IQS316 Design Guide

IQ Switch® - ProxSense® Series

Multi-channel Integrated Proximity Sensor with Micro-Processor Core

This design guide provides a description of the communication interface between the master and the IQS316 controller. The Memory Map of the IQS316 is provided in this document, followed by a description of each register and instruction. The IQS316 communicates in I^2C or SPI mode, both using a Memory Mapped structure. The last section of this document is dedicated to an example implementation and provides example code.

Contents

1 Memory Map

1.1 General Memory Map Structure

A general I^2C and SPI Memory Map is defined so that all ProxSense® devices can use a standard framework. The general mapping is shown below.

Table 1.1 **IQS316 Memory Mapping**

Address	Access	Size(Bytes)	
00H-0FH	R	16	Device Information

Address	Access	Size(Bytes)	
10H-30H	R	32	Device Specific Data

Address	Access	Size(Bytes)	
31H-34H	R	4	Proximity Status Bytes

Address	Access	Size(Bytes)	
35H-38H	R	4	Touch Status Bytes

Address	Access	Size(Bytes)	
39H-3CH	R	4	Halt Bytes

Address	Access	Size(Bytes)	
3DH-41H	R	4	Active Bytes (indicate cycle)

Address	Access	Size(Bytes)	
42H-82H	R	64	Current Samples

Address	Access	Size(Bytes)	LTAs
83H-C3H	R/W	64	

Address	Access	Size(Bytes)	Device Settings
C4h-FDh	R/W	64	

* Note ‘FE’ and ‘FF’ are reserved for other functions in communication.

1.2 IQS316 Memory Map

1.2.1 Device Information

Address	
00H	
Access	
R	

	Product Number							
Bit	7	6	5	4	3	2	1	0
Value	27 (Decimal)							

Address	
01H	
Access	
R	

	Version Number							
Bit	7	6	5	4	3	2	1	0
Value	Variable							

1.2.2 Device Specific Data

Address	Bit	7	6	5	4	3	2	1	0
		XY Info 1 (UI_FLAGS0)							
10H	Bit	7	6	5	4	3	2	1	0
Access	Value	SHOW_RESET	MODE_INDICATOR	~	~	~	ATI_BUSY	RESEED_BUSY	NOISE
R									

1.2.3 Proximity Status Bytes

Only the proximity status of the channels relating to the current group is available here.

Address		Proximity Status (Group dependant)							
31H	Bit	7	6	5	4	3	2	1	0
Access	Name	If Group = 0				CH3	CH2	CH1	CH0
R	Name	If Group = 1				CH7	CH6	CH5	CH4
	Name	If Group = 2				CH11	CH10	CH9	CH8
	Name	If Group = 3				CH15	CH14	CH13	CH12
	Name	If Group = 4				CH19	CH18	CH17	CH16

1.2.4 Touch Status Bytes

Only the touch status of the channels relating to the current group is available here.

Address		Touch Status (Group dependant)							
35H	Bit	7	6	5	4	3	2	1	0
Access	Name	If Group = 0				~	~	~	~
R	Name	If Group = 1				CH7	CH6	CH5	CH4
	Name	If Group = 2				CH11	CH10	CH9	CH8
	Name	If Group = 3				CH15	CH14	CH13	CH12
	Name	If Group = 4				CH19	CH18	CH17	CH16

*Note: This byte is not used for Group 0 (Prox Mode)

1.2.5 Halt Bytes

Only the filter halt status of the channels relating to the current group is available here.

Address	Halt Status (Group dependant)								
39H	Bit	7	6	5	4	3	2	1	0
Access	Name	If Group = 0				CH3	CH2	CH1	CH0
R	Name	If Group = 1				CH7	CH6	CH5	CH4
	Name	If Group = 2				CH11	CH10	CH9	CH8
	Name	If Group = 3				CH15	CH14	CH13	CH12
	Name	If Group = 4				CH19	CH18	CH17	CH16

1.2.6 Active Bytes

The group number is given here.

Address	Group Number								
3DH	Bit	7	6	5	4	3	2	1	0
Access	Value	Variable (0-4)							
R	Note	Indicates which group’s data is currently available							

1.2.7 Current Samples

The Current Samples of the current group are available here.

Address	Current Sample CH0 / CH4 / CH8 / CH12 / CH16								
42H	Bit	7	6	5	4	3	2	1	0
Access	Value	Variable (HIGH byte)							
R	Note	CH0 (Group0) / CH4 (Group1) / CH8 (Group2) / CH12 (Group3) / CH16 (Group4)							

Address		Current Sample CH0 / CH4 / CH8 / CH12 / CH16							
43H	Bit	7	6	5	4	3	2	1	0
Access	Value	Variable (LOW byte)							
R	Note	CH0 (Group0) / CH4 (Group1) / CH8 (Group2) / CH12 (Group3) / CH16 (Group4)							

Address		Current Sample CH1 / CH5 / CH9 / CH13 / CH17							
44H	Bit	7	6	5	4	3	2	1	0
Access	Value	Variable (HIGH byte)							
R	Note	CH1 (Group0) / CH5 (Group1) / CH9 (Group2) / CH13 (Group3) / CH17 (Group4)							

Address		Current Sample CH1 / CH5 / CH9 / CH13 / CH17							
45H	Bit	7	6	5	4	3	2	1	0
Access	Value	Variable (LOW byte)							
R	Note	CH1 (Group0) / CH5 (Group1) / CH9 (Group2) / CH13 (Group3) / CH17 (Group4)							

Address		Current Sample CH2 / CH6 / CH10 / CH14 / CH18							
46H	Bit	7	6	5	4	3	2	1	0
Access	Value	Variable (HIGH byte)							
R	Note	CH2 (Group0) / CH6 (Group1) / CH10 (Group2) / CH14 (Group3) / CH18 (Group4)							

Address		Current Sample CH2 / CH6 / CH10 / CH14 / CH18							
47H	Bit	7	6	5	4	3	2	1	0
Access	Value	Variable (LOW byte)							
R	Note	CH2 (Group0) / CH6 (Group1) / CH10 (Group2) / CH14 (Group3) / CH18 (Group4)							

Address		Current Sample CH3 / CH7 / CH11 / CH15 / CH19							
48H	Bit	7	6	5	4	3	2	1	0
Access	Value	Variable (HIGH byte)							
R	Note	CH3 (Group0) / CH7 (Group1) / CH11 (Group2) / CH15 (Group3) / CH19 (Group4)							

Address		Current Sample CH3 / CH7 / CH11 / CH15 / CH19							
49H	Bit	7	6	5	4	3	2	1	0
Access	Value	Variable (LOW byte)							
R	Note	CH3 (Group0) / CH7 (Group1) / CH11 (Group2) / CH15 (Group3) / CH19 (Group4)							

1.2.8 Long-Term Averages and Thresholds

The Long-Term averages, and each individual channels thresholds, of the current group are available here to read AND overwrite.

Address		Long-Term Average CH0 / CH4 / CH8 / CH12 / CH16							
83H	Bit	7	6	5	4	3	2	1	0
	Value	Touch Threshold		Prox Threshold		Variable (HIGH byte)			
Access	Default	0	0	0	0				
R/W	Note	CH0 (Group0) / CH4 (Group1) / CH8 (Group2) / CH12 (Group3) / CH16 (Group4)							

Address		Long-Term Average CH0 / CH4 / CH8 / CH12 / CH16							
84H	Bit	7	6	5	4	3	2	1	0
Access	Value	Variable (LOW byte)							
R/W	Note	CH0 (Group0) / CH4 (Group1) / CH8 (Group2) / CH12 (Group3) / CH16 (Group4)							

Address		Long-Term Average CH1 / CH5 / CH9 / CH13 / CH17							
85H	Bit	7	6	5	4	3	2	1	0
	Value	Touch Threshold		Prox Threshold		Variable (HIGH byte)			
Access	Default	0	0	0	0				
R/W	Note	CH1 (Group0) / CH5 (Group1) / CH9 (Group2) / CH13 (Group3) / CH17 (Group4)							

Address		Long-Term Average CH1 / CH5 / CH9 / CH13 / CH17							
86H	Bit	7	6	5	4	3	2	1	0
Access	Value	Variable (LOW byte)							
R/W	Note	CH1 (Group0) / CH5 (Group1) / CH9 (Group2) / CH13 (Group3) / CH17 (Group4)							

Address		Long-Term Average CH2 / CH6 / CH10 / CH14 / CH18							
87H	Bit	7	6	5	4	3	2	1	0
	Value	Touch Threshold		Prox Threshold		Variable (HIGH byte)			
Access	Default	0	0	0	0				
R/W	Note	CH2 (Group0) / CH6 (Group1) / CH10 (Group2) / CH14 (Group3) / CH18 (Group4)							

Address		Long-Term Average CH2 / CH6 / CH10 / CH14 / CH18							
88H	Bit	7	6	5	4	3	2	1	0
Access	Value	Variable (LOW byte)							
R/W	Note	CH2 (Group0) / CH6 (Group1) / CH10 (Group2) / CH14 (Group3) / CH18 (Group4)							

Address		Long-Term Average CH3 / CH7 / CH11 / CH15 / CH19							
89H	Bit	7	6	5	4	3	2	1	0
	Value	Touch Threshold		Prox Threshold		Variable (HIGH byte)			
Access	Default	0	0	0	0				
R/W	Note	CH3 (Group0) / CH7 (Group1) / CH11 (Group2) / CH15 (Group3) / CH19 (Group4)							

Address		Long-Term Average CH3 / CH7 / CH11 / CH15 / CH19							
8AH	Bit	7	6	5	4	3	2	1	0
Access	Value	Variable (LOW byte)							
R/W	Note	CH3 (Group0) / CH7 (Group1) / CH11 (Group2) / CH15 (Group3) / CH19 (Group4)							

1.2.9 Device Settings

An attempt is made so that the commonly used settings are situated closer to the top of the memory block. Settings that are regarded as more 'once-off' are placed further down.

Address		UI Settings 0 (UI_SETTINGS0)							
C4H	Bit	7	6	5	4	3	2	1	0
Access	Name	RESEED	ATI_MODE	PROX RANGE	TOUCH RANGE	FORCE PROX MODE	FORCE TOUCH MODE	ND	0
R/W	Default	0	0	1	0	0	0	1	0

Address		Power Settings (POWER_SETTINGS)							
C5H	Bit	7	6	5	4	3	2	1	0
Access	Name	~	~	~	~	SLEEP	MAIN_OSC	LP1	LP0
R/W	Default	~	~	~	~	0	0	0	0

Address		ProxSense® Module Settings 1 (PROX_SETTINGS_1)							
C6H	Bit	7	6	5	4	3	2	1	0
Access	Name	CXVSS	ZC_EN	HALT1	HALT0	AUTO_ATI	CXDIV2	CXDIV1	CXDIV0
R/W	Default	1	0	0	1	0	0	1	0

Address		ProxSense® Module Settings 2 (PROX_SETTINGS_2)							
C7H	Bit	7	6	5	4	3	2	1	0
Access	Name	~	SHIELD_EN	STOP_COMMS	ACK_RESET	SKIP_CONV	ACF_DISABLE	LTN_DISABLE	WDT_DISABLE
R/W	Default	~	0	0	0	0	0	0 (Note 1)	1

Note1: The LTN filter has a limitation: it is default ON, but it is recommended that this feature be disabled by the user (setting the bit).

