);
}
```

Listing 6: Write Operation

```
/*
Function name:   CommsIQS333 send
Parameters:     unsigned char send_data - data to be sent transmitted via the I2C
Return:         none
Description:     Transmits the data byte given as parameter via the I2C of the PIC16LF1827
                  Note that the I2C communication channel must already be active before calling
this
                  function, as no start bits are included in this function.
                  A timeout is incorporated to insure a stuck condition
                  does not occur on the i2c bus, holding the MCU stuck

*/
void CommsIQS_send(unsigned char send_data)
{
    unsigned int timeout = 1000;              // create a timeout for communication

    SSP1BUF = send_data;                     // write transmit byte to buffer

    timeout = 1000;                           // Reload timer
    while (PIR1bits.SSP1IF == 0 && timeout > 0) // wait for transmit complete flag
    {
        timeout--;
    }

    PIR1bits.SSP1IF = 0;                      // clear flag

    while (SSP1CON2bits.ACKSTAT == 1 && timeout > 0) // verify IQS333 acknowledge
    {
        timeout--;
    }
}
```



6.3 Read Operation

With the R/W bit SET in the control byte, a read is initiated. The process to read a register is as follows: write to the pointer (Word Address in Figure 6.5), 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 given after the master has read a byte. The slave then waits for a repeat start or a stop condition from the master.

Figure 6.6 shows an example of a register read operation, where the Touch bytes

register (register 03H) is read with 2 bytes of data. From Figure 6.6 the composition of the register read operation can be seen. First a start is sent to the I²C, followed by a write control byte (C8H) then the register to read from (register 03H in this case), which is then followed by another start (a repeat-start condition) and then the read control byte (C9H) followed by the data bytes (2 bytes is read in this example), with the last byte read with a NACK and then a stop follows, indicating the end of communication.

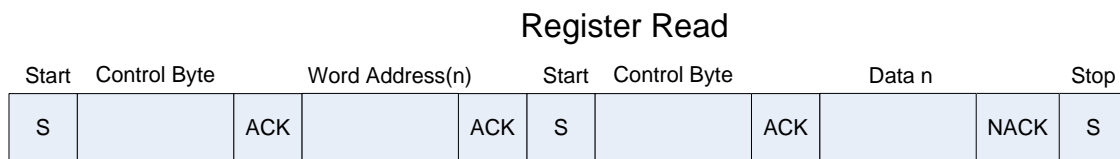


Figure 6.5 I²C Register Read

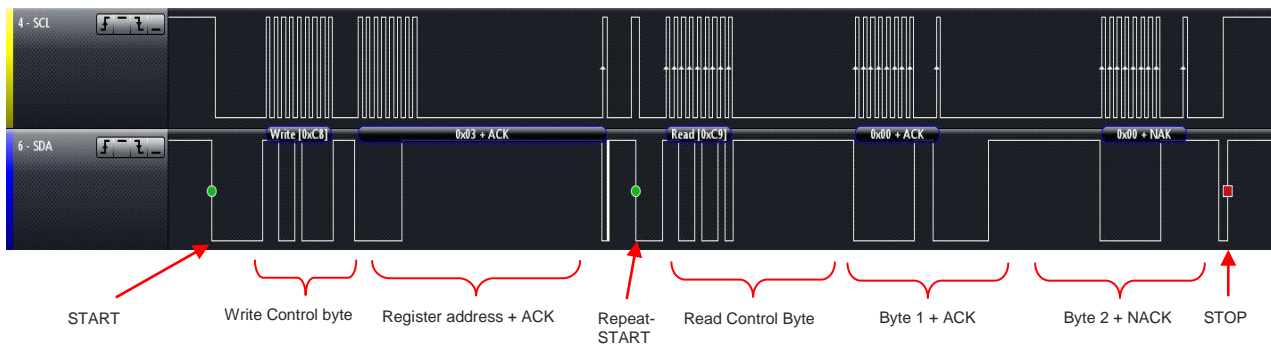


Figure 6.6 Example of a Register Read Operation reading 2 Bytes from the Touch bytes Register



Listing 7: Read I2C data

```
/*      ---- CommsIQS Read() ----
Description: Reads data from the IQS

Input:     read_addr - This is the IQS ADDRESS-COMMAND
           data - Pointer to an array of chars, where the data READ from the IQS is stored
           NoOfBytes - The number of bytes to read from the IQS333

