Application Note: AZD067

IQS5xx Trackpad Communication Interface

IQ Switch® - ProxSense® Series

Description of Master I²C Example Firmware for Trackpad Application on IQS5xx

This guide is provided to assist the designer to effortlessly develop firmware that interfaces with the trackpad application on the IQS550, IQS525 and IQS512 platforms. The register specific information for the IQS5xx can be found in the relating product datasheet documents. The I²C master configures and manages the IQS5xx I²C slave, and gives a good platform from which to develop application specific firmware.

Contents

1 Communication Interface

When implementing the master device firmware, please refer to the IQS5xx Trackpad Datasheet for a detailed description of the I^{2}C communication, available address-commands and registers.

1.1 General I^{2}C Hints / Suggestions

1.1.1 Communication Window

Upon implementing the I^{2}C firmware it is important to understand the working of the communication window.

When communicating via I^{2}C, the communication window will close when an I^{2}C STOP is received by the IQS5xx. The IQS5xx will then pull the RDY line low, marking the end of the communication window. It will now proceed with a new conversion, and also update all data. Once this new data is available, another communication window will become available, indicated by RDY going HIGH.

To perform multiple read and write commands within the same communication window, an I^{2}C REPEATED-START must be used to string them together.

1.1.2 I^{2}C- timeout (on IQS5xx)

An on-chip timeout is implemented to prevent stuck conditions on the I^{2}C bus. Depending on the implementation, a situation could arise where either the master or the slave is waiting for an event/condition to occur before proceeding, but due to various reasons it does not, and then the communication is stuck. The reason could be due to factors such as noise, interference etc. To prevent the IQS5xx from permanently waiting in such a state, a configurable timeout is implemented. The counter for this timeout is cleared for every successful byte read or write. If the timeout occurs, the SDA and SCL lines are released by the IQS5xx, and the RDY is pulled low, ending the communication window.

1.1.3 I^{2}C Pull-up Resistors

When implementing I^{2}C it is important to remember the pull-up resistors on the data and clock lines. 4.7kΩ is recommended, but for lower clock speeds bigger pull-ups will reduce power consumption.

1.1.4 NRST

Suggested implementation is to have the VDD and the pull-up resistors connect to the power supply of the device. The NRST pin should then be used to reset the IQS5xx. Remember to hold the NRST low until master setup has been done, with the relevant SDA and SCL I/O's correctly configured.

1.1.5 RDY

It is recommended to connect the RDY to an interrupt-on-change input on the master. This will simplify the program flow, by allowing the firmware to only trigger the data retrieval and processing when the communication window becomes available, without having to poll the

RDY line. This becomes even more convenient when configuring the device in EventMode, which only triggers a communication window when a selectable event has occurred.

For simplicity, this example did not have RDY connected to an interrupt I/O.

1.1.6 Master Timeout

It is recommended to implement a watchdog timer in the I^2C firmware engine on the master side also. This will prevent the master from being stuck in communications permanently. If no legitimate data transfer is seen for a certain period, then the SDA and SCL lines can be released and the communication can be repeated. This should not occur under normal conditions, but makes the system more robust incase of unexpected interference, which could render the master and slave out of sync.

1.2 Communication:

When RDY signals that the communication window is available, the master initiates communication and can read/write the applicable data to/from the slave. Standard I^2C read and write protocol is used, with the standard address replaced by an address-command implementation. An additional/optional RDY line is implemented which allows for optimal data transfer with respect to response rate.

1.2.1 IQS5xx Device Address

The slave device address for the IQS5xx can be found in the relating firmware description. For the purpose of this document the slave address is as shown below:

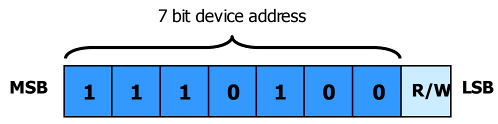

Figure 1.1 Control Byte (device address + r/w)

1.2.2 Address Byte

When reading/writing data bytes, the applicable address is initially written to the slave. Instead of having each byte relating to a specific address, an 'address-command' type structure is implemented. This means that each byte cannot be individually addressed, but instead blocks of data are addressed together under one address-command. For example, if the proximity status bits are required, the proximity status read address-command is configured as the 'address', and all the bytes are read out sequentially under that one address-command.

1.2.3 Writing to the IQS5xx

The master initiates communication by sending an I^2C START condition followed by the device slave address and WRITE bit (0x74 and 0x00 = 0xE8). Next the address-command relating to the required settings is sent, followed by the data bytes.

After all required bytes are written (less can also be sent), then a REPEATED-START can be sent if more Read/Write operations are required within the communication window, otherwise the window can be closed with an I^2C STOP.

Example:

START → ControlByte (0xE8) → Address-Command → Data0 → Data → Data → STOP

1.2.4 Reading from the IQS5xx

The master initiates communication by sending an I²C START condition followed by the device slave address and WRITE bit (0x74 and 0x00 = 0xE8). Next the address-command relating to the required data to read is written to the slave. Now to change to READ state, a REPEATED-START is given, followed by the device slave address and READ bit (0x74 and 0x01 = 0xE9). This can then be followed by reading out the number of required bytes.

After all required bytes are read (less can also be read), then a repeated start can be sent if more Read/Write operations are required within the communication window, otherwise the window can be closed with an I²C STOP.

Example (reading from a specific address-command):

START → ControlByte (0xE8) → Address-Command → Repeated START → ControlByte (0xE9) → Data0 → Data → Data → STOP

At the start of each communication window, a default address-command relating to XY Data Read is loaded. In most cases this will be the data required, and then setting the address-command first is not needed.

Example (reading from default address-command):

START → ControlByte (0xE9) → XYInfoByte → X-high → X-low→ Y-high → XY-low → STOP

Please Note: The default address-command is only reset to the default value at the beginning of the communication window. It is not reset when switching between read and write routines.

2 Example Implementation

A minimalist implementation of the IQS5xx is described in this section. The files required are listed here:

Main.c	*includes.h*
UpperLevel.c	*IQS5xx.h*
LowerLevel.c	*IQS5xx_Init.h*

These files and their functions are also clearly commented, and these together with this section will provide good explanation of the example implementation.

2.1 Overview

This implementation initiates communication between the master (PIC18F4550) and the IQS5xx. The master sends commands to configure the IQS5xx. Once the configuration is completed, the program enters an infinite loop.

In each loop cycle:

- The master waits until the conversion is completed. This is indicated by the RDY line going HIGH, indicating the availability of a communication window with new data available.
- Data is read from IQS5xx (XY Data and Snap Status bytes)
- The data is processed accordingly.

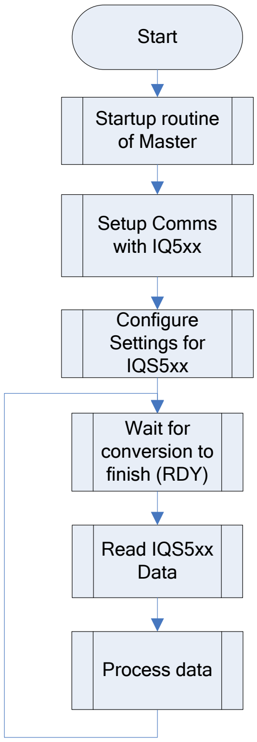

Figure 2.1 Overview

2.2 IQS5xx Settings File (IQS5xx_Init.h)

The device is configured according to the data obtained from the *IQS5xx_Init.h* file. The values can be manually edited in the file, or the file can be created by the PC GUI software. This provides an easy method for ensuring good initial settings are selected and correctly setup on-chip.