Address			ATI Multiplier C (ATI_MULT1)							
C8H	Bit		7	6	5	4	3	2	1	0
Access	Name	If Group = 0	CH3		CH2		CH1		CH0	
R/W		If Group = 1	CH7		CH6		CH5		CH4	
		If Group = 2	CH11		CH10		CH9		CH8	
		If Group = 3	CH15		CH14		CH13		CH12	
		If Group = 4	CH19		CH18		CH17		CH16	
	Default		0	0	0	0	0	0	0	0

Address		ATI Multiplier I (ATI_MULT2)							
C9H	Bit	7	6	5	4	3	2	1	0
Access	Name	CH3	CH2	CH1	CH0	If Group = 0			
R/W		CH7	CH6	CH5	CH4	If Group = 1			
		CH11	CH10	CH9	CH8	If Group = 2			
		CH15	CH14	CH13	CH12	If Group = 3			
		CH19	CH18	CH17	CH16	If Group = 4			
	Default	0	0	0	0	~	~	~	~

Address			ATI Compensation Setting (ATI_C0)							
CAH	Bit		7	6	5	4	3	2	1	0
Access	Value	If Group = 0	CH0							
R/W		If Group = 1	CH4							
		If Group = 2	CH8							
		If Group = 3	CH12							
		If Group = 4	CH16							

Address			ATI Compensation Setting (ATI_C1)							
CBH	Bit		7	6	5	4	3	2	1	0
Access	Value	If Group = 0	CH1							
R/W		If Group = 1	CH5							
		If Group = 2	CH9							
		If Group = 3	CH13							
		If Group = 4	CH17							

Address			ATI Compensation Setting (ATI_C2)							
CCH	Bit		7	6	5	4	3	2	1	0
Access	Value	If Group = 0	CH2							
R/W		If Group = 1	CH6							
		If Group = 2	CH10							
		If Group = 3	CH14							
		If Group = 4	CH18							

Address			ATI Compensation Setting (ATI_C3)							
CDH	Bit		7	6	5	4	3	2	1	0
Access	Value	If Group = 0	CH3							
R/W		If Group = 1	CH7							
		If Group = 2	CH11							
		If Group = 3	CH15							
		If Group = 4	CH19							

Address		Shield Settings (SHLD_SETTINGS)							
CEH	Bit	7	6	5	4	3	2	1	0
Access	Name	~	~	~	~	~	SHLD2	SHLD1	SHLD0
R/W	Default	0	0	0	0	0	0	0	0

* Note this byte will be ignored if SHIELD_EN (PROX_SETTINGS_2<6>) is set (i.e. if automated shield is selected).

Address		Unused (keep 00H)							
CFH	Bit	7	6	5	4	3	2	1	0
Access	Name	~	~	0	0	0	0	0	0
R/W	Default	~	~	0	0	0	0	0	0

Address		Cx Configuration (CX_CONFIG)							
D0H	Bit	7	6	5	4	3	2	1	0
Access	Name	CX_GPIO_1	CX_GPIO_0	~	~	Prox Mode Group Selection (PM_CX_SELECT)			
						GROUP4	GROUP3	GROUP2	GROUP1
R/W	Default	0	0	~	~	1	1	1	1

Address		DEFAULT_COMMS_POINTER							
D1H	Bit	7	6	5	4	3	2	1	0
Access	Default	10H (Beginning of Device Specific Data)							
R/W									

Address		Individual Channel Disable (CHAN_ACTIVE0)							
D2H	Bit	7	6	5	4	3	2	1	0
Access	Name	~	~	~	~	CH3	CH2	CH1	CH0
R/W	Default	~	~	~	~	0	0	1	1

*Note: Only group 0 and 1 are default on, this is because with more than 2 channels active, the AC Filter isn't sampled at the optimal frequency, and is thus less effective.

Address		Individual Channel Disable (CHAN_ACTIVE1)							
D3H	Bit	7	6	5	4	3	2	1	0
Access	Name	~	~	~	~	CH7	CH6	CH5	CH4
R/W	Default	~	~	~	~	1	1	1	1

Address		Individual Channel Disable (CHAN_ACTIVE2)							
D4H	Bit	7	6	5	4	3	2	1	0
Access	Name	~	~	~	~	CH11	CH10	CH9	CH8
R/W	Default	~	~	~	~	1	1	1	1

Address		Individual Channel Disable (CHAN_ACTIVE3)							
D5H	Bit	7	6	5	4	3	2	1	0
Access	Name	~	~	~	~	CH15	CH14	CH13	CH12
R/W	Default	~	~	~	~	1	1	1	1

Address		Individual Channel Disable (CHAN_ACTIVE4)							
D6H	Bit	7	6	5	4	3	2	1	0
Access	Name	~	~	~	~	CH19	CH18	CH17	CH16
R/W	Default	~	~	~	~	1	1	1	1

Address		Individual Channel Reseed (CHAN_RESEED0)							
D7H	Bit	7	6	5	4	3	2	1	0
Access	Name	~	~	~	~	CH3	CH2	CH1	CH0
R/W	Default	~	~	~	~	0	0	0	0

Address		Individual Channel Reseed (CHAN_RESEED1)							
D8H	Bit	7	6	5	4	3	2	1	0
Access	Name	~	~	~	~	CH7	CH6	CH5	CH4
R/W	Default	~	~	~	~	0	0	0	0

Address		Individual Channel Reseed (CHAN_RESEED2)							
D9H	Bit	7	6	5	4	3	2	1	0
Access	Name	~	~	~	~	CH11	CH10	CH9	CH8
R/W	Default	~	~	~	~	0	0	0	0

Address		Individual Channel Reseed (CHAN_RESEED3)							
DAH	Bit	7	6	5	4	3	2	1	0
Access	Name	~	~	~	~	CH15	CH14	CH13	CH12
R/W	Default	~	~	~	~	0	0	0	0

Address		Individual Channel Reseed (CHAN_RESEED4)							
DBH	Bit	7	6	5	4	3	2	1	0
Access	Name	~	~	~	~	CH19	CH18	CH17	CH16
R/W	Default	~	~	~	~	0	0	0	0

Address		Auto ATI Target							
DCH	Bit	7	6	5	4	3	2	1	0
Access	Value	Variable (HIGH Byte)							
R/W	Default	04H (giving a Target value of = 1024 decimal)							

Address		Auto ATI Target							
DDH	Bit	7	6	5	4	3	2	1	0
Access	Value	Variable (LOW Byte)							
R/W	Default	00H (giving a Target value of = 1024 decimal)							

Address		I/O Port							
DEH	Bit	7	6	5	4	3	2	1	0
Access	Name	GPIO_7	GPIO_6	GPIO_5	GPIO_4	GPIO_3	GPIO_2	GPIO_1	GPIO_0
R/W		I/O's can be read, or set/cleared here.							

Address		I/O Tris							
DFH	Bit	7	6	5	4	3	2	1	0
Access	Name	GPIO_7	GPIO_6	GPIO_5	GPIO_4	GPIO_3	GPIO_2	GPIO_1	GPIO_0
R/W	Default	1	1	1	1	1	1	1	1

NOTE: If the pins are used as Cx channels, they MUST be set to inputs in the TRIS register

1.3 Memory Map Description

1.3.1 Device Information

Product Number

The product number for the IQS316 is 27 (decimal).

Version Number

The version number of the device ROM can be read in this byte.

1.3.2 Device Specific Data

XY Info1 (UI_FLAGS0)

Bit 7: ***SHOW_RESET***: This bit can be read to determine whether a reset occurred on the device since the ***ACK_RESET*** bit has been set. The value of ***SHOW_RESET*** can be set to '0' by writing a '1' in the ***ACK_RESET*** bit in the **PROX_SETTINGS_2** byte.

0 = No reset has occurred since last cleared

1 = Reset has occurred

Bit 6: ***MODE_INDICATOR***: Indicates current mode of charging

0 = Currently in Prox Mode

1 = Currently in Touch Mode

Bit 5:3: ***Unused***

Bit 2: ***ATI_BUSY:*** Status of automated ATI routine

0 = Auto ATI is not busy

1 = Auto ATI in progress

Bit 1: ***RESEED_BUSY:*** Global Channel Reseed Status

0 = Reseed is not busy

1 = Reseed is currently taking place

Bit 0: ***NOISE:*** This bit indicates the presence of noise interference.

0 = Current cycle has not detected the presence of noise

1 = Current cycle has detected the presence of noise

1.3.3 Proximity Status Bytes

Proximity Status

The proximity status of the channels relating to the current group can be read here. The current group can be determined by reading the **Group Number** register. The channels and group numbers relate as shown in Table 1.2.

Table 1.2 **Channel data available**

Current Group Number	Channels available
0	CH0 / CH1 / CH2 / CH3
1	CH4 / CH5 / CH6 / CH7
2	CH8 / CH9 / CH10 / CH11
3	CH12 / CH13 / CH14 / CH15
4	CH16 / CH17 / CH18 / CH19

1.3.4 Touch Status Bytes

Touch Status

The touch status of the channels relating to the current group can be read here. The current group can be determined by reading the **Group Number** register. The channels and group numbers relate as shown in Table 1.2.

1.3.5 Halt Bytes

Halt Status

The halt status of the channels relating to the current group can be read here. The current group can be determined by reading the **Group Number** register. The channels and group numbers relate as shown in Table 1.2.

1.3.6 Active Bytes

Group Number

The group number that can be read in this byte indicates which group's data is currently available. Group 0 is the Prox Mode group, and Group 1-4 are the Touch Mode groups.

1.3.7 Current Samples

The Current Samples for the current group can be read in their respective addresses. The HIGH bytes and LOW bytes are found in consecutive addresses.

1.3.8 Long-Term Averages & Touch/Prox Thresholds

The LTA values for the current group can be read in their respective addresses. The HIGH bytes and LOW bytes are found in consecutive addresses.

The first four bits (high nibble) of each LTA HIGH Byte is the Prox and Touch Thresholds for the respective channel. Care must be taken when overwriting a LTA that the required settings are also included in the HIGH byte.