Limitations: This function must be preceded by one of the following function calls:
              CommsIQS_start(), CommsIQS_repeat_start(),

              This function must be followed by one of the following function calls:
              CommsIQS_repeat_start(), CommsIQS_stop
*/
void CommsIQS_Read(unsigned char i2c_addr, unsigned char read_addr, unsigned char *data, unsigned char
NoOfBytes)
{
    unsigned char i;

    CommsIQS_send((i2c_addr << 1) + 0x00); // device address + write
    CommsIQS_send(read_addr); // IQS address-command
    CommsIQS_repeat_start();
    CommsIQS_send((i2c_addr << 1) + 0x01); // device address + read
    if (NoOfBytes > 1)
    {
        for (i = 0; i < NoOfBytes - 1; i++)
            data[i] = CommsIQS_read_ack(); // all bytes except last must be followed by an ACK
    }
    data[NoOfBytes-1] = CommsIQS_read_nack(); // last byte read must be followed by a NACK
}
```




Listing 8: Read byte with an ACK

```
/*
Function name: CommsIQS_read_ack
Parameters:   none
Return:      unsigned char - the data received via the I2C
Description:  Enables the Master Receive Mode of the I2C module on the PIC16LF1827. The data received is
              returned and a ACK acknowledge is sent to the IQS259 to indicate that another read command
              will follow this one.
              A timeout is incorporated to insure a stuck condition
              does not occur on the i2c bus, holding the MCU stuck
*/
unsigned char CommsIQS_read_ack(void)
{
    unsigned char temp;
    unsigned int timeout = 1000;           // create a timeout for communication

    SSP1CON2bits.RCEN = 1;                // enable master receiver mode
    while (PIR1bits.SSP1IF == 0 && timeout > 0) // wait for byte received flag
    {
        timeout--;
    }

    PIR1bits.SSP1IF = 0;                  //clear flag

    timeout = 1000;                       // Reload timer
    while (SSP1STATbits.BF == 0 && timeout > 0) // wait for buffer full flag (receive complete)
    {
        timeout--;
    }

    temp = SSP1BUF;                       // store received byte

    SSP1CON2bits.ACKDT = 0;               // enable ACK
    SSP1CON2bits.ACKEN = 1;               // execute ACK sequence

    timeout = 1000;                       // Reload timer
    while (PIR1bits.SSP1IF == 0 && timeout > 0) // Wait for ACK transmission complete
    {
        timeout--;
    }

    PIR1bits.SSP1IF = 0;                  // clear flag

    timeout = 1000;                       // Reload timer
    while (SSP1CON2bits.ACKEN == 1 && timeout > 0) // verify acknowledge sequence is complete
    {
        timeout--;
    }

    return temp;                          // return received data
}
```



Listing 9: Read byte with a NACK

```
/*
Function name: CommsIQS read nack
Parameters:   none
Return:      unsigned char - the data received via the I2C
Description:  Enables the Master Receive Mode of the I2C module on the PIC16LF1827. The data received is
              returned and a NACK acknowledge is sent to the IQS259 to indicate that this was the final
              read of the current continuous read block.
              A stop or repeated start command has be called next.
              A timeout is incorporated to insure a stuck condition
              does not occur on the i2c bus, holding the MCU stuck
*/
unsigned char CommsIQS_read_nack(void)
{
    unsigned char temp;
    unsigned int timeout = 1000;                // create a timeout for communication

    SSP1CON2bits.RCEN = 1;                     // enable master receiver mode
    while (PIR1bits.SSP1IF == 0 && timeout > 0) // wait for byte received flag
    {
        timeout--;
    }

    PIR1bits.SSP1IF = 0;                       // clear flag

    timeout = 1000;                             // Reload timer
    while (SSP1STATbits.BF == 0 && timeout > 0) // wait for buffer full flag (receive complete)
    {
        timeout--;
    }

    temp = SSP1BUF;                             // store received byte

    SSP1CON2bits.ACKDT = 1;                     // enable NACK
    SSP1CON2bits.ACKEN = 1;                   // execute NACK sequence

    timeout = 1000;                             // Reload timer
    while (PIR1bits.SSP1IF == 0 && timeout > 0) // Wait for NACK transmission complete
    {
        timeout--;
    }