Please Note: When connecting the application hardware to the PC via the Azoteq USB Tool, it could alter the performance slightly. For example, if a battery application is being developed, connecting to the PC effectively grounds the application which does not represent true performance. However this change is usually very small, and even in this case, the settings obtained are good initial values from which to work. Small modifications can then be made as needed.

It is thus suggested for ease of use, to firstly connect the application hardware to the PC GUI. This has a convenient graphical interface to assist in obtaining the desired optimal settings. Once the designer is satisfied with the performance, the settings file can be generated. With the optimal settings configured in the GUI, simply click on the *Options* menu, and select *Export H File*, as shown below. The respective *IQS5xx.Init.h* file can now be saved, and used in the master firmware project.

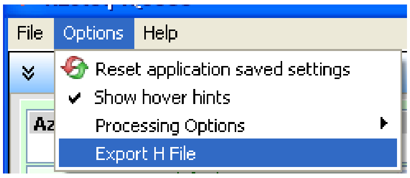

Figure 2.2 Export Settings

2.3 Miscellaneous Functions

2.3.1 Main

The *Main* function sets up the hardware, including writing all required initialisation data to the controller. After initialisation, the function runs the infinite loop to retrieve data from the IQS5xx and to process the data.

Listing 1. Main

```
void main(void)
{
	init();					// Initialise

	while(1)				// endless loop
	{
		IQS5xx_Refresh_Data();		// Obtain new data from IQS5xx
		IQS5xx_Process_New_Data();	// Process the new data
	}
}
```

2.3.2 Init

The *init* function executes commands to setup the system. The ports and registers of the PIC18F4550 are set first. The setup of the I^{2}C is done by calling the *Comms_init* function. The output pin connected to the IQS5xx NRST is set to high to release the device from reset. It is advised to execute all other hardware initialisation routines before initialising the IQS5xx, as other hardware may cause environmental conditions for the IQS5xx to change, affecting the sensing data. With the I^{2}C communication initialised, commands are sent to the IQS5xx via the I^{2}C to set up the required parameters on the device. This is done by calling the *IQS5xx_Settings* function.

Listing 2. Init

```
void init(void)
{
	ADCON1 = 0x0F;			//PORTA all digital operation
	LATA = 0x02;
	TRISA = 0x02;			//RA3 is power output for IQS5xx, RA1 (RDY) input.
	TRISD = 0x00;			//configure PORTD for output
	LATD = 0x0F;			//LEDS off

	LATB = 0xFF;
	TRISB = 0xFF;			//PORTB for input
	INTCON2bits.RBPU = 0;		//For PIC18LF4550-Eval-Rev4A hardware with 74HC573 latch
	TRISCbits.TRISC0 = 0;
	LATCbits.LATC0  = 1;		//un-latch 74HC573 to make OUTD follows LATD
	TRISCbits.TRISC1 = 0;		//enable the latch
	LATCbits.LATC1 = 0;

	Comms_init();

	LATB = 0xFF;
	LATD = 0x0F;

	button_mem = PORTB;
	InitGraph();

//	Place other functions responsible for hardware initialisation here.
	IQS5xx_Settings();
}
```

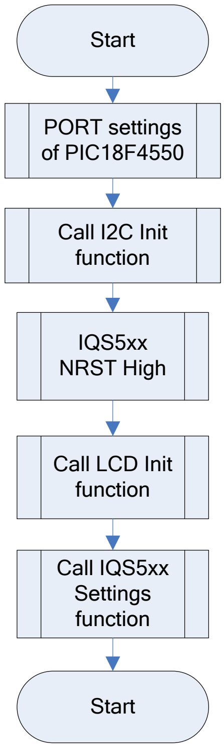

Figure 2.3 Init Function

2.3.3 Comms_init

The *Comms_Init* function sets the registers in the PIC18F4550 to configure the device for the I^{2}C communication.

For this example the IQS550 was powered by an I/O on the PIC, which was also switched on here.

Listing 3. Comms Init

```
void Comms_init()                           //Initiates i2c module on PIC18F4550
{
        SSPADD = 0x02;                      // settings for I2C frequency - 416kHz
        SSPSTAT |= 0x80;                    // slew rate control for high speed (400kHz)
        // SSPADD = 0x08;                   // settings for I2C frequency - 138kHz

        TRISB = TRISB | 0x03;               //set TRISB<0> Make I2c SDA and SCL inputs

        PIR1bits.SSPIF = 0;
        SSPCON1 = 0x28;                     //enables i2c, set to master

        LATA = LATA | 0x08;                 //switch on the IQS5xx
}
```

2.4 Lower Level I^{2}C Function Descriptions

The lower level functions will need to be modified by the designer when implementing the example code on different microcontrollers. If the functions are created with identical functionality, then the Upper Level functions can be reused exactly as they are.

The lower level functions address the PIC18F4550 specific registers and the functions are found in the file *LowerLevel.c*.

2.4.1 CommsIQS5xx_send

The *CommsIQS5xx_send* function is a basic function called by other I^2C communication functions. The data transmission is initiated by writing the data to the PIC data buffer. The transmission complete flag is then monitored, and once the byte is successfully transferred, the flag is cleared. Now the device waits for the ACK from the Slave, and the transmission is complete.

Listing 4. Data Send

```
void CommsIQS5xx_send(unsigned char send_data)
{
        SSPBUF = send_data;                     // write transmit byte to buffer
        while (PIR1bits.SSPIF == 0)             // wait for transmit complete flag
        {}
        PIR1bits.SSPIF = 0;                     //clear flag

        while (SSPCON2bits.ACKSTAT == 1)        //verify IQS5xx acknowledge
        {}
}
```

2.4.2 CommsIQS5xx_read_ack

A read is performed here followed by an ACK. This means that it is not the last byte in the read process.

Listing 5. Read and Acknowledge

```
unsigned char CommsIQS5xx_read_ack(void)
{
        unsigned char temp;

        SSPCON2bits.RCEN = 1;                   // enable master receiver mode
        while (PIR1bits.SSPIF == 0)             // wait for byte received flag
        {}
        PIR1bits.SSPIF = 0;                     //clear flag

        while (SSPSTATbits.BF == 0)             // wait for buffer full flag (receive complete)
        {}
        temp = SSPBUF;                          // store received byte

        SSPCON2bits.ACKDT = 0;                  // enable ACK
        SSPCON2bits.ACKEN = 1;                  // execute ACK sequence

        while (PIR1bits.SSPIF == 0)             // Wait for ACK transmission complete
        {}
        PIR1bits.SSPIF = 0;                     //clear flag

        while (SSPCON2bits.ACKEN == 1)          //verify acknowledge sequence is complete
        {}

        return temp;
}
```

2.4.3 CommsIQS5xx_read_nack

A read is performed here followed by a NACK. This means that this byte is the last one in the current read process.