LTA HIGH Byte

Bit 7-6: ***Touch Threshold:*** The value of these two bits, together with the global Touch Range bit determines the Touch Threshold, as shown in Table 1.3.

Bit 5-4: ***Prox Threshold:*** The value of these two bits, together with the Prox Range bit determines the Prox Threshold, as shown in 0.

Bit 3-0: ***LTA<11:8>:*** The upper 4 bits of the LTA.

Table 1.3 **Touch Threshold Values**

Touch Threshold <1:0>	TOUCH_RANGE = 0	TOUCH_RANGE = 1
	Touch Threshold:	
00	**1/32 (default)**	4/16
01	1/16	6/16
10	2/16	8/16
11	3/16	10/16

Table 1.4 **Prox Threshold Values**

Prox Threshold <1:0>	PROX_RANGE = 0	PROX_RANGE = 1
	Prox Threshold	
00	2	**8 (default)**
01	3	16
10	4	20
11	6	30

LTA LOW Byte

Bit 7-0: ***LTA<7:0>:*** The lower byte of the LTA.

1.3.9 Device Settings

UI Settings 0 (UI_SETTINGS0)

Bit 7: ***RESEED:*** Reseed the LTA filter. This can be used to adapt to an abrupt environment change, where the filter is too slow to track this change. Note that with the Short and Long Halt selections, an automatic Reseed will be performed when the halt time has expired, thus automatically adjusting to the new surroundings.

0 = Do not reseed

1 = Reseed (this is a global reseed)

Bit 6: ***ATI_MODE:*** This selects which mode to perform the auto ATI routine on, and the ***AUTO_ATI*** enable bit initiates the routine.

0 = Automated ATI will apply to Prox-Mode channels

1 = Automated ATI will apply to Touch-Mode channels

Bit 5: ***PROX RANGE*****:** Selects between two Prox threshold sets. The range is a global setting and applies to all channels; whereby each channel can then individually be setup to a custom threshold value within this selected range.

0 = Lower range threshold set

1 = Higher range threshold set

Bit 4: ***TOUCH RANGE*****:** Selects between two touch threshold sets. The range is a global setting and applies to all channels; whereby each channel can then individually be setup to a custom threshold value within this selected range.

0 = Lower range threshold set

1 = Higher range threshold set

Bit 3: ***FORCE PROX MODE*****:** Force charging to Prox Mode. If this bit is set, automatic transitions between Prox and Touch Mode are overwritten.

0 = Normal Operation

1 = Only Prox Mode charging

Bit 2: ***FORCE_TOUCH_MODE*****:** Force charging to Touch Mode. If this bit is set, automatic transitions between Prox and Touch Mode are overwritten. Note: this bit takes precedence over Bit 3.

0 = Normal Operation

1 = Only Touch Mode charging

Bit 1: ***ND*****:** Noise Detection Enable. This setting is used to enable the on-chip noise detection circuitry. With noise detected, the noise affected samples will be ignored, and have no effect on the Prox, touch or LTA calculations. The ***NOISE*** bit will appropriately be set as indication of the noise status.

0 = Disable noise detection

1 = Enable noise detection

Bit 0: ***Internal:*** This bit should always keep the value of 0

Power Settings (POWER_SETTINGS)

Bit 7:4: ***Unused***

Bit 3: ***SLEEP***: This bit puts the IC in SLEEP mode. Sleep is entered after termination of the communication window. No processing is done in the sleep state. This function is available in both SPI and I^2C. In SPI, to wake the device from sleep, the /SS line is pulled low, thus selecting the device, whereby waking it from sleep. Communication with the device is then immediately resumed.

In I^2C, to wake the device, the master simply is required to begin communication with the device.

In both cases, when the IC is woken from sleep, the firmware returns to the same communication window that was last used to put the device to sleep, thus no new sample data is available. Note that if the IC has been in SLEEP for a considerable time, it is recommended to reseed the channels, if no interaction is assumed.

0 = No effect

1 = Puts device in sleep mode

Bit 2: ***MAIN_OSC***: Select the frequency of the main oscillator

0 = 8MHz

1 = 4MHz (not recommended)

Bit 1-0: ***LP***: Low Power (LP) options

00 = Normal Power

01 = LP1 ~100ms charging

10 = LP2 ~ 200ms charging

11 = LP3 ~ 300ms charging

ProxSense® Module Settings 1 (PROX_SETTINGS_1)

Bit 7: ***CXVSS***: Ground Cx channels when inactive. The default and recommended setting is grounded. The result is illustrated by means of an example. If for instance Group 1 is charging, all surrounding sensing lines not part of Group 1 are grounded, and thus in a defined state. If the Cx's are set to float, then their state is unknown, and the sensors influence each other greatly, which is not ideal.

0 = Cx's float

1 = Cx's grounded

Bit 6: ***ZC_EN***: Enable zero-cross (ZC) triggered conversions. An input signal must be connected to the ZC_IN I/O to synchronise the charging to. This is occasionally used in high AC noise applications, whereby synchronising the charging to the AC, the noise is reduced. This input allows the timing of the conversions to be accurately controlled. Possibly the conversions can be sliced between noise events to keep the samples noise free.

0 = No Zero-Cross signal implemented

1 = Conversions synchronised to ZC_IN

Bit 5-4: ***HALT*****:** LTA Filter Halt selections

00 = Short (LTA filter halts for ~20 seconds, then reseeds)

01 = Long (LTA filter halts for ~40 seconds, then reseeds)

10 = Never (LTA filter never halts)

11 = Always (LTA filter is halted permanently)

Bit 3: ***AUTO_ATI*****:** Enable the automated ATI routine. By enabling this bit, the device will perform an automated ATI routine on the selected groups (selected by ATI_MODE), and will attempt to reach the target setup in AUTO-ATI Target. Note that the ATI routine is only started after the communication window is closed, and thus the ATI_BUSY bit will only be set in the following communication window.

0 = No action

1 = Begin auto ATI routine

Bit 2-0: ***CXDIV[2:0]*****:** Selection bits for charge transfer frequency

Table 1.5 **Charge transfer frequency**

MAIN_OSC = 4MHz		MAIN_OSC = 8MHz	
CXDIV	**Conversion Frequency**	**CXDIV**	**Conversion Frequency**
000	2MHz	000	4MHz
001	1MHz	001	2MHz
010	500kHz	**010**	**1MHz (default)**
011	250kHz	011	500kHz
100	125kHz	100	250kHz
101-111	62.5kHz	101-111	125kHz

The charge transfer frequency is a very important parameter. Dependant on the design application, the device frequency must be optimised. For example, if keys are to be used in an environment where steam or water droplets could form on the keys, a higher transfer frequency improves immunity. Also, if a sensor antenna is a very large object/size, then a lower frequency must be selected since the capacitance of the sensor is large, and a lower frequency is required to allow effective capacitive sensing on the sensor.

ProxSense® Module Settings 2 (PROX_SETTINGS_2)

Bit 7: ***Unused***

Bit 6: ***SHIELD_EN*****:** Automatic shield implementation. Each group will have a shield setup automatically on the two shield outputs, according to Table 1.6.

Table 1.6 **Automated Shield Channels**

Group	SHLD_A	SHLD_B
0	CxA0	CxB0
1	CxA0	CxB0
2	CxA1	CxB1
3	CxA2	CxB2
4	CxA3	CxB3

0 = Shield is set by SHIELD_SETTING byte

1 = Shield is automatically loaded according to Table 1.6

Bit 5: ***STOP_COMMS*****:** Skip the SPI/I^{2}C communication window. This can be used if the master does not want to service the IQS316 every charge cycle. Normal operation of the IC continues, and only the communication window is bypassed. Only when the master initiates, or when a Prox is sensed, will the communication be resumed.

0 = Normal Communication

1 = Communication aborted until Prox detected, or master forces a resume

Bit 4: ***ACK_RESET*****:** Acknowledge 'SHOW_RESET'.

0 = Nothing

1 = Clear the flag **SHOW_RESET** (send only once)

Bit 3: ***SKIP_CONV*****:** Don't perform conversion. This can be used, for example if settings for all the groups are to be written. The current groups' settings can be completed, and the communication window can then be terminated. The device then loads the next groups' data (without performing a conversion), and the next communication window is available. Stepping through all the groups can thus be done without the need to wait for a conversion to complete.

0 = Normal operation

1 = Skip conversions (Load next group's data and return to communication window)

Bit 2: ***ACF_DISABLE*****:** Disable the AC Filter on Group 0.

0 = AC Filter is enabled

1 = AC Filter is disabled

Bit 1: ***LTN_DISABLE*****:** Disable the LTN Filter on Group 0.

0 = LTN Filter is enabled

1 = LTN Filter is disabled (recommended due to device limitation)

Bit 0: **WDT_DISABLE**: Device watchdog timer (WDT) disable.

0 = Enabled

1 = Disabled

ATI Multiplier C (ATI_MULT1)

The ATI Multiplier and ATI Compensation bits allow the controller to be compatible with a large range of sensors, and in many applications with different environments. ATI allows the user to maintain a specific sample value on all channels. The ATI Multiplier parameters would produce the largest changes in sample values and can be thought of as the high bits of ATI. The ATI Compensation bits are used to influence the sample values on a smaller scale to provide precision when balancing all channels as close as possible to the target. The ATI Multiplier parameters are further grouped into two parameters namely ATI Multiplier C and ATI Multiplier I. ATI multiplier I consists of a single bit and has the biggest effect on the sample value and can be considered as the highest bit of the ATI parameters.

The ATI_MULT1 byte contains the ATI Multipliers C settings for all channels of the current group. Each channel has two ATI Multiplier C bits where the value of '11' would provide the highest CS value and the value of '00' would provide the lowest.

ATI Multiplier I (ATI_MULT2)

The ATI Multiplier I bit is the ATI bit which would make the largest adjustment to the sample value. The ATI_MULT3 byte contains the ATI Multiplier I settings for all the channels in the current group, where a value of '1' would produce the largest sample value and a value of '0' would produce the smallest sample value.

ATI Compensation Settings

The ATI Compensation parameter can be configured for each channel in a range between 0-255 (decimal). The ATI compensation bits can be used to make small adjustments of the sample values of the individual channels.

Shield Settings (SHLD_SETTINGS)

If the **SHIELD_EN** bit is set, the value written to the **SHLD_SETTINGS** register is simply ignored. Otherwise the shield can be manually configured here.

The **SHLD_SETTINGS** byte is used to enable or disable the two active shields. Bit 0-2 control which sensor lines are to be shielded on **SHLD_A** and **SHLD_B**. By default the shields are disabled with **SHLD_SETTINGS** = 0. Manual configuration is implemented as shown in Table 1.7.