    PIR1bits.SSP1IF = 0;                       // clear flag

    timeout = 1000;                             // Reload timer
    while (SSP1CON2bits.ACKEN == 1 && timeout > 0) // verify acknowledge sequence is complete
    {
        timeout--;
    }

    return temp;                               // return data
}
```



6.4 Adjusting Settings for IQS333

Refer to the IQS333 Command/Address Structure in its datasheet for specific addresses of registers. Setup for the IQS333 can be done anytime that a communication window is available, which means the RDY is pulled low by the IQS333. The setup for the IQS333 is done by sending an array of values to the required register. This is done for all the registers required for setup.

Figure 6.7 shows a flow diagram of the initialisation of the IQS333.

Listing 10 shows an example with only a few settings, on how to setup the IQS333.

From within the GUI of the IQS333 (available on the web at

<http://www.azoteq.com/proximity-switches-design/capacitance-sensor-software-and-tools.html>) it is possible to export an *.h file, IQS333_init.h, which contains all of the settings of the IQS333 as setup in the GUI. This file can then be pasted into the working folder of the example code, or user code, in order to setup the device as it was setup in the GUI. This enables quick setup of the IQS333 for specific devices. Figure 6.3 shows how to export the *.h file from the GUI. Note that this is only available on version 1.0.1.40 or later.

Replace the IQS333_init.h file in the example project (or user project) folder to load the new settings.

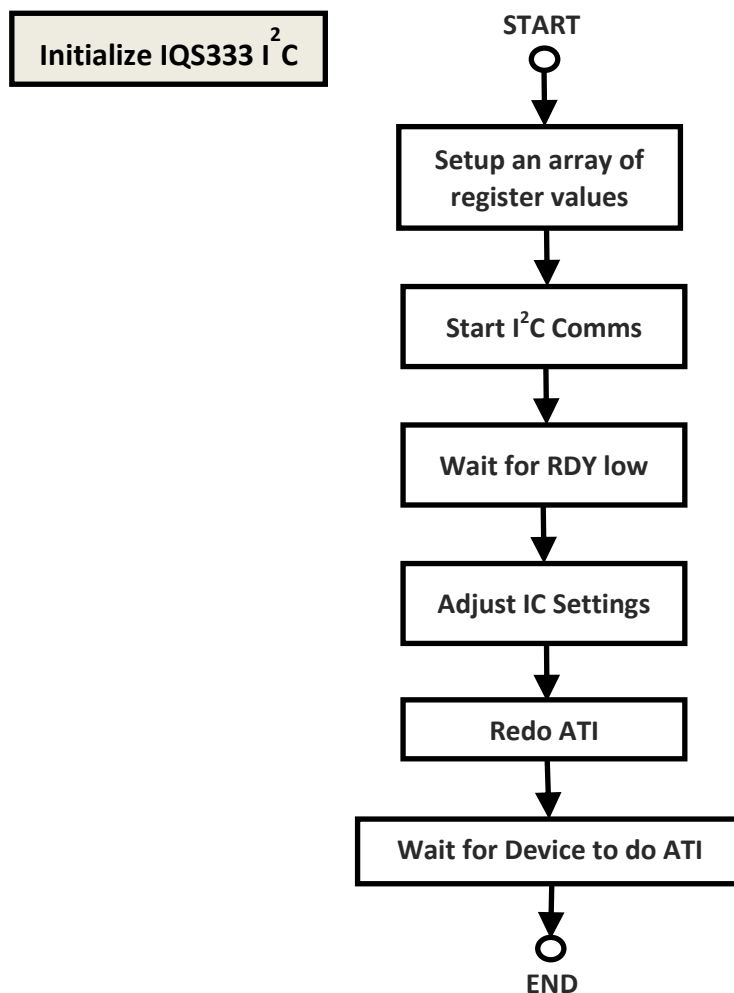


Figure 6.2 Initialize I2C Flow Diagram

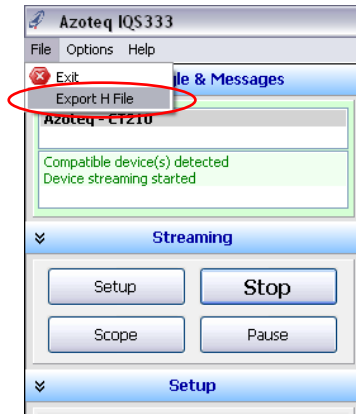


Figure 6.3 Export settings from the IQS333 GUI to the Example code



Listing 10: Adjusting IQS333 settings