Listing 6. Read and Not-Acknowledge

```
unsigned char CommsIQS5xx_read_nack(void)
```

```
{
        unsigned char temp;

        SSPCON2bits.RCEN = 1;                 // enable master receiver mode
        while (PIR1bits.SSPIF == 0)           // wait for byte received flag
        {}
        PIR1bits.SSPIF = 0;                   //clear flag

        while (SSPSTATbits.BF == 0)           // wait for buffer full flag (receive complete
        {}
        temp = SSPBUF;                        // store received byte

        SSPCON2bits.ACKDT = 1;                // enable NACK
        SSPCON2bits.ACKEN = 1;                // execute NACK sequence

        while (PIR1bits.SSPIF == 0)           // Wait for NACK transmission complete
        {}
        PIR1bits.SSPIF = 0;                   //clear flag

        while (SSPCON2bits.ACKEN == 1)        //verify acknowledge sequence is complete
        {}

        return temp;
}
```

2.4.4 CommsIQS5xx_start

Firstly this function confirms that the IQS5xx communication window is active, by ensuring that the RDY line is HIGH.

Once this is confirmed, an I²C START condition is generated. This function is the first call at the start of a communication window. If numerous reads and writes are implemented, then the *CommsIQS5xx_repeat_start* function can be used.

Listing 7. I²C Start

```
void CommsIQS5xx_start(void)
{
        while (PORTAbits.RA1 == 0)            //wait for ready
        {}

        SSPCON2bits.SEN = 1;                  //start condition

        while (PIR1bits.SSPIF == 0)           //wait for start condition to be generated
        {}
        PIR1bits.SSPIF = 0;                   //clear flag

        while (SSPCON2bits.SEN == 1)          // verify start is complete
        {}
}
```

2.4.5 CommsIQS5xx_repeat_start

This function generates an I²C START condition. This is used to string together numerous read and writes within the same communication window. A REPEATED-START is exactly the same as a START, except that it is done without being preceded by a STOP. Therefore the same communication window is kept and different address-commands can be read/written.

Listing 8. I²C Repeated Start

```
void CommsIQS5xx_repeat_start(void)
{
```

```
	SSPCON2bits.RSEN = 1;				//start condition

	while (PIR1bits.SSPIF == 0)			//wait for start condition to be generated
	{}
	PIR1bits.SSPIF = 0;				//clear flag

	while (SSPCON2bits.RSEN == 1)			// verify start is complete
	{}
}
```

2.4.6 CommsIQS5xx_stop

This function generates an I²C STOP condition. This ends the communication window, and the IQS5xx will then pull RDY low, and resume with new sensing and data processing.

Listing 9. I²C Stop

```
void CommsIQS5xx_stop(void)
{
	SSPCON2bits.PEN = 1;				//stop condition

	while (PIR1bits.SSPIF == 0)			//wait for stop condition to be generated
	{}
	PIR1bits.SSPIF = 0;				//clear flag

	while (SSPCON2bits.PEN == 1)			// verify stop is complete
	{}
}
```

2.5 Upper Level I²C Function Descriptions

The upper level functions are found in the file *UpperLevel.c*. The lower level functions are used by these to implement the required I²C data protocol. They are designed to be strung together to allow the developer full control over the termination of the communication window, and the required data transfers. The building blocks provided have limitations in terms of the order in which they can be implemented. What this means is that a data WRITE cannot be called without firstly calling an I²C START for example, as is the logical order determined by the I²C protocol. To illustrate this, a flow diagram shows the standard functions implemented with their possible implementations.

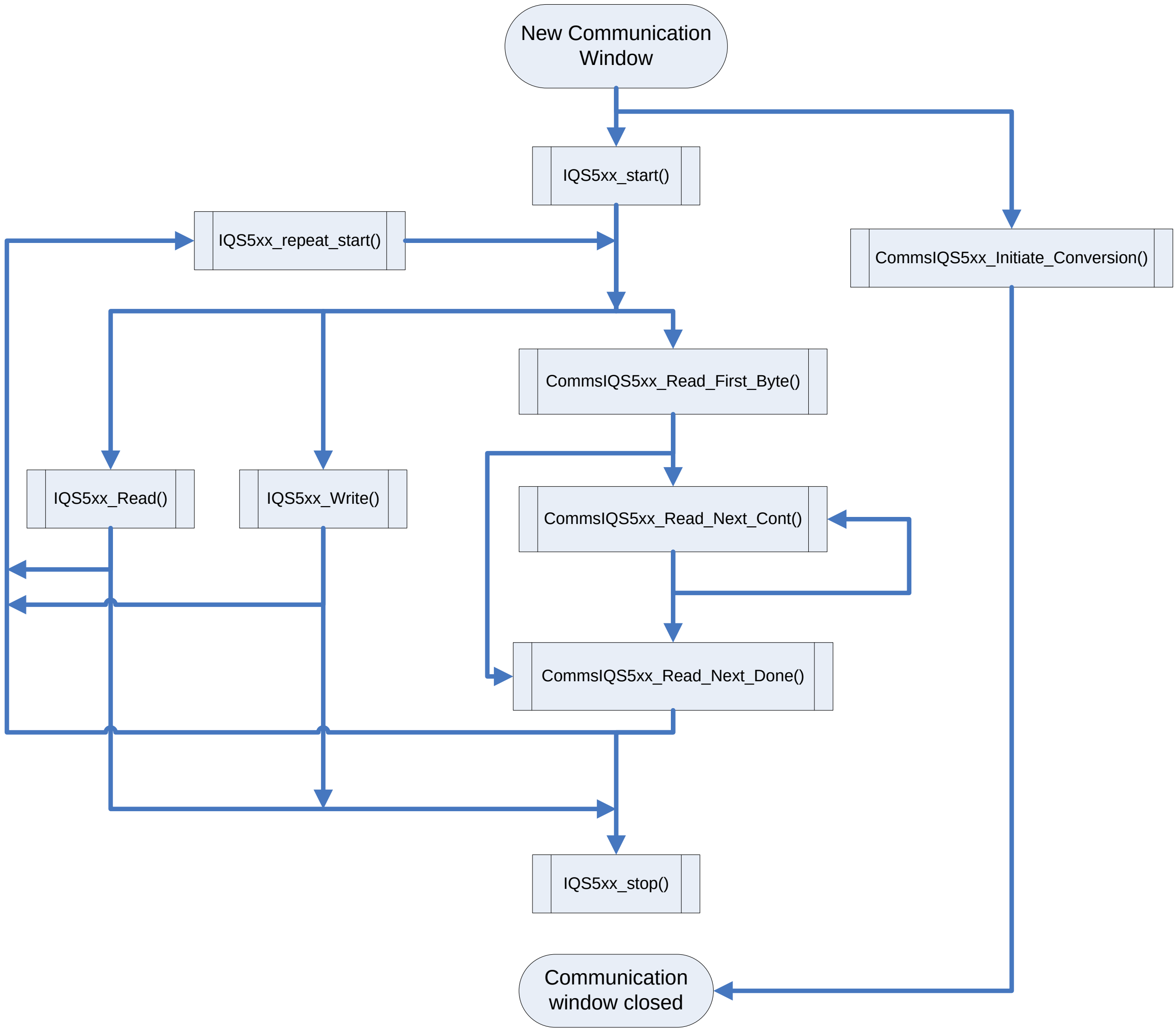

Figure 2.4 Implemented Functions Flow Diagram

2.5.1 IQS5xx_Settings

The *IQS5xx_Settings* function sends the values to the IQS5xx to set the registers necessary for the desired functionality of the IQS5xx. This function uses the data from the *IQS5xx_Init.h* file and configures the device accordingly.