Table 1.7 **SHLD_A and SHLD_B configuration**

SHLD_SETTINGS<2:0>	SHLD_A input connected to	SHLD_B input connected to
000	**SHLDL off (default)**	**SHLDR off (default)**
001	CxA6	CxB6
010	CxA5	CxB5
011	CxA4	CxB4
100	CxA3	CxB3
101	CxA2	CxB2
110	CxA1	CxB1
111	CxA0	CxB0

Cx Configuration (CX_CONFIG)

Bit 7: ***CX_GPIO_1*: Cx or I/O selection.**

0 = CxA7, CxA6, CxB7 and CxB6 are used as sensor lines

1 = GPIO_7, GPIO_6, GPIO_5 and GPIO_4 are implemented as I/O's

Table 1.8 **Upper Nibble of I/O Port Selection**

CX_GPIO_1 Selection	CxA7 / GPIO_7 function	CxA6 / GPIO_6 function	CxB7 / GPIO_5 function	CxB6 / GPIO_4 function
0	CxA7	CxA6	CxB7	CxB6
1	GPIO_7	GPIO_6	GPIO_5	GPIO_4

Bit 6: ***CX_GPIO_0*: Cx or I/O selection.**

0 = CxA5, CxA4, CxB5 and CxB4 are used as sensor lines

1 = GPIO_3, GPIO_2, GPIO_1 and GPIO_0 are implemented as I/O's

Table 1.9 **Lower Nibble of I/O Port Selection**

CX_GPIO_1 Selection	CxA5 / GPIO_3 function	CxA4 / GPIO_2 function	CxB5 / GPIO_1 function	CxB4 / GPIO_0 function
0	CxA5	CxA4	CxB5	CxB4
1	GPIO_3	GPIO_2	GPIO_1	GPIO_0

Please note that if the pins are selected as I/O's, then the TRIS and PORT can be configured as required. However if the pins are used as Cx sensors, then the TRIS MUST be set as inputs ('1') for those specific channels.

Bit 3-0: ***PM_CX_SELECT:*** Groups who's Cx's are included in Prox Mode charging

0 = Group not included

1 = Group included

In this register, a selection of groups 4-1 is made to determine which sensor lines will be used during Prox Mode charging (Group 0). Each bit therefore represents four sensor lines to be added or removed from Group 0.

*Note that at least two groups have to be set for this selection.

Table 1.10 **Sensor Line Selection for Prox Mode**

CX_CONFIG bit	CH0	CH1	CH2	CH3
0 (Group 1 channels)	CxA0	CxB0	CxA4	CxB4
1 (Group 2 channels)	CxA1	CxB1	CxA5	CxB5
2 (Group 3 channels)	CxA2	CxB2	CxA6	CxB6
3 (Group 4 channels)	CxA3	CxB3	CxA7	CxB7

To help illustrate this, an example is provided. If bit 0 and 2 are set in CX_CONFIG, the channels used in Prox Mode are shown in Table 1.11. It can be seen that the Proximity channel 0 (CH0) consists of the two sensor lines CxA0, and CxA2. And similarly the CH1 to CH3's sensor lines can be noted. This example thus has 8 of the 16 sensor lines also providing Proximity input. The other 8 have no influence on the Prox Mode channels.

Table 1.11 **PM_CX_SELECT example**

CX_CONFIG	CH0	CH1	CH2	CH3
CX_CONFIG= 05H	CxA0, CxA2	CxB0, CxB2	CxA4, CxA6	CxB4, CxB6

It can be seen that if all 4 bits are set, all 16 of the Cx sensor lines are antenna inputs for the Prox Mode. It is recommended that if the design has any sensor buttons close to noise sources (negative influence on proximity), that these can be chosen to fall in the same group, which can then be excluded from Prox Mode by means of the PM_CX_SELECT register.

Default Comms Pointer

The value stored in this register will be loaded into the Comms Pointer at the start of a communication window. For example, if the design only requires the Proximity Status information each cycle, then the *Default Comms Pointer* can be set to ADDRESS '31H'. This would mean that at the start of each communication window, the comms pointer would already be set to the Proximity Status register, simply allowing a READ to retrieve the data, without the need of setting up the address.

Individual Channel Disable

Each channel can be individually disabled in these registers. Note that the current group number has no influence on these registers as each channel disable register has a unique address.

Individual Channel Reseed

Each channel can be individually reseeded in these registers. Note that the current group number has no influence on these registers as each channel reseed register has a unique address. Also note that these bits are set initially by the IQS316 so that all channels are reseeded at startup, but are cleared immediately when each cycle is processed. However, the defaults are shown as '0', since after 1 cycle they are then cleared.

Auto-ATI Target

The automated ATI target can be set in these two consecutive registers. These registers are used for the Prox Mode, as well as the Touch Mode ATI targets. The selection between which of these modes to Auto-ATI, is set by **ATI_MODE** in **UI_SETTINGS0<6>**.

For example, if the Prox Mode channels must be tuned to sample values = 800, and the Touch Mode channels to sample values = 400, the following steps are taken:

Step 1: Set *Auto ATI Target* to 800

Step 2: Select Prox Mode for ATI by clearing ATI_MODE bit (UI_SETTINGS0<6> = 0)

Step 3: Start Auto-ATI procedure by setting AUTO-ATI bit (PROX_SETTINGS_1<3>)

Step 4: Wait for Prox Mode ATI to complete, which is when ATI_BUSY bit clears (UI_FLAGS0<2> = 0)

Step 5: Set *Auto ATI Target* to 400

Step 6: Select Touch Mode for ATI by setting ATI_MODE bit (UI_SETTINGS0<6> = 1)

Step 7: Start Auto-ATI procedure by setting AUTO-ATI bit (PROX_SETTINGS_1<3>)

I/O Port and Tris

When setup to be used as I/O's (CX_GPIO_1 and CX_GPIO_0 settings), the data direction can be set in the **I/O Tris** register as shown in Table 1.12.

If used as Cx's, the TRIS must be set as inputs!

Table 1.12 **Tris Configuration**

Tris bit <7:0>	I/O configuration
0	Output
1	Input / Tri-state

If setup as outputs, the state of the I/O's can then be set in the register as shown in Table 1.13

Table 1.13 **I/O Outputs**

Port bit <7:0>	I/O status
0	Output LOW
1	Output HIGH

If setup as inputs, the status of the I/O's can be read from the register.

2 General Implementation hints

When implementing the communication interface with the IQS316, please refer to the IQS316 datasheet for a detailed description of the SPI and I^2C communication. This section contains some general guidelines and hints regarding the communication interface.

2.1 Communication window

Upon implementing either SPI or I^2C it is important to note the difference in the working of the communication window.

2.1.1 SPI Communication window

When communicating via SPI, the communication window will remain open until a new conversion command is received (FE written to IQS316 in 'address' time-slot of write transaction). While the communication window is open the master may initiate and terminate as many read and write communication sessions as required.

2.1.2 I^2C Communication window

When communicating via I^2C, the communication window will automatically close when an I^2C STOP bit is received by the IQS316. The IQS316 will then proceed to start with a new conversion and the READY line will be pulled low until the new conversion is complete.

Note that there is no command via I^2C to initiate a new conversion. To perform multiple read and write commands, the repeated start function of the I^2C must be used to stack the commands together.

2.2 Startup Procedure

The following procedures are for setup of specific features of the IQS316 that requires more attention. More features can be setup by setting the appropriate registers as required.

2.2.1 Individual Prox and Touch Thresholds

Step 1: First set PROX_SETTINGS_2<3> to skip conversion. This will ensure that the system always cycles through all the groups.

Step 2: Read the group number from the GROUP_NUM register (3Dh).

Step 3: In a 'switch' construct, check which group number was read.

Step 4: For this group set the thresholds by writing the chosen threshold values bits 7-4 of 83h, 85h, 87h or 89h, depending on the group number. Remember the threshold values are also determined by the Prox range bit, UI_SETTINGS0<5>, which will set the range for all the groups.

Step 5: End the I^2C window and allow for small delay. No conversions will take place but the group number will increment.

Step 6: Read the group number again, and repeat Steps 3-5 until all the groups 0 to 4 has been configured.

Step 7: Make sure to disable skip conversions so that sensing can resume.

2.2.2 Auto ATI Procedure

For example, if the Prox Mode channels must be tuned to sample values = 800, and the Touch Mode channels to sample values = 400. It is necessary to force the IQS316 to Prox- or touch-mode during setup of the auto ATI. The following steps are taken:

Step 1: Set *Auto ATI Target* to 800

Step 2: Select Prox Mode for ATI by clearing ATI_MODE bit (UI_SETTINGS0<6> = 0), here it is crucial to end the communication window so that the next cycle is in Prox Mode.

Step 3: Start Auto-ATI procedure by setting AUTO-ATI bit (PROX_SETTINGS_1<3>)

Step 4: Wait for Prox Mode ATI to complete, which is when ATI_BUSY bit clears (UI_FLAGS0<2> = 0)

Step 5: Set *Auto ATI Target* to 400

Step 6: Select Touch Mode for ATI by setting ATI_MODE bit (UI_SETTINGS0<6> = 1), here it is crucial to end the communication window so that the next cycle is in Prox Mode.

Step 7: Start Auto-ATI procedure by setting AUTO-ATI bit (PROX_SETTINGS_1<3>)

Step 8: Wait for Touch Mode ATI to complete, which is when ATI_BUSY bit clears (UI_FLAGS0<2> = 0)

Although the ATI is finished, the current samples will take a few conversions to settle at the correct value.

2.2.3 Post Setup

After sending initial settings to the IQS316, it is recommended to execute a reseed. If the Auto ATI was done last, it may not be necessary to reseed.

2.3 General I²C Hints

2.3.1 I²C Pull-up resistors

When implementing I²C 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.

2.3.2 MCLR

Suggested implementation is to have the IQS316 and the pull-up resistors connect to the power supply of the device. The MCLR pin should then be used to reset the IQS316. Remember to hold the MCLR low until master setup has been done.

2.3.3 Reset Device while using I²C

When a reset occurs, some care needs to be taken to ensure that the IQS316 restarts correctly. The reset pin needs to be LOW before the IQS316 can be initialised, otherwise the master will read a ready signal prematurely. To accomplish this without any delays, define the ready pin on the master as an output and pull it LOW. Then, redefine it as an input line just before initializing the IQS316.

3 Sample implementation

An example implementation of the IQS316 is described in this section. This implementation performs a setup of the IQS316, and then retrieves Prox and touch data for each cycle. For this implementation a PIC18F4550 was used as the master device.

Communication between the master and the IQS316 was done in SPI and for I²C, and are both covered in this section. For further explanations of the I²C and SPI protocol, refer to the IQS316 datasheet.

The example implementation firmware was done in MPLAB X version 1.85.

The compiler used was C18 version 3.46.

Complete projects for SPI and I²C are available for reference.