```
/**
 * This function sets up the IQS333 for optimum performance in this device -
 * these values can however be overwritten from the GUI
 * Params: None
 */
void IQS333_Button_Wheel_Settings(void)
{
    unsigned char data_buffer[12];

    lights_on();

    /* Read product number */
    CommsIQS_start();
    CommsIQS_Read(IQS333_ADDR, VERSION_INFO, &data_buffer[0], 2);
    CommsIQS_stop();

    /* Switch the IQS333 to projection mode */
    data_buffer[0] = SYSTEM_FLAGS_VAL;
    CommsIQS_start();
    CommsIQS_Write(IQS333_ADDR, FLAGS, &data_buffer[0], 1);
    CommsIQS_stop();

    // set active channels on iqs333 CH0 CH1 CH2 CH4 CH5 CH
    /* Set active channels */
    data_buffer[0] = ACTIVE_CH0;
    data_buffer[1] = ACTIVE_CH1;
    CommsIQS_start();
    CommsIQS_Write(IQS333_ADDR, ACTIVE_CHANNELS, &data_buffer[0], 2);
    CommsIQS_stop();

    /* Setup Touch and Prox thresholds for each channel */
    data_buffer[0] = PROX_THRESHOLD;
    data_buffer[1] = TOUCH_THRESHOLD_CH1;
    data_buffer[2] = TOUCH_THRESHOLD_CH2;
    data_buffer[3] = TOUCH_THRESHOLD_CH3;
    data_buffer[4] = TOUCH_THRESHOLD_CH4;
    data_buffer[5] = TOUCH_THRESHOLD_CH5;
    data_buffer[6] = TOUCH_THRESHOLD_CH6;
    data_buffer[7] = TOUCH_THRESHOLD_CH7;
    data_buffer[8] = TOUCH_THRESHOLD_CH8;
    data_buffer[9] = TOUCH_THRESHOLD_CH9;
    CommsIQS_start();
    CommsIQS_Write(IQS333_ADDR, THRESHOLDS, &data_buffer[0], 10);
    CommsIQS_stop();

    /* Set the ATI Targets (Target Counts) */
    data_buffer[0] = ATI_TARGET_CH0;
    data_buffer[1] = ATI_TARGET_CH0_9;
    CommsIQS_start();
    CommsIQS_Write(IQS333_ADDR, ATI_TARGETS, &data_buffer[0], 2);
    CommsIQS_stop();

    /* wait until the ATI algorithm is done */
    do
    {
        delay_ms(15);
        CommsIQS_start();
        CommsIQS_Read(IQS333_ADDR, FLAGS, &data_buffer[0], 1);
        CommsIQS_stop();
    }
    while ((data_buffer[0] & 0b00000100) == 0b00000100);

    // read the error bit to determine if an ATI error occurred
    CommsIQS_start();
    CommsIQS_Read(IQS333_ADDR, PROXSETTINGS, &data_buffer[0], 2);
    CommsIQS_stop();

    // if an ATI error occurred, switch on buzzer
    if (data_buffer[1] & 0x02 == 0x02)
    {
        buzzer(5);
    }