This function must make provision that ANY of the settings can be altered from their default values, and thus all settings are configured. However if default values are used, the specific settings don't need to be sent. The steps taken to setup the device are summarised here:

1- The version information is read, and must match that provided in the *IQS5xx_Init.h* file. This assists in confirming that communication is successfully implemented, and also that the correct device is being used.

2- The reset is acknowledged, which will clear the SHOW_RESET bit in *XYInfoByte*.

3- Channel setup is performed

4- Thresholds required are setup

5- Normal Mode ATI is configured, and the Auto-ATI algorithm is executed

6- Filter settings are configured
7- Hardware parameters are configured (usually not changed)
8- Active Channels are configured (usually not changed)
9- Debounce values are set
10- ProxMode ATI is configured, and the Auto-ATI algorithm is executed
11- Finally, the system operational settings (low-power, sleep, EventMode, system mode etc) are configured as required. These settings are done last, since you want to complete all the settings before changing the way the cycles and communication work.

Now the device is correctly setup and operating according to the supplied parameters.

Listing 10. IQS5xx Setting

```
void IQS5xx_Settings(void)
{
        unsigned char data_buffer[30];

        CommsIQS5xx_start();                        // check comms by confirming Version information
        CommsIQS5xx_Read(VERSION_INFO, &data_buffer, 3);
        CommsIQS5xx_stop();

        if ( (((unsigned int)data_buffer[0]<<8 + (unsigned int)data_buffer[1]) != PRODUCT_NUMBER) ||
         (((unsigned int)data_buffer[2]<<8 + (unsigned int)data_buffer[3]) != PROJECT_NUMBER) ||
         (data_buffer[4] != VERSION_NUMBER) )       // These constants must be updated in the IQS5xx_Init.h file
        {
                // Handle the comms error here!
        }

// ACKNOWLEDGE THE RESET INDICATION FLAG
        data_buffer[0] = ACK_RESET;             // Set ACK_RESET flag, which clears SHOW_RESET in XYInfoByte
        CommsIQS5xx_start();
        CommsIQS5xx_Write(CONTROL_SETTINGS, &data_buffer[0], 1);
        CommsIQS5xx_stop();                     // Now the bit is clear, and if ever SHOW_RESET is set..
                                                // ..an unexpected reset can be handled

// -----------------------------------------------------
// FIRSTLY SETUP THE CHANNELS USED:

        // ChannelSetup Data:
        data_buffer[0] = TOTALRXS_VAL;              // TotalRx
        data_buffer[1] = TOTALTXS_VAL;              // TotalTx
        data_buffer[2] = TRACKPADRXS_VAL;           // TrackPadRx
        data_buffer[3] = TRACKPADTXS_VAL;           // TrackPadTx
        data_buffer[4] = PMSETUP0_VAL;              // PMSetup register
        data_buffer[5] = TXHIGH_VAL;
        data_buffer[6] = TXLOW_VAL;             // Which Tx's are used for the ProxMode channel (projected only)

        CommsIQS5xx_start();
        CommsIQS5xx_Write(CHANNEL_SETUP, &data_buffer[0], 7);
        CommsIQS5xx_stop();

// -----------------------------------------------------
// CONFIGURE THE THRESHOLDS:

        // Threshold settings data
        data_buffer[0] = PROXTHRESHOLD_VAL;         // Prox Threshold
        data_buffer[1] = TOUCHMULTIPLIER_VAL;       // Touch Multiplier
        data_buffer[2] = TOUCHSHIFTER_VAL;          // Touch Shifter
```

```
        data_buffer[3] = PMPROXTHRESHOLD_VAL;                   // PM Prox Threshold
        data_buffer[4] = (unsigned char)(SNAPTHRESHOLD_VAL>>8);        // Snap threshold
        data_buffer[5] = (unsigned char)SNAPTHRESHOLD_VAL;  // Snap threshold
        data_buffer[6] = PROXTHRESHOLD2_VAL;                    // Non-trackpad channels prox threshold
        data_buffer[7] = TOUCHMULTIPLIER2_VAL;                  // Non-trackpad channels Touch Multiplier
        data_buffer[8] = TOUCHSHIFTER2_VAL;                     // Non-trackpad channels Touch Shifter

        CommsIQS5xx_start();
        CommsIQS5xx_Write(THRESHOLD_SETTINGS, &data_buffer[0], 9);  // send thresholds
        CommsIQS5xx_stop();

// ----------------------------------------------------
// SETUP THE ATI PARAMETERS FOR NORMAL MODE, AND EXECUTE AUTO-ATI ALGORITHM;

        // ATI Settings Data:
        data_buffer[0] = (unsigned char)(ATITARGET_VAL>>8);
        data_buffer[1] = (unsigned char)ATITARGET_VAL;          // ATI Target
        data_buffer[2] = ATIC_VAL;                              // ATI C
        data_buffer[3] = (unsigned char)(ATITARGET2_VAL>>8);
        data_buffer[4] = (unsigned char)ATITARGET2_VAL;         // Non-trackpad channels ATI Target
        data_buffer[5] = ATIC2_VAL;                             // Non-trackpad channels ATI C

        CommsIQS5xx_start();
        CommsIQS5xx_Write(ATI_SETTINGS, &data_buffer[0], 6); // Write the ATI Parameters
        data_buffer[0] = AUTO_ATI;                               // Set Auto_ATI, with Mode set to Normal Mode.
        CommsIQS5xx_repeat_start();
        CommsIQS5xx_Write(CONTROL_SETTINGS, &data_buffer[0], 1);
        CommsIQS5xx_stop();                                     // Now IQS5xx will permform the Auto-ATI algorithm

// ----------------------------------------------------
// SETUP THE FILTER PARAMETERS;

        // Filter Settings Data:
        data_buffer[0] = FILTERSETTINGS0_VAL;                   // Numerous filter settings
        data_buffer[1] = TOUCHDAMPING_VAL;                      // XY touch point filtering parameter
        data_buffer[2] = HOVERDAMPING_VAL;                      // XY hover point filtering parameter
        data_buffer[3] = PMCOUNTDAMPING_VAL;                    // ProxMode count value filter parameter (full-speed)
        data_buffer[4] = LPPMCOUNTDAMPING_VAL;                  // ProxMode count value filter parameter (Low Power)
        data_buffer[5] = NMCOUNTDAMPING_VAL;                    // Normal Mode count value filter parameter

        CommsIQS5xx_start();
        CommsIQS5xx_Write(FILTER_SETTINGS, &data_buffer[0], 6);
        CommsIQS5xx_stop();

// ----------------------------------------------------
// SETUP THE TIMING PARAMETERS;

        // Timing Settings Data:
        data_buffer[0] = RESEEDTIME_VAL;                        // LTA reseed timer