3.1 Overview

- Firstly an initialisation function configures the PIC microcontroller
- Then the communication is configured (either SPI or I²C) for communication between the PIC and the IQS316. (NOTE: the selection between SPI and I²C must be done separately in hardware on the IQS316 PCB by correctly configuring the SPI_SELECT)
- A delay is added to allow the IQS316 to start up correctly (the datasheet says 16ms can be expected until RDY is active for the first time)
- Now the settings on the IQS316 are configured via I²C or SPI
- The setup is now completed and the system enters an endless loop where new data is obtained from the IQS316, and then processed.

3.1.1 Communications:

For a detailed description of the communication protocol refer to the IQS316 datasheet.

<u>SPI:</u>

Writing to IQS316: The master initiates communication by writing a zero (00H) to the IQS316. Next the address to write is sent to the IQS316. The byte sent after the address will be written to that address.

Another address can now be sent to the IQS316. Communication is terminated by sending FFH instead of an address. (This only ends the transaction, and not the current communication window)

E.g. write 35H to address 12H:

Master writes 00H to IQS316. (Initiates comms in write mode, FFH returned)

Master writes 12H to IQS316. (Setup address, returns 00H)

Master writes 35H to IQS316. (35H stored at address 12H, 01H returned)

Master writes FFH to IQS316. (End write cycle, 00H returned)

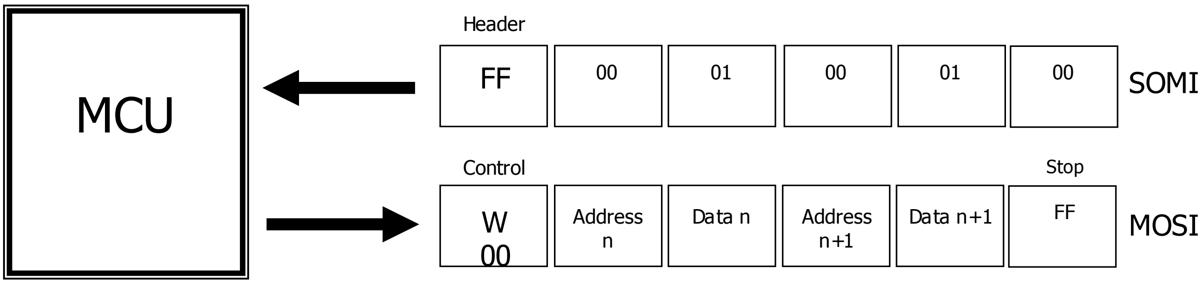

Figure 3.1 SPI write sequence

Additionally, if the master writes FEH to the IQS316, a new conversion will be initiated and the communication window will be terminated.

Reading from the IQS316: The master initiates communication by writing a one (01H) to the IQS316. During each communication cycle (one byte transmitted and received) the data stored at the location indicated by the address pointer will be sent to the master. The address pointer value in turn is replaced by the data sent to the IQS316 by the master. However, upon receiving FEH from the master the address pointer is simply incremented. The default value of the address pointer is 10H. The master ends this transfer by writing FFH to the IQS316.

E.g. read address 15H and 16H:

Master writes 01H to IQS316. (Initiates comms in read mode, FFH returned)

Master writes 15H to IQS316. (Set pointer to 15H, data stored at current pointer address returned)

Master writes FEH to IQS316 (pointer incremented, data stored at 15H returned)

Master writes FFH to IQS316. (End read cycle, data stored at 16H returned)

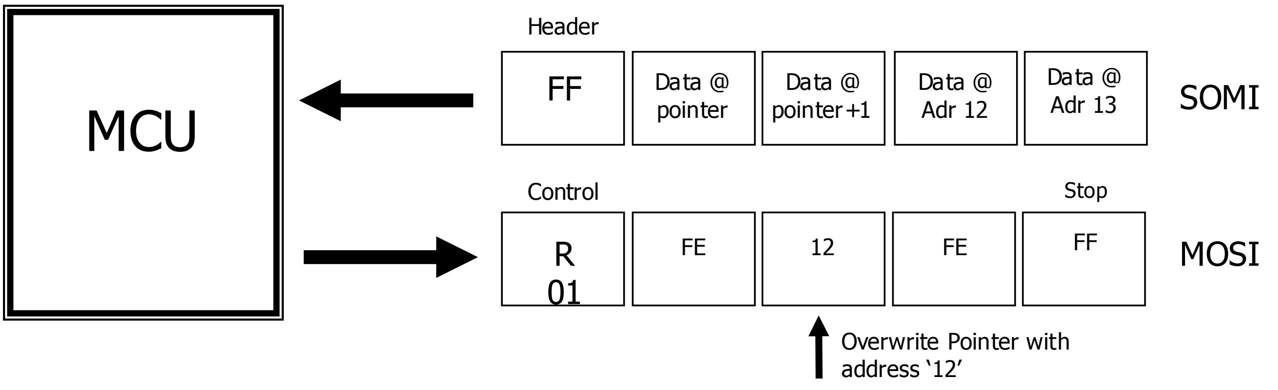

Figure 3.2 SPI read sequence

Please Note: The internal address pointer is only reset to the default value (10H) when a new conversion is called. It is not reset when switching between read and write routines.

I^2C

Standard I^2C read and write protocol is used.

Writing to IQS316: To write an I^2C START condition is generated. This is followed by the device address with the WRITE bit configured. The next byte is the starting address of where to write. All following bytes are then written to the IQS316. Once complete, either a repeated start is given to start another data transaction, or an I^2C STOP is done, which ends the communication window.

Reading from the IQS316: The master again sends and I^2C START, followed by the device address with a WRITE bit configured. The address from where to READ is then WRITTEN to the IQS316. Now a repeated-START is sent, and the device address with a READ bit is sent. Then all the required bytes can be read from the IQS316, again ending in a STOP if the comms window can be terminated.

3.2 Functions

The library example functions are provided here, with short descriptions. Also refer to the actual firmware where the code is also well commented and explained.

3.2.1 IQS316_Settings

During initialization, an example setup of the IQS316 is performed. Naturally this will need to be adapted for each application, but gives a good guideline of an efficient and correct setup procedure. The steps performed during this setup are as follows:

1. To confirm that communication is working correctly, and also that the expected IQS316 IC is present, the Product and Version numbers are read from the device. If these are not as expected, an error hook is executed.
2. The reset is acknowledged and the SHOW_RESET bit is thus cleared. This will allow the master to monitor for any unexpected reset events, which would then require another setup of the IQS316 (this should not occur if power is stable, and the MCLR is not pulled low). It is also confirmed that the SHOW_RESET bit does clear, otherwise another debugging error hook is executed.

3. Now the required channels can be selected, with any unwanted channels disabled here.
4. The next section is tricky, since these settings need to be sent within a specific group. The ATI Multiplier settings and prox/touch thresholds are cycle specific. To allow the communication to cycle through all groups, the SKIP_CONVERSION bit must be set. This overrides the natural charging sequence, and forces the device to cycle through all groups, for easy parameter setup. Once all required groups have been configured, the SKIP_CONVERSION setting must be disabled.
5. The Prox/Touch range settings are done, completing the threshold configuration.
6. Now the ATI is configured. Firstly for ProxMode the ATI Target is configured. Then the routine is started and the firmware waits for the ATI_Busy flag to clear, indicating that the process is complete. The system mode is changed to TouchMode, and again the target is configured, and the ATI routine is executed for these channels.
7. No further settings are configured, but if required it is recommended to add settings such as Low-Power and EventMode at the end of the settings function.