    lights_off();
} // End_IQS333_Button_Wheel_Settings()
```



7 Timing Diagrams

A few important timing diagrams of the IQS333 are shown in this section. The first communication window is shown in Figure 7.1, which occurs approximately 10ms after the IQS333 is switched on (VDDHI = 1). This window occurs when the RDY line is pulled low by the IQS333 for the first time. Only at this stage can communication take place between the MCU and the IQS333. Figure 7.2 shows the time between two adjacent communication windows, which happens each time that the IQS333 has finished with conversions. The time between the two communication windows is approximately 14ms, with 7 channels activated. Note that this time is subject to the active channels, as well as other settings.

The time between the stop command of the I²C communication and the RDY line going high is approximately 55 μ s and is shown in Figure 7.3. Also shown in Figure 7.3 is the time from the RDY line going high to the charging of channel 1, which is approximately 0.3ms. Avoid sending a Start before the current communications window has closed.

All of the timing diagrams, including the viewer software, can be downloaded from <http://www.azoteq.com/proximity-sensors-products/proxsense-products/121-products-proxsense-iqs333.html>.



Figure 7.1 The First communication window on the IQS333 – RDY is low for the first time

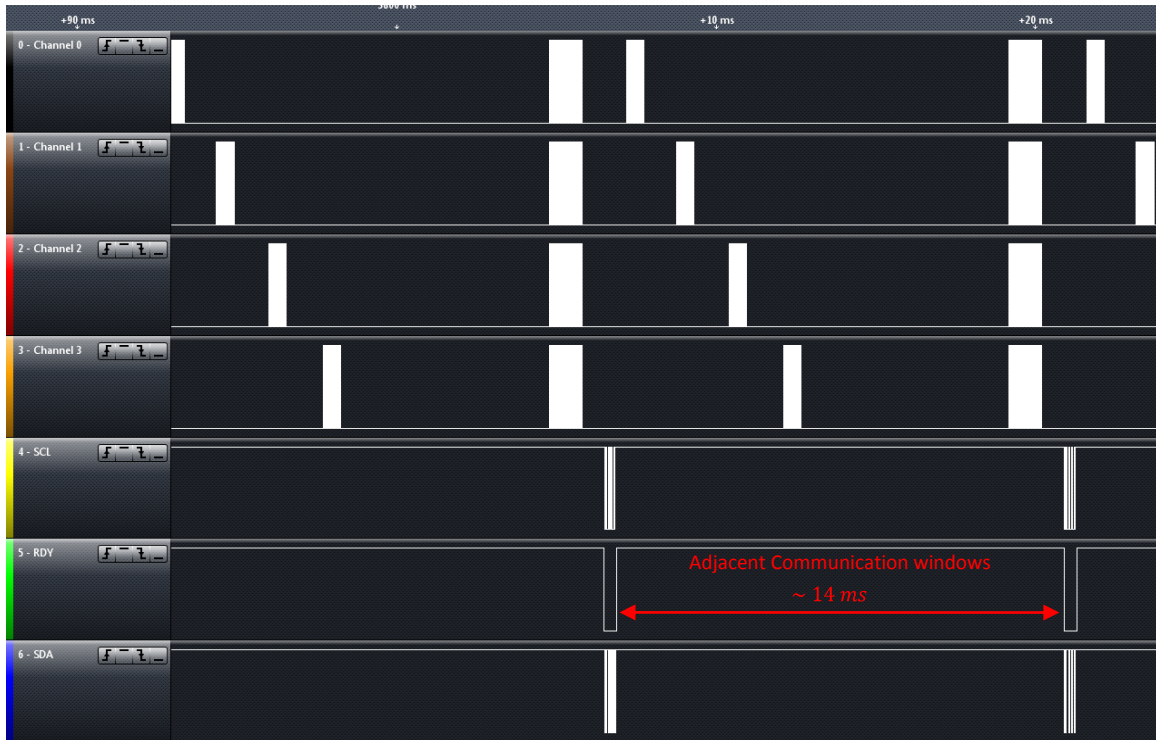


Figure 7.2 Two adjacent communication windows with 8 active channels

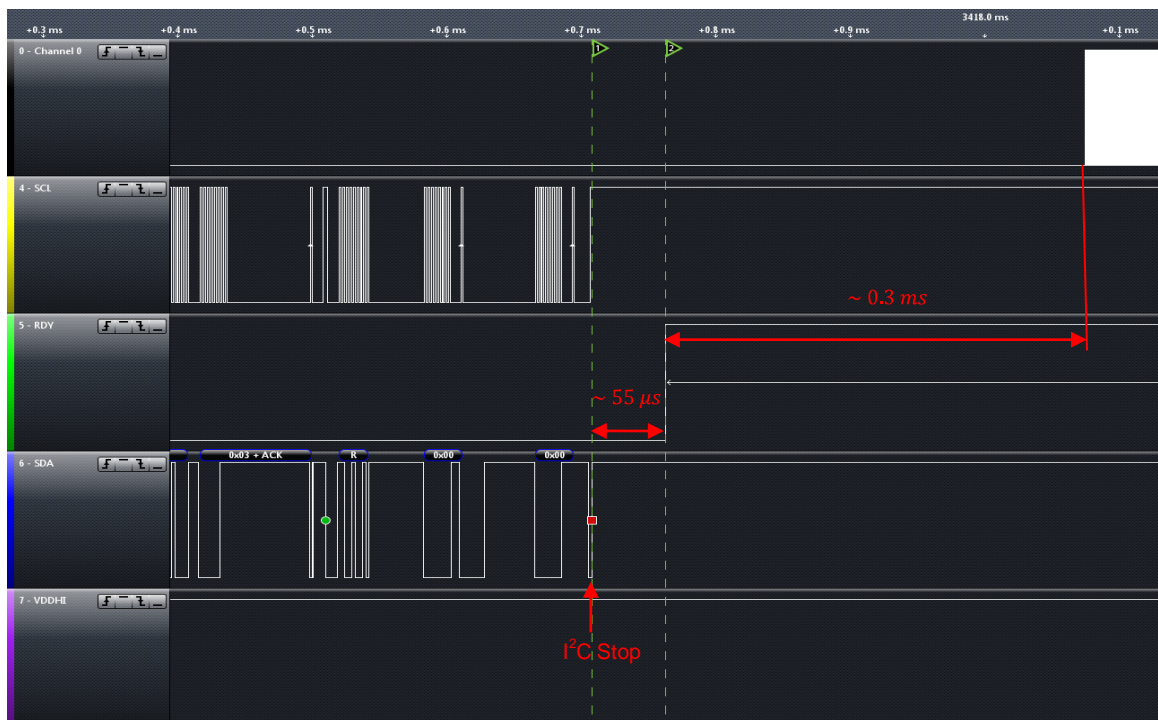


Figure 7.3 Timing between I²C Stop and RDY high and RDY high to next Conversion of channel 1. Avoid sending a start before the current window has closed.




8 Contact Information

	USA	Asia	South Africa
Physical Address	6507 Jester Blvd Bldg 5, suite 510G Austin TX 78750 USA	Rm1725, Glittery City Shennan Rd Futian District Shenzhen, 518033 China	109 Main Street Paarl 7646 South Africa
Postal Address	6507 Jester Blvd Bldg 5, suite 510G Austin TX 78750 USA	Rm1725, Glittery City Shennan Rd Futian District Shenzhen, 518033 China	PO Box 3534 Paarl 7620 South Africa
Tel	+1 512 538 1995	+86 755 8303 5294 ext 808	+27 21 863 0033
Fax	+1 512 672 8442		+27 21 863 1512
Email	kobusm@azoteq.com	linayu@azoteq.com.cn	info@azoteq.com

Please visit www.azoteq.com for a list of distributors and worldwide representation.