        data_buffer[1] = COMMSTIMEOUT_VAL;                      // Inactive i2c timeout value
        data_buffer[2] = MODETIME_VAL;                          // Mode timer value (Switching between modes time)
        data_buffer[3] = LPTIME_VAL;                            // Low power time added in low-power state
        data_buffer[4] = SLEEPTIME_VAL;                         // Sleep time permanently added

        CommsIQS5xx_start();
        CommsIQS5xx_Write(TIMING_SETTINGS, &data_buffer[0], 5);
        CommsIQS5xx_stop();

// ----------------------------------------------------
// SETUP THE HARDWARE PARAMETERS;
```

```
        // Hardware Config Settings Data:
        data_buffer[0] = PROXSETTINGS0_VAL;
        data_buffer[1] = PROXSETTINGS1_VAL;
        data_buffer[2] = PROXSETTINGS2_VAL;
        data_buffer[3] = PROXSETTINGS3_VAL;

        CommsIQS5xx_start();
        CommsIQS5xx_Write(HW_CONFIG_SETTINGS, &data_buffer[0], 4);
        CommsIQS5xx_stop();

// ----------------------------------------------------
// SETUP THE ACTIVE CHANNELS

        // Active Channels Data:
        data_buffer[0] = (unsigned char)(ACTIVECHANNELS0_VAL>>8);
        data_buffer[1] = (unsigned char)ACTIVECHANNELS0_VAL;
        data_buffer[2] = (unsigned char)(ACTIVECHANNELS1_VAL>>8);
        data_buffer[3] = (unsigned char)ACTIVECHANNELS1_VAL;
        data_buffer[4] = (unsigned char)(ACTIVECHANNELS2_VAL>>8);
        data_buffer[5] = (unsigned char)ACTIVECHANNELS2_VAL;
        data_buffer[6] = (unsigned char)(ACTIVECHANNELS3_VAL>>8);
        data_buffer[7] = (unsigned char)ACTIVECHANNELS3_VAL;
        data_buffer[8] = (unsigned char)(ACTIVECHANNELS4_VAL>>8);
        data_buffer[9] = (unsigned char)ACTIVECHANNELS4_VAL;
        data_buffer[10] = (unsigned char)(ACTIVECHANNELS5_VAL>>8);
        data_buffer[11] = (unsigned char)ACTIVECHANNELS5_VAL;
        data_buffer[12] = (unsigned char)(ACTIVECHANNELS6_VAL>>8);
        data_buffer[13] = (unsigned char)ACTIVECHANNELS6_VAL;
        data_buffer[14] = (unsigned char)(ACTIVECHANNELS7_VAL>>8);
        data_buffer[15] = (unsigned char)ACTIVECHANNELS7_VAL;
        data_buffer[16] = (unsigned char)(ACTIVECHANNELS8_VAL>>8);
        data_buffer[17] = (unsigned char)ACTIVECHANNELS8_VAL;
        data_buffer[18] = (unsigned char)(ACTIVECHANNELS9_VAL>>8);
        data_buffer[19] = (unsigned char)ACTIVECHANNELS9_VAL;
        data_buffer[20] = (unsigned char)(ACTIVECHANNELS10_VAL>>8);
        data_buffer[21] = (unsigned char)ACTIVECHANNELS10_VAL;
        data_buffer[22] = (unsigned char)(ACTIVECHANNELS11_VAL>>8);
        data_buffer[23] = (unsigned char)ACTIVECHANNELS11_VAL;
        data_buffer[24] = (unsigned char)(ACTIVECHANNELS12_VAL>>8);
        data_buffer[25] = (unsigned char)ACTIVECHANNELS12_VAL;
        data_buffer[26] = (unsigned char)(ACTIVECHANNELS13_VAL>>8);
        data_buffer[27] = (unsigned char)ACTIVECHANNELS13_VAL;
        data_buffer[28] = (unsigned char)(ACTIVECHANNELS14_VAL>>8);
        data_buffer[29] = (unsigned char)ACTIVECHANNELS14_VAL;

        CommsIQS5xx_start();
        CommsIQS5xx_Write(ACTIVE_CHANNELS, &data_buffer[0], 30);
        CommsIQS5xx_stop();

// ----------------------------------------------------
// SETUP THE DEBOUNCE SETTINGS;

        // DebounceSettings Data:
        data_buffer[0] = PROXDB_VAL;              // Prox Debounce values
        data_buffer[1] = TOUCHSNAPDB_VAL;         // Touch and Snap debounce values


        CommsIQS5xx_start();
        CommsIQS5xx_Write(DB_SETTINGS, &data_buffer[0], 2);
        CommsIQS5xx_stop();
```

```
// -----------------------------------------------------
// SETUP THE ATI PARAMETERS FOR PROX MODE, AND EXECUTE AUTO-ATI ALGORITHM

        // First change the mode to ProxMode:
        data_buffer[0] = MODE_SELECT;                            // Set Mode bit to ProxMode
        CommsIQS5xx_start();
        CommsIQS5xx_Write(CONTROL_SETTINGS, &data_buffer[0], 1);
        CommsIQS5xx_stop();                                      // now next cycle will be in ProxMode

        // ProxMode ATI Settings Data:
        data_buffer[0] = (unsigned char)(PMATITARGET_VAL>>8);
        data_buffer[1] = (unsigned char)PMATITARGET_VAL;         // PM ATI Target
        data_buffer[2] = PMATIC_VAL;                             // PM ATI C

        CommsIQS5xx_start();
        CommsIQS5xx_Write(PM_ATI_SETTINGS, &data_buffer[0], 3);  // Write the ATI Parameters
        data_buffer[0] = MODE_SELECT | AUTO_ATI;                 // Keep ProxMode selected, and enable auto-ati
        CommsIQS5xx_repeat_start();
        CommsIQS5xx_Write(CONTROL_SETTINGS, &data_buffer[0], 1);
        CommsIQS5xx_stop();                                      // go and perform PM Auto-ATI Routine

// -----------------------------------------------------
// NOW FINALLY SETUP THE LOW-POWER, SLEEP, SYSTEM MODE AND EVENT SETTINGS

        // Control Settings Data:
        data_buffer[0] = CONTROLSETTINGS0_VAL;
        data_buffer[1] = CONTROLSETTINGS1_VAL;
        CommsIQS5xx_start();
        CommsIQS5xx_Write(CONTROL_SETTINGS, &data_buffer[0], 2);
        CommsIQS5xx_stop();

// ---- Setup Complete -------
}
```

2.5.2 CommsIQS5xx_Write

The slave device address and the WRITE bit are sent to the IQS5xx. This is followed by the specific address-command. After this, the amount of bytes specified to write, are sequentially written.

As can be seen from Figure 2.4, this function must be preceded with a START or REPEATED-START.

It must also be followed by a STOP or REPEATED-START.

Listing 11. Write

```
void CommsIQS5xx_Write(unsigned char write_addr, unsigned char *data, unsigned char NoOfBytes)
{
        unsigned char i;

        CommsIQS5xx_send((IQS5xx_ADDR << 1) + 0x00);             // device address + write
        CommsIQS5xx_send(write_addr);                            // IQS5xx address-command
        for (i = 0 ; i < NoOfBytes ; i++)
                CommsIQS5xx_send(data[i]);
}
```

2.5.3 CommsIQS5xx_Read

The slave device address and the WRITE bit are sent to the IQS5xx. This is because the address-command must first be written to the device, before the data relating to that address-command can be read. This is followed by the specific address command. After this a

REPEATED-START is done, followed by the slave device address with a READ bit. Now the amount of bytes required are sequentially read.