Listing 1. IQS316_Settings Function (I^{2}C and SPI)

```
void
IQS316_Settings(void)
{
   uint8_t ui8StartGroup, ui8CurrentGroup;
   uint8_t ui8ProdNo, ui8VersionNo;
   uint8_t ui8DataArray[20];
   //
   // Confirm comms are working correctly, and also that expected IQS316
   // IC version is used.  Do this by reading back the Product and Version
   // numbers from the IQS316
   //
   IQS316_Read(PROD_NUM, ui8DataArray, 2);
   ui8ProdNo = ui8DataArray[0];
   ui8VersionNo = ui8DataArray[1];
   IQS316_End_Comms_Window();

#ifdef DEBUG_IQS316
   if((ui8ProdNo != 27) || (ui8VersionNo != 1))
   {
      // Error condition, handle this here
      // (fix comms or get correct IQS316 version)
      //
      while(1);
   }
#endif
   //
   // Acknowledge the reset by sending an ACK_RESET to the IQS316.  This will
   // clear the SHOW_RESET bit in UI_FLAGS0 register.  From here on further, if
   // this SHOW_RESET bit ever becomes set, we know an unexpected reset has
   // occurred on the IQS316, and we should repeat the setup
   //
   ui8DataArray[0] = (ACK_RESET | LTN_DISABLE | WDT_DISABLE);
   IQS316_Write(PROX_SETTINGS_2, ui8DataArray, 1);
   IQS316_End_Comms_Window();

#ifdef DEBUG_IQS316
   IQS316_Read(UI_FLAGS0, ui8DataArray, 1);
   IQS316_End_Comms_Window();
```

```
  if((ui8DataArray[0] & SHOW_RESET) != 0)
  {
    // The show reset bit should be cleared after writing the ACK_RESET
    // previously.  Check write procedures, and make sure comms window is
    // closed after sending ACK_RESET.
    while(1);
  }
#endif
  //
  // IQS316 Application specific SETUP
  // 1 - CHANNEL SETUP
  //
  ui8DataArray[0] = 0x03;       // CHAN_ACTIVE0
  ui8DataArray[1] = 0x0F;       // CHAN_ACTIVE1
  ui8DataArray[2] = 0x0F;       // CHAN_ACTIVE2
  ui8DataArray[3] = 0x0F;       // CHAN_ACTIVE3
  ui8DataArray[4] = 0x0F;       // CHAN_ACTIVE4

  IQS316_Write(CHAN_ACTIVE0, ui8DataArray, 5);
  IQS316_End_Comms_Window();
  //
  // 2 - Setup ATI and thresholds (settings which must be sent in specific
  // comms window - depending which group is active)
  //
  IQS316_Read(GROUP_NUM, ui8DataArray, 1);
  ui8StartGroup = ui8DataArray[0];
  //
  // Enable skip conversions, so that IQS316 cycles through the the groups
  // 0, 1, 2, 3, 4, 0, 1, ....   to allow configuring settings which must be
  // setup while in a specific cycle.
  //
  ui8DataArray[0] = (SKIP_CONV | LTN_DISABLE | WDT_DISABLE);
  IQS316_Write(PROX_SETTINGS_2, ui8DataArray, 1);
  ui8CurrentGroup = ui8StartGroup;

  do
  {
    switch(ui8CurrentGroup)
    {
      case 0:
      {
        // ATI C and ATI I settings
        //
        ui8DataArray[0] = 0x00;         // ATI_MULT1
        ui8DataArray[1] = 0x00;         // ATI_MULT2
        IQS316_Write(ATI_MULT1, ui8DataArray, 2);
        //
        // Set thresholds (in upper nibble of LTA)
        // NOTE: this will overwrite the LTA value also, but auto-ATI
        // will be done later, which will reseed the LTAs correctly
        //
        ui8DataArray[0] = PROX_THRES_8;  // LTA_04_HI
        ui8DataArray[1] = 0x00;         // low byte - irrelevant
        ui8DataArray[2] = PROX_THRES_8;  // LTA_15_HI
        ui8DataArray[3] = 0x00;         // low byte - irrelevant
        ui8DataArray[4] = PROX_THRES_8;  // LTA_26_HI
        ui8DataArray[5] = 0x00;         // low byte - irrelevant
        ui8DataArray[6] = PROX_THRES_8;  // LTA_37_HI
        IQS316_Write(LTA_04_HI, ui8DataArray, 7);
        break;
```

```
}
case 1:
{
   ui8DataArray[0] = 0x00;        // ATI_MULT1
   ui8DataArray[1] = 0x00;        // ATI_MULT2
   IQS316_Write(ATI_MULT1, ui8DataArray, 2);

   ui8DataArray[0] = PROX_THRES_20 | TOUCH_THRES_3_16;  // LTA_04_HI
   ui8DataArray[1] = 0x00;        // low byte - irrelevant
   ui8DataArray[2] = PROX_THRES_20 | TOUCH_THRES_3_16;  // LTA_15_HI
   ui8DataArray[3] = 0x00;        // low byte - irrelevant
   ui8DataArray[4] = PROX_THRES_20 | TOUCH_THRES_3_16;  // LTA_26_HI
   ui8DataArray[5] = 0x00;        // low byte - irrelevant
   ui8DataArray[6] = PROX_THRES_20 | TOUCH_THRES_3_16;  // LTA_37_HI
   IQS316_Write(LTA_04_HI, ui8DataArray, 7);
   break;
}
case 2:
{
   ui8DataArray[0] = 0x00;        // ATI_MULT1
   ui8DataArray[1] = 0x00;        // ATI_MULT2
   IQS316_Write(ATI_MULT1, ui8DataArray, 2);

   ui8DataArray[0] = PROX_THRES_20 | TOUCH_THRES_3_16;  // LTA_04_HI
   ui8DataArray[1] = 0x00;        // low byte - irrelevant
   ui8DataArray[2] = PROX_THRES_20 | TOUCH_THRES_3_16;  // LTA_15_HI
   ui8DataArray[3] = 0x00;        // low byte - irrelevant
   ui8DataArray[4] = PROX_THRES_20 | TOUCH_THRES_3_16;  // LTA_26_HI
   ui8DataArray[5] = 0x00;        // low byte - irrelevant
   ui8DataArray[6] = PROX_THRES_20 | TOUCH_THRES_3_16;  // LTA_37_HI
   IQS316_Write(LTA_04_HI, ui8DataArray, 7);
   break;
}
case 3:
{
   ui8DataArray[0] = 0x00;        // ATI_MULT1
   ui8DataArray[1] = 0x00;        // ATI_MULT2
   IQS316_Write(ATI_MULT1, ui8DataArray, 2);

   ui8DataArray[0] = PROX_THRES_20 | TOUCH_THRES_3_16;  // LTA_04_HI
   ui8DataArray[1] = 0x00;        // low byte - irrelevant
   ui8DataArray[2] = PROX_THRES_20 | TOUCH_THRES_3_16;  // LTA_15_HI
   ui8DataArray[3] = 0x00;        // low byte - irrelevant
   ui8DataArray[4] = PROX_THRES_20 | TOUCH_THRES_3_16;  // LTA_26_HI
   ui8DataArray[5] = 0x00;        // low byte - irrelevant
   ui8DataArray[6] = PROX_THRES_20 | TOUCH_THRES_3_16;  // LTA_37_HI
   IQS316_Write(LTA_04_HI, ui8DataArray, 7);
   break;
}
case 4:
{
   ui8DataArray[0] = 0x00;        // ATI_MULT1
   ui8DataArray[1] = 0x00;        // ATI_MULT2
   IQS316_Write(ATI_MULT1, ui8DataArray, 2);

   ui8DataArray[0] = PROX_THRES_20 | TOUCH_THRES_3_16;  // LTA_04_HI
   ui8DataArray[1] = 0x00;        // low byte - irrelevant
   ui8DataArray[2] = PROX_THRES_20 | TOUCH_THRES_3_16;  // LTA_15_HI
   ui8DataArray[3] = 0x00;        // low byte - irrelevant
   ui8DataArray[4] = PROX_THRES_20 | TOUCH_THRES_3_16;  // LTA_26_HI
   ui8DataArray[5] = 0x00;        // low byte - irrelevant
```

```
        ui8DataArray[6] = PROX_THRES_20 | TOUCH_THRES_3_16;  // LTA_37_HI
        IQS316_Write(LTA_04_HI, ui8DataArray, 7);
        break;
      }
    }
    IQS316_End_Comms_Window();

    IQS316_Read(GROUP_NUM, ui8DataArray, 1);
    ui8CurrentGroup = ui8DataArray[0];
  } while (ui8CurrentGroup != ui8StartGroup);
  //
  // Now Group specific settings are done, so disable the skip conversions
  //
  ui8DataArray[0] = (LTN_DISABLE | WDT_DISABLE);
  IQS316_Write(PROX_SETTINGS_2, ui8DataArray, 1);
  //
  // Set the high/low settings for prox and touch thresholds
  //
  ui8DataArray[0] = (PROX_THRES_RANGE | ND);
  IQS316_Write(UI_SETTINGS0, ui8DataArray, 1);
  //
  // Set ATI Target - For Prox Mode
  //
  ui8DataArray[0] = 0x03;
  ui8DataArray[1] = 0x20;
  IQS316_Write(AUTO_ATI_TARGET_HI, ui8DataArray, 2);
  IQS316_End_Comms_Window();
  //
  // Perform automated ATI routine (to setup ATI Compensation values)
  // NOTE: ATI_MODE already set to ProxMode, no need to configure.
  //
  ui8DataArray[0] = CXVSS | HALT0 | AUTO_ATI | CXDIV1;
  IQS316_Write(PROX_SETTINGS_1, ui8DataArray, 1);
  IQS316_End_Comms_Window();
  //
  // Read ATI Busy flag until it clears, then ProxMode ATI is done
  //
  do
  {
    IQS316_Read(UI_FLAGS0, ui8DataArray, 1);
    IQS316_End_Comms_Window();

  } while ((ui8DataArray[0] & ATI_BUSY) != 0);
  //
  // Perform ATI for Touch Mode
  // Set ATI_MODE to Touch
  //
  ui8DataArray[0] = ATI_MODE | PROX_THRES_RANGE | ND;
  IQS316_Write(UI_SETTINGS0, ui8DataArray, 1);
  IQS316_End_Comms_Window();
  //
  // Set ATI Target - For Touch Mode
  //
  ui8DataArray[0] = 0x03;
  ui8DataArray[1] = 0x20;
  IQS316_Write(AUTO_ATI_TARGET_HI, ui8DataArray, 2);
  IQS316_End_Comms_Window();
  //
  // Perform automated ATI routine (to setup ATI Compensation values)
  //
  ui8DataArray[0] = CXVSS | HALT0 | AUTO_ATI | CXDIV1;
```

```
    IQS316_Write(PROX_SETTINGS_1, ui8DataArray, 1);
    IQS316_End_Comms_Window();
    //
    // Read ATI Busy flag until it clears, then ATI is done
    //
    do
    {
      IQS316_Read(UI_FLAGS0, ui8DataArray, 1);
      IQS316_End_Comms_Window();

    } while ((ui8DataArray[0] & ATI_BUSY) != 0);
    //
    // Now setup the advanced settings as required by the design, such as the
    // following:  Low-Power, charging mode, eventMode
    //
}
```

3.2.2 IQS316_Refresh_Data

New data from each cycle is read into the PIC master device. Each time the following bytes are read: UI_FLAGS0, PROX_STATUS, TOUCH_STATUS, HALT_STATUS and GROUP_NUM. Note that HALT_STATUS is not needed, but it is much faster to read through this byte than to reconfigure a new read for a specific address.

The reset bit is monitored to catch any unexpected reset events. If such a situation is seen, then the IQS316 is reconfigured.

The elements of the IQS316 structure are updated accordingly, with the GROUP_NUM used to identify which channels data must be updated.

Listing 2. IQS316_Refresh_Data Function (I^2C and SPI)

```
void
IQS316_Refresh_Data(void)
{
    uint8_t ui8CurrentGroup, ui8TempTouch, ui8TempProx;
    uint8_t ui8DataArray[5], ui8TempUIFlags0;

    IQS316_ReadCurrentAddress(ui8DataArray, 5);
    //
    // Comms window is now ended. Note if other data is required then obtain
    // this before ending the window
    //
    IQS316_End_Comms_Window();
    //
    // Temporarily store the received data
    //
    ui8TempUIFlags0 = ui8DataArray[0];
    ui8TempProx =ui8DataArray[1];
    ui8TempTouch =ui8DataArray[2];
    ui8CurrentGroup =ui8DataArray[4];
    //
    // Make sure an unexpected reset has not occurred
    //
    if((ui8TempUIFlags0 & SHOW_RESET) != 0)
    {
      // handle reset here, suggestion is to repeat IQS316 init
      //
      IQS316_Settings();
    }
```

```
  //
  // Here an example is given of how the data can be placed into an IQS316
  // structure.  This is purely for example purposes
  //
  if(ui8CurrentGroup == 0)
  {
    if(ui8TempProx == 0)
    {
      IQS316.prox_detected = 0;
    }
    else
    {
      IQS316.prox_detected = 1;
    }
  }
  else
  {
    // Update the specific groups data
    //
    switch(ui8CurrentGroup)
    {
      case 1:
        IQS316.prox4_11 &= 0xF0;
        IQS316.touch4_11 &= 0xF0;

        IQS316.prox4_11 |= (ui8TempProx & 0x0F);
        IQS316.touch4_11 |= (ui8TempTouch & 0x0F);
        break;
      case 2:
        IQS316.prox4_11 &= 0x0F;
        IQS316.touch4_11 &= 0x0F;

        IQS316.prox4_11 |= ((ui8TempProx & 0x0F) << 4);
        IQS316.touch4_11 |= ((ui8TempTouch & 0x0F) << 4);
        break;
      case 3:
        IQS316.prox12_19 &= 0xF0;
        IQS316.touch12_19 &= 0xF0;

        IQS316.prox12_19 |= (ui8TempProx & 0x0F);
        IQS316.touch12_19 |= (ui8TempTouch & 0x0F);
        break;
      case 4:
        IQS316.prox12_19 &= 0x0F;
        IQS316.touch12_19 &= 0x0F;

        IQS316.prox12_19 |= ((ui8TempProx & 0x0F) << 4);
        IQS316.touch12_19 |= ((ui8TempTouch & 0x0F) << 4);
        break;
    }
  }
}
```

3.2.3 IQS316_Process_Data

A short example function is added here, and converts the key pressed to a binary number which is displayed on 4 I/O pins.