The following patents relate to the device or usage of the device: US 6,249,089 B1, US 6,952,084 B2, US 6,984,900 B1, US 7,084,526 B2, US 7,084,531 B2, EP 1 120 018 B2, EP 1 206 168 B1, EP 1 308 913 B1, EP 1 530 178 A1, ZL 99 8 14357.X, AUS 761094, HK 104 14100A, US13/644,558, US13/873,418

IQ Switch®, SwipeSwitch™, ProxSense®, LightSense™, AirButton® and the  logo are trademarks of Azoteq.

The information in this Datasheet is believed to be accurate at the time of publication. Azoteq uses reasonable effort to maintain the information up-to-date and accurate, but does not warrant the accuracy, completeness or reliability of the information contained herein. All content and information are provided on a "as is" basis only, without any representations or warranties, express or implied, of any kind, including representations about the suitability of these products or information for any purpose. Azoteq disclaims all warranties and conditions with regard to these products and information, including but not limited to all implied warranties and conditions of merchantability, fitness for a particular purpose, title and non-infringement of any third party intellectual property rights. Azoteq assumes no liability for any damages or injury arising from any use of the information or the product or caused by, without limitation, failure of performance, error, omission, interruption, defect, delay in operation or transmission, even if Azoteq has been advised of the possibility of such damages. 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, express or implied, by estoppel or otherwise, under any intellectual property rights. In the event that any of the abovementioned limitations or exclusions does not apply, it is agreed that Azoteq's total liability for all losses, damages and causes of action (in contract, tort (including without limitation, negligence) or otherwise) will not exceed the amount already paid by the customer for the products. Azoteq reserves the right to alter its products, to make corrections, deletions, modifications, enhancements, improvements and other changes to the content and information, its products, programs and services at any time or to move or discontinue any contents, products, programs or services without prior notification. For the most up-to-date information and binding Terms and Conditions please refer to www.azoteq.com.

WWW.AZOTEQ.COM

info@azoteq.com