This function must be preceded with a START or REPEATED-START.

It must also be followed by a STOP or REPEATED-START.

Listing 12. Read

```
void CommsIQS5xx_Read(unsigned char read_addr, unsigned char *data, unsigned char NoOfBytes)
{
        unsigned char i;

        CommsIQS5xx_send((IQS5xx_ADDR << 1) + 0x00);            // device address + write
        CommsIQS5xx_send(read_addr);                            // IQS5xx address-command
        CommsIQS5xx_repeat_start();
        CommsIQS5xx_send((IQS5xx_ADDR << 1) + 0x01);            // device address + read

        if (NoOfBytes > 1)
        {
                for (i = 0; i < NoOfBytes - 1; i++)
                        data[i] = CommsIQS5xx_read_ack();       // all bytes except last must be followed by an ACK
        }

        data[NoOfBytes-1] = CommsIQS5xx_read_nack();            // last byte read must be followed by a NACK
}
```

2.5.4 CommsIQS5xx_Read_First_Byte

If the amount of bytes to read are unknown before starting to read (such as first reading the *XYInfoByte* and then deciding how many XY points are available), then the read function can be broken up into the functions: *CommsIQS5xx_Read_First_Byte*, *CommsIQS5xx_Read_Next_Cont* and *CommsIQS5xx_Read_Next_Done.*

Here the device addressing and address-command setup is performed, similar to the *CommsIQS5xx_Read* function. However only one byte is returned and an ACK is sent to the slave, which indicates that this is not the last read of the sequence.

This function must be preceded by a START or REPEATED-START.

It must be followed by a *CommsIQS5xx_Read_Next_Cont* or *CommsIQS5xx_Read_Next_Done* function.

Listing 13. Read First Byte

```
unsigned char CommsIQS5xx_Read_First_Byte(unsigned char start_addr)
{
        CommsIQS5xx_send((IQS5xx_ADDR << 1) + 0x00);
        CommsIQS5xx_send(start_addr);
        CommsIQS5xx_repeat_start();
        CommsIQS5xx_send((IQS5xx_ADDR << 1) + 0x01);
        return CommsIQS5xx_read_ack();
}
```

2.5.5 CommsIQS5xx_Read_Next_Cont

A next byte is simply read from the slave, and another ACK is sent, again indicating that this is not the last byte to read in the sequence.

This function must be preceded by a *CommsIQS5xx_Read_First_Byte* or *CommsIQS5xx_Read_Next_Cont*.

It must be followed by a *CommsIQS5xx_Read_Next_Cont* or *CommsIQS5xx_Read_Next_Done* function.

Listing 14. Read Next Not Last Byte

```
unsigned char CommsIQS5xx_Read_Next_Cont(void)
{
	return CommsIQS5xx_read_ack();
}
```

2.5.6 CommsIQS5xx_Read_Next_Done

A final byte is read from the slave, and this is indicated by sending the appropriate NACK to the slave, indicating the read sequence is complete.

This function must be preceded by a *CommsIQS5xx_Read_First_Byte* or *CommsIQS5xx_Read_Next_Cont*.

It must be followed by a STOP or REPEATED-START.

Listing 15. Read Last Byte

```
unsigned char CommsIQS5xx_Read_Next_Done(void)
{
	return CommsIQS5xx_read_nack();
}
```

2.5.7 CommsIQS5xx_Initiate_Conversion

This function is simply an example of how to end a communication session quickly without any data being required from that communication window. There is an on-chip I^2C timeout implemented, but if the communication window must be quickly skipped without waiting for that timeout to occur, it can be terminated with this function, which simply gives a START, addresses the IQS5xx, and then gives a STOP.

Listing 16. Initiate a Conversion

```
void CommsIQS5xx_Initiate_Conversion(void)
{
	CommsIQS5xx_start();
	CommsIQS5xx_send((IQS5xx_ADDR << 1) + 0x00);
	CommsIQS5xx_stop();
}
```

2.5.8 IQS5xx_Refresh_Data

This section describes the section labeled ‘Read IQS5xx Data’ from the diagram in Figure 2.1. As mentioned this is simply an example, and any data can be read as required by the application.

The master reads the first byte of the XY Data (address-command 0x01), namely the *XYInfoByte*.

The SHOW_RESET bit is then monitored to flag any unexpected reset condition, which will then need to trigger a repeat of the IQS5xx setup procedure.

Depending on the number of active XY points (NO_OF_FINGERS in *XYInfoByte*), the relative amount of ID, X, Y and Touch Strength data is then read from the IQS5xx.

This example then assumes that the Snap functionality is also implemented. If the SNAP_OUTPUT bit indicates that there is an active snap output, then it proceeds to read these additional status bytes. This is done by using a REPEAT-START to string together the multiple transmissions within the same communication window

For this example no further data is required, and the communication window is closed (to allow the IQS5xx to return to get new data) by sending an I^{2}C STOP.

The following flow diagram illustrates an example of a data retrieving section:

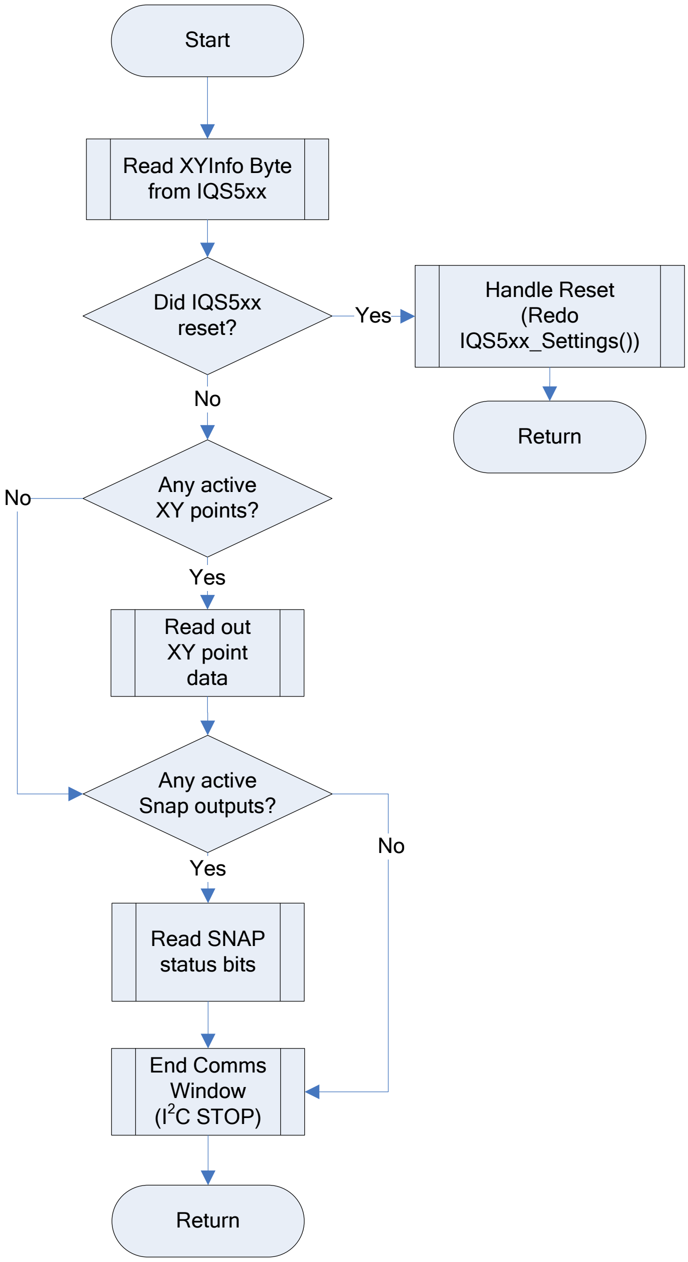

Figure 2.5 Data Retrieval

Listing 17. Refresh Data

```
void IQS5xx_Refresh_Data(void)
{
        unsigned char data_buffer[37], i, j;

        // Read the XY data, but only read as many XY co-ordinates as shown in the XYInfoByte:
        i = 0;
```

```
	CommsIQS5xx_start();
	data_buffer[i++] = CommsIQS5xx_Read_First_Byte(XY_DATA);	// Read first byte (XYInfo)

	//process XYInfoBYte data:
	if ((XYInfoByte & SHOW_RESET) != 0)			// check for an unexpected reset.
	{
		data_buffer[i] = CommsIQS5xx_Read_Next_Done();	// end the read in progress
		CommsIQS5xx_stop();
		// IQS5xx must be configured with application specific settings, since after reset all will return to default
		// ....... add code here .......
	}

	XYInfoByte = data_buffer[0];
	NoOfFingers = XYInfoByte & 0x0F;

	if (NoOfFingers > 0)
	{
		for (j = 0; j < ((NoOfFingers*7)-1); j++)
		{
			data_buffer[i++] = CommsIQS5xx_Read_Next_Cont();
		}
		data_buffer[i++] = CommsIQS5xx_Read_Next_Done();	// last read must have a NACK
	}
	else
		data_buffer[i] = CommsIQS5xx_Read_Next_Done();	// MUST read another because must..
								// ..know about NACK beforehand.

// sort the received XY data in structure
	for (i = 0; i < NoOfFingers; i++)
	{
		IQS5xx[i].ID = data_buffer[(i*7)+1];

		IQS5xx[i].Xpos = (int)(data_buffer[(i*7)+2])<<8;
		IQS5xx[i].Xpos |= data_buffer[(i*7)+3];

		IQS5xx[i].Ypos = (int)(data_buffer[(i*7)+4])<<8;
		IQS5xx[i].Ypos |= data_buffer[(i*7)+5];

		IQS5xx[i].TouchStrength = (int)(data_buffer[(i*7)+6])<<8;
		IQS5xx[i].TouchStrength |= data_buffer[(i*7)+7];
	}

	if ((XYInfoByte & SNAP_OUTPUT) != 0)			// if there are active snap outputs
	{
		CommsIQS5xx_repeat_start();
		CommsIQS5xx_Read(SNAP_STATUS, &data_buffer[0], (TOTALTXS_VAL<<1)); // 2 bytes per Tx for snap status

		// sort the received SNAP data
		for (i = 0; i < TOTALTXS_VAL; i++)
			SnapStatus[i] = (((unsigned int)data_buffer[i<<1])<<8) + ((unsigned int)data_buffer[(i<<1) + 1]);
				// add the upper and lower bytes to get the full word
	}
	else
	{
		for (i = 0; i < TOTALTXS_VAL; i++)
			SnapStatus[i] = 0x0000;			// no snaps, so set registers all to zero
	}

	CommsIQS5xx_stop();
}
```

2.5.9 IQS5xx_Process_New_Data

This function is not provided, since it is dependent on the required application how the data is to be utilised.

2.6 Variables

Some of the variables used in the firmware example are listed here.

Listing 18. Useful Variables

```
typedef struct {
            unsigned char       ID;
            unsigned int        Xpos;
            unsigned int        Ypos;
            unsigned int        TouchStrength;
} IQS5xx_TypeDef;

            unsigned char NoOfFingers;
            unsigned char XYInfoByte;
            unsigned char prevXYInfoByte;

            IQS5xx_TypeDef IQS5xx[5];
            unsigned int SnapStatus[TOTALTXS_VAL<<1];
```

2.7 Constant Declarations

2.7.1 IQS5xx.h

Some device constants are provided in the IQS5xx.h header file. The I²C device slave address used in the functions is defined here. The address-commands implemented are defined. Bit definitions for certain bytes are also defined below this, although not regularly used in the example firmware they can be used for application specific firmware.