Listing 3. IQS316_Process_Data Function (I²C and SPI)

```
void
IQS316_Process_Data(void)
{
   uint8_t i, ui8ButtonNumber;
   uint16_t ui16BitMask, ui16TempTouch;
   //
   // Place code here to process data available in the IQS316 structure.
   // this example places the binary value of the pressed button on 4 LEDs
   //
   ui16BitMask = 0x0001;
   //
   // place the touch bits 4 to 19 into a word
   //
   ui16TempTouch = (uint16_t)IQS316.touch4_11 | (((uint16_t)IQS316.touch12_19)<<8);

   for(i = 0 ; i < 16 ; i++)
   {
      if((ui16TempTouch & ui16BitMask) != 0)
         ui8ButtonNumber = i;
     ui16BitMask =  ui16BitMask<<1;
   }
   // Display binary value on PD7..PD4
   //
   ui8ButtonNumber = (LATD & 0x0F) | (ui8ButtonNumber<<4);

   LATD = ui8ButtonNumber;
}
```

3.2.4 Main Function (I²C and SPI)

The Main function sets up the hardware by writing all required initialization data to the controller. After initialization the function runs the infinite loop to retrieve data from the IQS316 and to process the data as required.

Listing 4. Main Function

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

   while(1)
   {
      // Get data from latest comms window
      //
      IQS316_Refresh_Data();
      //
      // Process this new data accordingly
      //
      IQS316_Process_Data();
   }
}
```

3.2.5 Comms_init

The Comms_Init function sets the registers in the PIC18F4550 for either SPI or I²C communication.

At the end the IQS316 MCLR is released to allow the IQS316 to come out of reset.

Listing 5. Comms_Init Function (I²C)

```
void
Comms_init()
{
   TRISB = TRISB | 0x03;        //set TRISB<1:0> SDA and SCL
   TRISA = TRISA | 0x02;        //I2C ready line input

   PIR1bits.SSPIF = 0;          //clear I2C interrupt flag

   SSPADD = 0x1C;               //settings for I2C frequency - 416kHz
   SSPSTAT |= 0x80;             //slew rate control for high speed (400kHz)

   SSPCON1 = 0x28;              //enables I2C module on the PIC18F4550

   LATB = LATB | 0x04;          //IQS316 MCLR High
}
```

Listing 6. Comms_Init Function (SPI)

```
void
Comms_init()
{
   TRISB |= 0x01;               //SOMI input on B0
   TRISB &= 0xFD;               //SCK output on B1
   TRISA |= 0x01;               //RDY input on RA0
   TRISA &= 0xFD;               //SS output on RA1
   TRISC &= 0x7F;               //MOSI output on RC7

   SSPSTAT = 0x80;
   SSPCON1 = 0x32;              //enables SSP, SCK idle high

   LATA |= 0x02;                //Set SS high

   LATB = LATB | 0x04;          //IQS316 MCLR High
}
```

HIGHER LEVEL COMMS READ AND WRITE FUNCTIONS:

The higher level communication functions (reading and writing data) can be called in any sequence. All of them will wait for RDY to be set before performing the data read/write. None of them terminate the communication window. Thus numerous read and write functions can be called, and then the window can be ended when required.

3.2.6 IQS316_Read

The IQS316_Read function requires an address from which to read as parameter. Also an array is sent where the read data is to be placed. The final parameter indicates how many bytes are to be read during this data read transaction. The communication window is NOT closed after this read function.

Listing 7. IQS316_Read Function (I²C)

```
void
```

```
IQS316_Read(uint8_t ui8Address, uint8_t *ui8Data, uint8_t ui8Length)
{
  uint8_t i;
  //
  // Wait for RDY and give I2C START (could be repeated start also)
  //
  CommsIQS316_start();
  //
  // Initiate comms by sending device address plus WRITE
  //
  CommsIQS316_send((IQS316_ADDR << 1) + 0x00);
  //
  // Send the address of where to write the data
  //
  CommsIQS316_send(ui8Address);
  //
  // Repeated start
  //
  CommsIQS316_start();
  //
  // Send device address plus READ
  //
  CommsIQS316_send((IQS316_ADDR << 1) + 0x01);
  //
  // Read in all the required data bytes, last read ends with a NACK
  //
  for(i = 0 ; i < ui8Length ; i++)
  {
    if(i == (ui8Length-1))
      ui8Data[i] = CommsIQS316_read_nack();
    else
      ui8Data[i] = CommsIQS316_read_ack();
  }
}
```

Listing 8. IQS316_Read Function (SPI)

```
void
IQS316_Read(uint8_t ui8Address, uint8_t *ui8Data, uint8_t ui8Length)
{
  uint8_t ui8Header, i;
  //
  // Writing a 0x01 on MOSI requests a READ operation
  //
  ui8Header = CommsIQS316_send(0x01);
  //
  // Make sure the header (0xFF) is received from the slave, otherwise error
  //
  if (ui8Header != 0xFF)
  {
    // Handle the error here, the 0xFF header should always be received
    // first. So this should not be called, during debugging use this
    // function to correct any comms issues.
    //
    Comms_Error();
  }
  //
  // Send specific address to read from (ignore returned data)
  //
  CommsIQS316_send(ui8Address);
  //
```

```
   //
   // Read in as many bytes as specified
   //
   for(i = 0 ; i < ui8Length ; i++)
   {
      ui8Data[i] = CommsIQS316_send(0xFE);
   }
   //
   // End this read session (not the comms window)
   //
   CommsIQS316_send(0xFF);
}
```

3.2.7 IQS316_ReadCurrentAddress

The IQS316_ReadCurrentAddress function does NOT require an address from which to read since it is assumed that the address pointer is already correct. The function does require an array of where the read data is to be placed as well as a parameter indicating how many bytes are to be read during this data read transaction. The communication window is NOT closed after this read function.

This function is often used when retrieving the data from each new cycle, since the default address pointer is already correctly configured at the start of each communication session.

Listing 9. IQS316_ReadCurrentAddress Function (I^2C)

```
void
IQS316_ReadCurrentAddress(uint8_t *ui8Data, uint8_t ui8Length)
{
   uint8_t i;
   //
   // Wait for RDY and give I2C START (could be repeated start also)
   //
   CommsIQS316_start();
   //
   // Send device address plus READ
   //
   CommsIQS316_send((IQS316_ADDR << 1) + 0x01);
   //
   // Read in all the required data bytes, last read ends with a NACK
   //
   for(i = 0 ; i < ui8Length ; i++)
   {
      if(i == (ui8Length-1))
         ui8Data[i] = CommsIQS316_read_nack();
      else
         ui8Data[i] = CommsIQS316_read_ack();
   }
}
```

Listing 10. IQS316_Read Function (SPI)

```
void
IQS316_ReadCurrentAddress(uint8_t *ui8Data, uint8_t ui8Length)
{
   uint8_t ui8Header, i;
   //
   // Writing a 0x01 on MOSI requests a READ operation
   //
   ui8Header = CommsIQS316_send(0x01);
   //
```

```
   // Make sure the header (0xFF) is received from the slave, otherwise error
   //
   if (ui8Header != 0xFF)
   {
      Comms_Error();
   }
   //
   // Read in as many bytes as specified
   //
   for(i = 0 ; i < ui8Length ; i++)
   {
      ui8Data[i] = CommsIQS316_send(0xFE);
   }
   //
   // End this read session (not the comms window)
   //
   CommsIQS316_send(0xFF);
}
```

3.2.8 IQS316_Write

The IQS316_Write function requires the same parameters as the read function. The address where to write, the data array containing the data bytes to write, and the number of bytes that must be written must be provided. Again the communication window is not closed after the write transaction.

Listing 11. IQS316_Write Function (I^2C)

```
void
IQS316_Write(uint8_t ui8Address, uint8_t *ui8Data, uint8_t ui8Length)
{
   uint8_t i;
   //
   // Wait for RDY and give I2C START (could be repeated start also)
   //
   CommsIQS316_start();
   //
   // Initiate comms by sending device address plus WRITE
   //
   CommsIQS316_send((IQS316_ADDR << 1) + 0x00);
   //
   // Send the address of where to write the data
   //
   CommsIQS316_send(ui8Address);
   //
   // Write in all the required data bytes
   //
   for(i = 0 ; i < ui8Length ; i++)
   {
      CommsIQS316_send(ui8Data[i]);
   }
}
```

Listing 12. IQS316_Write Function (SPI)

```
void
IQS316_Write(uint8_t ui8Address, uint8_t *ui8Data, uint8_t ui8Length)
{
  uint8_t ui8Header, i;
   //
```

```
   // Writing a 0x00 on MOSI requests a WRITE operation
   //
   ui8Header = CommsIQS316_send(0x00);
   //
   // Make sure the header (0xFF) is received from the slave, otherwise error
   //
   if (ui8Header != 0xFF)
   {
      Comms_Error();
   }
   //
   // Write in as many bytes as specified
   //
   for(i = 0 ; i < ui8Length ; i++)
   {
      CommsIQS316_send(ui8Address++);
            CommsIQS316_send(ui8Data[i]);
   }
   //
   // End this read session (not the comms window)
   //
   CommsIQS316_send(0xFF);
}
```

3.2.9 IQS316_End_Comms_Window

To allow the IQS316 to exit communication mode, and perform sensing and processing, the communication window must be correctly ended. In I^2C this is done by sending an I^2C STOP, and in SPI during a WRITE transaction, an address of 0xFE must be sent. Both of these functions wait for the RDY to change to the LOW state. This is just precautionary since on fast microcontrollers, a following read/write could already start to process before the RDY has transitioned from HIGH to LOW. It would then look as though a new communication window is active, where actually it is the old window that is busy being closed.

Listing 13. IQS316_End_Comms_Window Function (I^2C)

```
void
IQS316_End_Comms_Window(void)
{
   // To end the comms window you send an I2C STOP condition
   //
   CommsIQS316_stop();
   //
   // Wait for RDY to go LOW
   //
   while (RDY != 0)
   {}
}
```

Listing 14. IQS316_End_Comms_Window Function (SPI)

```
void
IQS316_End_Comms_Window(void)
{
  uint8_t ui8Header;
   //
   // Writing a 0x00 on MOSI requests a WRITE operation
   //
   ui8Header = CommsIQS316_send(0x00);
```