Listing 19. Constants declared in IQS5xx.h

```
// i2c slave device address
#define IQS5xx_ADDR 0x74

// Definitions of Address-commands implemented on IQS5xx
#define   VERSION_INFO          0x00        // Read
#define   XY_DATA               0x01        // Read
#define   PROX_STATUS           0x02        // Read
#define   TOUCH_STATUS          0x03        // Read
#define   COUNT_VALUES          0x04        // Read
#define   LTA_VALUES            0x05        // Read
#define   ATI_COMP              0x06        // Read / Write
#define   PORT_CONTROL          0x07        // Read / Write
#define   SNAP_STATUS           0x08        // Read

#define   CONTROL_SETTINGS      0x10        // Read / Write
#define   THRESHOLD_SETTINGS    0x11        // Read / Write
#define   ATI_SETTINGS          0x12        // Read / Write
#define   FILTER_SETTINGS       0x13        // Read / Write
#define   TIMING_SETTINGS       0x14        // Read / Write
#define   CHANNEL_SETUP         0x15        // Read / Write
#define   HW_CONFIG_SETTINGS    0x16        // Read / Write
#define   ACTIVE_CHANNELS       0x17        // Read / Write
#define   DB_SETTINGS           0x18        // Read / Write

#define   PM_PROX_STATUS        0x20        // Read
#define   PM_COUNT_VALUES       0x21        // Read
```

```
#define   PM_LTA_VALUES          0x22             // Read
#define   PM_ATI_COMP            0x23             // Read / Write
#define   PM_ATI_SETTINGS        0x24             // Read / Write

// BIT DEFINITIONS FOR IQS5xx
// XYInfoByte0
#define   NO_OF_FINGERS0         0x01             // Indicates how many co-ordinates are available
#define   NO_OF_FINGERS1         0x02             // Indicates how many co-ordinates are available
#define   NO_OF_FINGERS2         0x04             // Indicates how many co-ordinates are available
#define   SNAP_OUTPUT            0x08             // 0 = no snap outputs / 1 = at least one snap output
#define   LP_STATUS              0x10             // 0 = Charging full-speed  /  1 = Charging in LP duty cycle
#define   NOISE_STATUS           0x20             // 0 = No noise  /  1 = Noise affected data
#define   MODE_INDICATOR         0x40             // 0 = Normal Charging  /  1 = ProxMode charging
#define   SHOW_RESET             0x80             // Indicates reset has occurrd

// Bit definitions - ControlSettings0
#define   EVENT_MODE             0x01             // 0= no event mode / 1=event mode active
#define   TRACKPAD_RESEED        0x02             // Reseed all the normal mode channels
#define   AUTO_ATI               0x04             // Perform AutoATI routine (depend on Mode selected)
#define   MODE_SELECT            0x08             // 0 = Normal Mode  /  1 = ProxMode
#define   PM_RESEED              0x10             // Reseed the Prox Mode channels

#define   AUTO_MODES             0x40             // 0 = Normal/PM manual, 1=Auto switch between NM and PM
#define   ACK_RESET              0x80             // clear the SHOW_RESET flag

// Bit definitions - ControlSettings1
#define   SNAP_EN                0x01             // 0= snaps calculated / 1=not calculated
#define   LOW_POWER              0x02             // 0= normal power charging 1=low power charging
#define   SLEEP_EN               0x04             // 0= no sleep added / 1=permanent sleep time added
#define   REVERSE_EN             0x08     // 0= disabled (conventional prox detection) / 1=enabled (prox trips both ways)
#define   DIS_PMPROX_EVENT       0x10             // 0 = PMProx event enabled / 1 = enabled for EventMode
#define   DIS_SNAP_EVENT         0x20             // 0 = Snap event enabled / 1 = disabled for EventMode

#define   DIS_TOUCH_EVENT        0x40             // 0 = Touch event enabled / 1 = disabled for EventMode

// Bit definitions - FilterSettings0
#define   DIS_TOUCH_FILTER       0x01             // 0=enabled    1=disabled
#define   DIS_HOVER_FILTER       0x02             // 0=enabled    1=disabled
#define   SELECT_TOUCH_FILTER    0x04             // 0=Dynamic filter  1=fixed beta
#define   DIS_PM_FILTER          0x08             // 0 = CS filtered in PM 1= CS raw in PM
#define   DIS_NM_FILTER          0x10             // 0 = CS filtered in NM 1= CS raw in NM

// Bit definitions - PMSetup0
#define   RX_SELECT0             0x01             //   Decimal value selects an INDIVIDUAL Rx for ProxMode
#define   RX_SELECT1             0x02
#define   RX_SELECT2             0x04
#define   RX_SELECT3             0x08

#define   RX_GROUP               0x40             // 0 = RxA  /  1 = RxB
#define   CHARGE_TYPE            0x80             // 0 = Projected  /  1 = Self / surface

// Bit definitions - ProxSettings0
#define   NOISE_EN               0x20             // 0 = noise detection disabled / 1 = noise detection enabled
```

2.7.2 IQS5xx_Init

The default values implemented on-chip are represented in the listing of the constants in the IQS5xx_Init.h file below. As mentioned these will be changed according to the project requirements, but for illustration purposes they are shown with default values here.

Listing 20. Constants declared in IQS5xx_Init.h

```
// Product Number
#define   PRODUCT_NUMBER              40        // Note: 2 bytes
#define   PROJECT_NUMBER              0         // Note: 2 bytes
#define   VERSION_NUMBER              54

#define   CONTROLSETTINGS0_VAL        0x00
#define   CONTROLSETTINGS1_VAL        0x00

#define   PROXTHRESHOLD_VAL           10
#define   TOUCHMULTIPLIER_VAL         5
#define   TOUCHSHIFTER_VAL            7
#define   PMPROXTHRESHOLD_VAL         10
#define   SNAPTHRESHOLD_VAL           100       // Note: 2 bytes
#define   PROXTHRESHOLD2_VAL          10
#define   TOUCHMULTIPLIER2_VAL        5
#define   TOUCHSHIFTER2_VAL           7

#define   ATITARGET_VAL               600       // Note: 2 bytes
#define   ATIC_VAL                    0
#define   ATITARGET2_VAL              600       // Note: 2 bytes
#define   ATIC2_VAL                   0

#define   FILTERSETTINGS0_VAL         0x00
#define   TOUCHDAMPING_VAL            128
#define   HOVERDAMPING_VAL            38
#define   PMCOUNTDAMPING_VAL          16
#define   LPPMCOUNTDAMPING_VAL 128
#define   NMCOUNTDAMPING_VAL          3

#define   RESEEDTIME_VAL              80
#define   COMMSTIMEOUT_VAL            100
#define   MODETIME_VAL                8
#define   LPTIME_VAL                  8
#define   SLEEPTIME_VAL               3

#define   TOTALRXS_VAL                10
#define   TOTALTXS_VAL                15
#define   TRACKPADRXS_VAL             10
#define   TRACKPADTXS_VAL             15
#define   PMSETUP0_VAL                0x40
#define   TXHIGH_VAL                  0x7F
#define   TXLOW_VAL                   0xFF

#define   PROXSETTINGS0_VAL           0x24
#define   PROXSETTINGS1_VAL           0x72
#define   PROXSETTINGS2_VAL           0x15
#define   PROXSETTINGS3_VAL           0x43

#define   ACTIVECHANNELS0_VAL         0x3FF     // Note: 2 bytes
#define   ACTIVECHANNELS1_VAL         0x3FF     // Note: 2 bytes
#define   ACTIVECHANNELS2_VAL         0x3FF     // Note: 2 bytes
#define   ACTIVECHANNELS3_VAL         0x3FF     // Note: 2 bytes
#define   ACTIVECHANNELS4_VAL         0x3FF     // Note: 2 bytes
#define   ACTIVECHANNELS5_VAL         0x3FF     // Note: 2 bytes
#define   ACTIVECHANNELS6_VAL         0x3FF     // Note: 2 bytes
#define   ACTIVECHANNELS7_VAL         0x3FF     // Note: 2 bytes
#define   ACTIVECHANNELS8_VAL         0x3FF     // Note: 2 bytes
#define   ACTIVECHANNELS9_VAL         0x3FF     // Note: 2 bytes
#define   ACTIVECHANNELS10_VAL        0x3FF     // Note: 2 bytes
```

```
#define   ACTIVECHANNELS11_VAL          0x3FF    // Note: 2 bytes
#define   ACTIVECHANNELS12_VAL          0x3FF    // Note: 2 bytes
#define   ACTIVECHANNELS13_VAL          0x3FF    // Note: 2 bytes
#define   ACTIVECHANNELS14_VAL          0x3FF    // Note: 2 bytes

#define   PROXDB_VAL                    0x44
#define   TOUCHSNAPDB_VAL               0x44

#define   PMATITARGET_VAL               500      // Note: 2 bytes
#define   PMATIC_VAL                    0
```

PRETORIA OFFICE

Physical Address
160 Witch Hazel Avenue
Hazel Court 1, 1st Floor
Highveld Techno Park
Centurion, Gauteng
Republic of South Africa

Tel: +27 12 665 2880
Fax: +27 12 665 2883

PAARL OFFICE

Physical Address
109 Main Street
Paarl
7646
Western Cape
Republic of South Africa

Tel: +27 21 863 0033
Fax: +27 21 863 1512

Please visit www.azoteq.com for a full portfolio of the ProxSense™ Capacitive Sensors, Datasheets, Application Notes and Evaluation Kits available.
ProxSenseSupport@azoteq.com