```
    //
    // Make sure the header (0xFF) is received from the slave, otherwise error
    //
    if (ui8Header != 0xFF)
    {
      Comms_Error();
    }
    // Write a 0xFE into address timeslot to end comms window
    //
    CommsIQS316_send(0xFE);
    //
    // Wait for RDY to go LOW
    //
    while (RDY != 0)
    {}
}
```

3.2.10 Comms_Error

The Comms_Error function can be called from any of the SPI functions if an unexpected value is received. During developmental stages, this function may be used to indicate that an error has occurred during communication. In final stages it would probably be preferred to simply restart the system in the case that an error is detected.

Listing 15. Comms_Error Function (SPI)

```
void Comms_Error(void)
{
    //
    // Place error routine code here
    //
    while (1)
    {}
}
```

LOWER LEVEL COMMS READ AND WRITE FUNCTIONS:

The lower level functions are specific to the microcontroller used for the application. These functions are thus specific to the PIC18F4550. These functions are provided below with short explanations.

NOTE: If the designer can reproduce the functionality of these lower level functions EXACTLY the same when implementing on a different controller, then the rest of the higher level firmware can remain unchanged.

3.2.11 I^{2}C byte write

Send a byte and wait for the acknowledge.

Listing 16. CommsIQS316_send (I^{2}C)

```
void
CommsIQS316_send(uint8_t send_data)
{
        SSPBUF = send_data;

        while (PIR1bits.SSPIF == 0)
        {}
        PIR1bits.SSPIF = 0;                //clear flag
```

```
        while (SSPCON2bits.ACKSTAT == 1)        //verify IQS316 acknowledge
        {}
}
```

3.2.12 Read with NACK

Read a byte, and indicate it is the last byte to be read by sending a NACK after the byte.

Listing 17. CommsIQS316_read_nack (I^{2}C)

```
uint8_t
CommsIQS316_read_nack(void)
{
        unsigned char temp;

        SSPCON2bits.RCEN = 1;
        while (PIR1bits.SSPIF == 0)
        {}
        PIR1bits.SSPIF = 0;             //clear flag

        while (SSPSTATbits.BF == 0)
        {}
        temp = SSPBUF;

        SSPCON2bits.ACKDT = 1;
        SSPCON2bits.ACKEN = 1;

        while (PIR1bits.SSPIF == 0)
        {}
        PIR1bits.SSPIF = 0;             //clear flag

        while (SSPCON2bits.ACKEN == 1) {}

        return temp;
}
```

3.2.13 Read with ACK

Read a byte, and indicate more bytes are to be read by sending an ACK after the byte.

Listing 18. CommsIQS316_read_ack (I^{2}C)

```
uint8_t
CommsIQS316_read_ack(void)
{
        unsigned char temp;

        SSPCON2bits.RCEN = 1;
        while (PIR1bits.SSPIF == 0)
        {}
        PIR1bits.SSPIF = 0;             //clear flag

        while (SSPSTATbits.BF == 0)
        {}
        temp = SSPBUF;

        SSPCON2bits.ACKDT = 0;
        SSPCON2bits.ACKEN = 1;

        while (PIR1bits.SSPIF == 0)
```

```
        {}
        PIR1bits.SSPIF = 0;                 //clear flag

        while (SSPCON2bits.ACKEN == 1) {}

        return temp;
}
```

3.2.14 I²C START

Create an I²C start event.

Listing 19. CommsIQS316_start (I²C)

```
void
CommsIQS316_start(void)
{
        while (RDY == 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)
        {}
}
```

3.2.15 I²C STOP

Create an I²C stop event.

Listing 20. CommsIQS316_stop (I²C)

```
void
CommsIQS316_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)
        {}
}
```

3.2.16 SPI Receive/Transmit

The SPI protocol is considerably more basic, and the only lower level function required is the receive transmit function. This function receives a data byte on the SOMI line, and at the same time transmits a data byte on the MOSI line.

Listing 21. CommsIQS316_RxTx (SPI)

```
uint8_t
CommsIQS316_RxTx(uint8_t ui8SendData)
{
```

```
    uint8_t ui8ReceiveData;
    //
    // Wait for ready signal
    //
    while(RDY == 0)
    {}
    //
    // Select IQS316 by pulling SS low
    //
    LATA = LATA & 0xFD;
    //
    // reset transmission complete flag
    //
    PIR1bits.SSPIF = 0;
    //
    // Perform read
    //
    ui8ReceiveData = SSPBUF;
    //
    // Initiate transmission
    //
    SSPBUF = ui8SendData;
    //
    // Wait for transmission complete flag
    //
    while (PIR1bits.SSPIF == 0)
    {}
    //
    // Temp store received byte
    //
    ui8ReceiveData = SSPBUF;
    //
    // Release SS line on IQS316
    //
    LATA = LATA | 0x02;

    return ui8ReceiveData;
}
```

3.2.17 Constant Declarations

The IQS316 Memory map is declared in IQS316.h. These constants can be used to easily configure the registers. Bit definitions are also provided here.

Listing 22. IQS316.h Memory Map Constants (SPI)

```
#define PROD_NUM                0x00
#define VERSION_NUM             0x01

#define UI_FLAGS0               0x10

#define PROX_STAT               0x31
#define TOUCH_STAT              0x35
#define HALT_STAT               0x39
#define GROUP_NUM               0x3D

#define CUR_SAM_04_HI           0x42
#define CUR_SAM_04_LO           0x43
#define CUR_SAM_15_HI           0x44
#define CUR_SAM_15_LO           0x45
#define CUR_SAM_26_HI           0x46
```

```
#define CUR_SAM_26_LO          0x47
#define CUR_SAM_37_HI          0x48
#define CUR_SAM_37_LO          0x49

#define LTA_04_HI              0x83
#define LTA_04_LO              0x84
#define LTA_15_HI              0x85
#define LTA_15_LO              0x86
#define LTA_26_HI              0x87
#define LTA_26_LO              0x88
#define LTA_37_HI              0x89
#define LTA_37_LO              0x8A

#define UI_SETTINGS0           0xC4
#define POWER_SETTINGS         0xC5
#define PROX_SETTINGS_1        0xC6
#define PROX_SETTINGS_2        0xC7
#define ATI_MULT1              0xC8
#define ATI_MULT2              0xC9
#define ATI_C0                 0xCA
#define ATI_C1                 0xCB
#define ATI_C2                 0xCC
#define ATI_C3                 0xCD
#define SHLD_SETTINGS          0xCE
#define INT_CAL_SETTINGS       0xCF
#define PM_CX_SELECT           0xD0
#define DEFAULT_COMMS_PTR      0xD1
#define CHAN_ACTIVE0           0xD2
#define CHAN_ACTIVE1           0xD3
#define CHAN_ACTIVE2           0xD4
#define CHAN_ACTIVE3           0xD5
#define CHAN_ACTIVE4           0xD6
#define CHAN_RESEED0           0xD7
#define CHAN_RESEED1           0xD8
#define CHAN_RESEED2           0xD9
#define CHAN_RESEED3           0xDA
#define CHAN_RESEED4           0xDB
#define AUTO_ATI_TARGET_HI     0xDC
#define AUTO_ATI_TARGET_LO     0xDD

#define DIRECT_ADDR_RW         0xFC
#define DIRECT_DATA_RW         0xFD


// BIT DEFINITIONS

// UI_FLAGS0
#define SHOW_RESET             0x80
#define MODE_INDICATOR         0x40
// unused                      0x20
// unused                      0x10
// unused                      0x08
#define ATI_BUSY               0x04
#define RESEED_BUSY            0x02
#define NOISE                  0x01

// TOUCH THRESHOLDS
// with touch LOW range selected
#define TOUCH_THRES_1_32       0x00
#define TOUCH_THRES_1_16       0x40
#define TOUCH_THRES_2_16       0x80
```

```
#define TOUCH_THRES_3_16            0xC0
// with touch HIGH range selected
#define TOUCH_THRES_4_16            0x00
#define TOUCH_THRES_6_16            0x40
#define TOUCH_THRES_8_16            0x80
#define TOUCH_THRES_10_16           0xC0

// PROX THRESHOLDS
// with prox LOW range selected
#define PROX_THRES_2                0x00
#define PROX_THRES_3                0x10
#define PROX_THRES_4                0x20
#define PROX_THRES_6                0x30
// with prox HIGH range selected
#define PROX_THRES_8                0x00
#define PROX_THRES_16               0x10
#define PROX_THRES_20               0x20
#define PROX_THRES_30               0x30

// UI_SETTINGS0
#define RESEED                      0x80
#define ATI_MODE                    0x40
#define PROX_THRES_RANGE            0x20
#define TOUCH_THRES_RANGE           0x10
#define FORCE_PROX_THRES_MODE       0x08
#define FORCE_TOUCH_THRES_MODE      0x04
#define ND                          0x02
// unused                           0x01

// POWER_SETTINGS
// unused                           0x80
// unused                           0x40
// unused                           0x20
// unused                           0x10
#define SLEEP                       0x08
#define MAIN_OSC                    0x04
#define LP1                         0x02
#define LP0                         0x01

// PROX_THRES_SETTINGS_1
#define CXVSS                       0x80
#define ZC_EN                       0x40
#define HALT1                       0x20
#define HALT0                       0x10
#define AUTO_ATI                    0x08
#define CXDIV2                      0x04
#define CXDIV1                      0x02
#define CXDIV0                      0x01

// PROX_THRES_SETTINGS_2
// unused                           0x80
#define SHIELD_EN                   0x40
#define STOP_COMMS                  0x20
#define ACK_RESET                   0x10
#define SKIP_CONV                   0x08
#define ACF_DISABLE                 0x04
#define LTN_DISABLE                 0x02
#define WDT_DISABLE                 0x01

// CX_CONFIG
#define CX_GPIO_1                   0x80
```

```
#define CX_GPIO_0        0x40
// unused                0x20
// unused                0x10
#define GROUP4           0x08
#define GROUP3           0x04
#define GROUP2           0x02
#define GROUP1           0x01
```

v0.02 Changes

- Corrected ATI C (ATI_MULT1) defaults
- Individual reseed bits defaults changed to ‘0’, since they are cleared after the first cycle for the applicable group.
- Added prox/touch defaults to bit definitions, and made default clearer in tables
- Added clarity that if Cx/GPIO selection is set to Cx, then TRIS register must be set to inputs.
- Updated Section 2.2
- Corrected Table 1.11
- Corrected PM_CX_SELECT bit definition

v0.03 Changes

- Added links to make ease document navigation

v0.04 Changes

- Major changes done to the Example I^2C and SPI firmware. All relevant sections updated.

	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 83035294 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.